THE EXPERT'S VOICE® IN ORACLE

Cost-Based Oracle Fundamentals

"The insights that Jonathan provides into the workings of the cost-based optimizer will make a DBA a better designer and a developer a better SQL coder. Both groups will become better troubleshooters."

—Thomas Kyte

Jonathan Lewis

Foreword by Thomas Kyte
Vice President (Public Sector), Oracle Corporation

Apress®

Cost-Based Oracle Fundamentals

Jonathan Lewis

Apress®

Cost-Based Oracle Fundamentals

ISBN (pbk): 1-59059-636-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Tony Davis
Technical Reviewers: Christian Antognini, Wolfgang Breitling
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Production Director and Project Manager: Grace Wong
Copy Edit Manager: Nicole LeClerc
Senior Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreaders: Lori Bring, Kim Burton, Nancy Sixsmith
Indexer: Valerie Perry
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

Contents at a Glance

Contents

Foreword

Arthur C. Clarke once wrote that "any sufficiently advanced technology is indistinguishable from magic." I believe that observation to be entirely accurate. Someone else later observed that "any technologist with sufficient knowledge is indistinguishable from a magician." With that in mind, what you have in your hands right now is a book on magic.

However, for as long as I've known the author of this book, Jonathan Lewis (some 11 years according to my research on Google going back over the newsgroup archives), he has never been content to accept "magic." He wants to know *why* something happens the way it does. So, fundamentally, his book is all about *understanding*:

> *understand v. understood, (-std) understanding, understands*
>
> *1. To perceive and comprehend the nature and significance of*
>
> *2. To know thoroughly by close contact or long experience with*

More precisely, it is all about understanding the Oracle cost based optimizer (CBO), how it works, and why it does what it does. Jonathan conveys to us his understanding of the Oracle CBO through practice and example, and with this understanding, with this knowledge, new options and solutions become available.

Put simply, the Oracle CBO is a mathematical model; you feed it inputs (queries, statistics), and it produces outputs (query plans). In order to use the CBO successfully, it is critical that you understand what these inputs are and how the CBO uses them. Consider the following question, one that I've tried to answer many times: What is the best way to collect statistics, what statistics should I gather? Seems simple enough, very straightforward—there should be an answer and there is, but it is no "one size fits all" answer. It depends on your circumstances, your data distribution, your queries, the type of system you have (transactional, data warehouse)—a whole host of factors—and it is only through an understanding of how the CBO works and of how these factors affect the CBO that you'll be able to answer the question for yourself.

My favorite chapter in this book is Chapter 7, which provides an excellent discussion on what histograms do, how the CBO uses them, and some of the myths that surround them (they are a frequently misunderstood input to the CBO). One of the reasons it is my favorite chapter is because I still remember *hearing* this chapter for the first time (not reading it, hearing it). It was about three years ago, at the NoCOUG (Northern California Oracle User Group) meeting. I attended Jonathan's *Histograms* session and, for the first time, felt that I truly *understood* how histograms worked in Oracle. Jonathan provided practical information, of immediate day-to-day use, and as a result I was in a much better position to answer the question just posed. In this book, you will find many such topics described in that same practical, immediately usable way.

The insights that Jonathan provides into the workings of the cost based optimizer will make a DBA a better designer and a developer a better SQL coder. Both groups will become better troubleshooters as a result. Jonathan's meticulous and abundant examples make the complex workings of the CBO understandable for the rest of us.

Time and time again, it has been proven that you can only use a tool safely and effectively once you understand how it works. This is true of software, hardware, and pretty much everything else in life. In which case, what you have in your hands is a book that will allow you to make safe and effective use of the tool that is the Oracle CBO.

Thomas Kyte

VP (Public Sector), Oracle Corporation

About the Author

JONATHAN LEWIS is a qualified teacher with a mathematics degree from Oxford University. Although his interest in computers came to light at the tender age of about 12—in the days when high-technology meant you used a keyboard, rather than a knitting needle to punch holes in cards—it wasn't until he was four years out of university that he moved into computing professionally. Apart from an initial year as an incompetent salesman, he has been self-employed his entire career in the computer industry.

His initiation into Oracle was on version 5.1 running on a PC, which he used to design and build a risk-management system for the crude-trading floor of one of the major oil companies. He had written the first version of the system using a PC program called dBase III—which did use tables and indexes and made some claims to being a relational database management system. Suddenly he found out what a proper Relational Database Management System ought to be able to do.

Since that day of revelation, Jonathan has focused exclusively on the use and abuse of the Oracle RDBMS. After three or four years of contract work helping to build large systems, he decided to move from the "contractor" market to the "consultant" market, and now shares his time evenly between short-term consultancy work, running seminars, and "research."

At the time of writing, he is one of the directors of the UK Oracle User Group (`www.ukoug.com`), and a regular contributor to their quarterly magazine and presenter at their Special Interest Groups (particularly the RDBMS and UNIX groups). Whenever the opportunity arises, he tries to find time for speaking to user groups outside the UK, sometimes as short evening events at the end of a day's consultancy work. He also maintains a web site (`www.jlcomp.demon.co.uk`) of articles and notes on things that you can do with the Oracle RDBMS.

Jonathan has been married to Diana (currently a primary school teacher after many years of being an accountant) for nearly 20 years, and has two children: Anna (actress and trumpet player) and Simon (rugby and saxophone player). Given the choice between working on a Saturday and supporting a rugby match or school concert—it's the school event that comes first.

About the Technical Reviewers

CHRISTIAN ANTOGNINI has focused on understanding how the Oracle database engine works since 1995. His main interests range from logical and physical database design, the integration of databases with Java applications, to the cost based optimizer, and basically everything else related to performance management and tuning. He is currently working as a senior consultant and trainer at Trivadis AG (`www.trivadis.com`) in Zürich, Switzerland. If he is not helping one of his customers to get the most out of Oracle, he is somewhere lecturing on optimization or new Oracle database features for developers.

Christian lives in Ticino, Switzerland, with his wife, Michelle, and their two children, Sofia and Elia. He spends a great deal of his spare time with his wonderful family and, whenever possible, reading books, enjoying a good movie, or riding one of his BMX bikes.

WOLFGANG BREITLING was born in Stuttgart, Germany, and studied mathematics, physics, and computer sciences at the University of Stuttgart. After graduating, he joined the QA department of IBM Germany's development laboratory in Böblingen. There he became one of a team of two to develop an operating system to test the hardware of the /370 model machines developed in the Böblingen lab.

Wolfgang's first direct foray into the field of performance evaluation/tuning was the development of a program to test the speed of individual operating codes of the /370 architecture. After IBM Germany, he worked as a systems programmer on IBM's hierarchical databases DL/1 and IMS in Switzerland before emigrating to his current home in Calgary, Canada. Following several years as systems programmer for IMS and later DB2 databases on IBM mainframes, he got on the project to implement Peoplesoft on Oracle.

In 1996, he became an independent consultant specializing in administering and tuning Peoplesoft on Oracle. In that capacity, Wolfgang has been engaged in several Peoplesoft installation and upgrade projects. The particular challenges in tuning Peoplesoft, with often no access to the SQL, motivated him to explore Oracle's cost based optimizer in an effort to better understand how it works and use that knowledge in tuning. He has shared the findings from this research in papers and presentations at IOUG, UKOUG, local Oracle user groups, and other conferences and newsgroups dedicated to Oracle performance topics.

Wolfgang has been married to his wife Beatrice for over 30 years and has two children, Magnus and Leonie. When he is not trying to decipher the CBO, Wolfgang enjoys canoeing and hiking in the Canadian prairies and Rocky Mountains.

Acknowledgments

First and foremost, I have to thank my wife and children for putting up with my distant expressions, long silences, and my habit of being physically present while mentally absent. They have been very patient with my mantra to "give me just five more minutes."

I would like to thank those Oracle experts and adventurers around the world who (knowingly or unknowingly) have extended my knowledge of the database. In particular, though not exclusively, I owe thanks to (in alphabetical order) Steve Adams, Wolfgang Breitling, Julian Dyke, K. Gopalakrishnan, Stephan Haisley, Anjo Kolk, Tom Kyte, James Morle, and Richmond Shee. Each of them has, at various times, supplied me with insights that have allowed me to open up whole new areas of investigation and understanding. Insights are so much more important than answers.

Christian Antognini and Wolfgang Breitling merit special thanks for reading the draft chapters, and for making a number of suggestions to improve the quality of the book. Any outstanding errors are, of course, all my own work.

There are many more individuals who deserve to be named as sources of ideas and information, but the more I name, the greater the injustice to those whose names I let slip. So let me cover them all by thanking those who post on the *comp.databases.oracle* newsgroups, those who post on the *Oracle-L* mailing list, and those who post on Oracle Corporation's *MetaLink* forums. And, of course, there are the members of the *Oak Table Network* who can be relied upon to provide entertaining and stimulating discussions.

I would also like to mention the audiences who attend my seminar *Optimizing Oracle: Performance by Design*. At the end of each of these three-day events, I always have several new questions to answer, points to clarify, and strange cases to investigate.

The most important part of what I do is to deal with production database systems, so I also want to thank the many companies who have decided to call me in for a few days to examine their problems, validate their designs, or just give them some guidelines. Without continuous input and challenge from real users, real requirements, and real applications, it would be impossible to write a book that could be of practical benefit to its readers.

Finally, I owe a big thank you to the Apress editorial and production teams who worked incredibly hard to beat a tough deadline: Tony Davis, Ami Knox, Katie Stence, and Grace Wong.

Introduction

When I wrote *Practical Oracle 8i*, I said in the foreword that "if you write a technical book about Oracle, then it will be out of date by the time you've finished writing it." Addison-Wesley published the book about the same time that Larry Ellison announced the official release of Oracle 9*i*. Three weeks after publication, I got the first e-mail message asking me if I had plans for a 9*i* version of the book.

From that day onwards, I have resisted all requests for an "upgrade," on the basis that (a) it was far too much hard work, (b) I would have to spend a couple of years using Oracle 9*i* before I thought I could put some useful new information into a book, and (c) it would still be just the same book with a few small changes.

So there I was: I started writing in September 2003 (yes really, 2003; it took me 22 months to write this volume), exactly four years after I decided to write *Practical Oracle 8i*. (Remember September 1999, when nobody wanted to employ an Oracle specialist unless they could debug 40-year-old COBOL?) I had had enough exposure to Oracle 9*i* to make it worth writing a new book. Of course, in those four years there had been several upgrades to Oracle 8*i* (finishing at 8.1.7.4), two major releases of Oracle 9*i*, and, as I started writing, Oracle 10*g* was launched at Oracle World. I was about ready to write *Practical Oracle 9i*—just too late.

In fact, just as I finished writing this book (in June 2005), the Linux port for 10*g* Release 2 was made available on OTN! So the first thing you can do with this book is to start running my test cases against 10*g*R2 to see how many things have changed.

Instead of producing an upgrade for *Practical Oracle 8i*, I've embarked on a project to write up all I know about cost-based optimization. It sounded like a very easy project to start with—I can talk for hours about things that the optimizer does, and why it does them; all I had to do was write it down.

Unfortunately, the task has turned out to be a lot harder than I expected. Pouring out words is easy—producing a well-structured book that will be useful is a completely different matter. Show me something that the CBO has done and I can explain why—perhaps after creating and testing a couple of theories. Ask me to give someone else enough generic information about the optimizer to allow them to do the same, and it's a completely different issue.

Eventually, I managed to produce a framework for the task, and realized that I had to write at least three books: the fundamentals, some enhanced stuff, and all the peripherals. The book you are holding covers just the fundamentals of cost-based optimization.

Why Bother?

Why do we want to know how the CBO works? Because when we hit a problem where the optimizer produces a very bad execution plan, we want to understand what the problem is so that we can supply the right fix.

Of course, we may be able to fix that one problem by adding some hints to the SQL or doing a cunning little rewrite, but if we take that approach, we may have to do that same thing time and time again as the problem reappears in other places.

On the other hand, if we understand the underlying issue, we can fix the issue once and know that we have addressed every occurrence of the problem.

What's in This Book

This volume covers the basic details of optimization. It is not intended as a complete reference to how the optimizer works—you only have to notice that, with 14 chapters available, I don't get on to joins until Chapter 10, and you will realize that there is a huge amount to cover.

The important buzz words for optimization are

- *Selectivity and cardinality*: What fraction of the data will a predicate identify and how many rows does that turn into.
- *Access path*: Should a query use a B-tree index, or perhaps combine a couple of bitmaps indexes, or ignore indexes completely when visiting a table.
- *Join order*: Which table should a query visit first, and where should it go from there to do the least work to get the required result.

Although I will make a few comments about some of the more subtle features that need to be considered, this book really does restrict itself to just a critical handful of core concepts. How does the optimizer work out how much data a predicate is going to produce? How does it invent a number to represent the work of doing a tablescan, and how does it compare that with the work needed to use an index? What sort of figures go into estimating the resources needed to sort, or do a hash join?

I can examine a query, the objects in it, and the `10053 trace` file, and usually explain why one path has been taken in preference to another path. Unfortunately, I can't tell you how to do the same with every trace file you are ever going to see, because I can't cover all the options (I haven't even seen all the options yet), and this would be a very tedious book if I tried.

But even though I can't possibly tell you all the answers, I believe this book may give enough of the basic methods to let you work out what's going on in most of the cases you will have to examine.

What's Not in This Book

Inevitably there have to be omissions. Some things I have ignored because they are peripheral to the core activity of the optimizer, some I have ignored because they still have a very small audience, some had to be excluded through lack of space.

I haven't mentioned the rule based optimizer (RBO) at all because everyone should be trying to get rid of it. I haven't mentioned anything in the realm of the extensible optimizer (including context and spatial indexing) because they are not mainstream topics. I haven't mentioned analytic functions, model clauses (10*g*), or OLAP, because they are all initially driven by the need to acquire data before they do their own brand of number crunching—and the acquisition is probably the most time-critical aspect of the job.

I haven't mentioned objects—because they don't exist as far as the optimizer is concerned. When you create an object type and create data segments of object types, Oracle turns them into simple tables and indexes—the optimizer doesn't care about objects.

Finally, I have hardly mentioned parallel query, partitioned tables, distributed queries, and some of the slightly more subtle physical options of Oracle, such as clusters and IOTs. These omissions are necessary for two reasons: first, lack of room, and second, a desire to avoid blurring the important points in a welter of related details. There is a lot to the optimizer, it's hard to keep it in focus, and it's best to take it a bit at a time.

What's in Future Books

This book is the first in a series of three. Future volumes will cover important Oracle features, in particular partitioned tables, parallel execution, index organized tables, dynamic sampling, and query rewrite.

There will also be more advanced information on material previously covered in this volume, such as further access paths for B-tree indexes, comparisons between cluster access and indexed access, and more details on histograms.

The final important tranche of information that goes with cost-based optimization is the infrastructure to support it and allow you to understand it. The major topics here are understanding and interpreting execution plans, understanding the meaning and use of hints, and getting the best out of the `dbms_stats` package.

This book is based on 9.2, with observations on its differences from 8*i*, and comments about how 10*g* has introduced more change. Future volumes will make very little explicit reference to 8*i*, and say much more about 10*g*.

Organization

The chapters of this book cover the following topics in the order listed:

- *Tablescans*: Which help to start us off simply, and say something about CPU costing.
- *Simple selectivity*: Only one table, but lots of important arithmetic ideas.
- *Simple B-tree indexes*: The difference between single and multiblock reads.
- *The clustering factor*: Perhaps the most critical feature of an index.
- *More subtle selectivity*: An introduction to the many minor variations on a basic theme.
- *Histograms*: Why you may need a few, and the difference between OLTP and DSS/DW.
- *Bitmap indexes*: Because not all indexes are equal.
- *Transformations*: What you see is not necessarily what you get.
- *Joins*: Four whole chapters just to join two tables.
- *The 10053 trace*: A worked example.
- *Upgrade issues*: A collection of warning and notes, collated from the rest of the book.

Each chapter contains a number of code extracts from a set of SQL scripts that are available by download from Apress (www.apress.com). These scripts are there so that you can run them on your own system to reproduce and investigate the observation made in the chapter. Make sure you examine these scripts, as they contain extra comments and some extra tests that are not mentioned in the main body of the text. I will also be publishing these scripts on my web site (www.jlcomp.demon.co.uk) enhancing them, and adding further commentary from time to time.

The scripts are important—things do change, and changes can have a serious effect on your production systems. If you have scripts to test basic mechanisms, then you can repeat the tests on every upgrade to see if there are any changes that you need to know about.

One important point to note about the extracts that appear in the body of the text is that they often include lines to standardize the test environment, for example:

```
alter session set "_optimizer_system_stats_usage" = false;
```

Don't use commands like these in a production system just because they appear in this book. They are not there as an example of good programming practice; they are there in an effort to avoid the side effects that appear when one database has (for example) a completely different set of system statistics from another.

You will also find three init.ora files on the CD, and a script for creating test tablespaces in 9*i* and 10*g*. All four files will have to be edited to deal with problems of directory naming conventions; and the init.ora files for 9*i* and 10*g* will also have to be adjusted to suit your rollback/undo management options. I have chosen to run with init.ora files for 9*i* and 10*g* to avoid accidental changes to an spfile, but you may wish to merge the init.ora settings into an spfile.

The Obligatory Dire Warning

Whenever someone asks me to autograph a copy of *Practical Oracle 8i*, alongside my signature I always offer to include my motto: *Never believe all you read.* (If you ask me to sign your copy of this book, the motto will be *Just because it's printed, doesn't mean it's true.*) There are always special cases, different parameter settings, and bugs. (Not to mention thousands of things that I don't know, and some things that I think I know but haven't got right.)

Consider the following simple experiment (script in-list.sql in the online code suite)—run on a locally managed tablespace, using manual segment space management, with an 8KB block size:

```
create table t1 as
select
        trunc((rownum-1)/100)   n1,
        rpad('x',100)           padding
from
        all_objects
where
        rownum <= 1000
;

--      Collect statistics using dbms_stats here
```

```
set autotrace traceonly explain

select  *
from    t1
where
        n1 in (1,2)
;
```

Because of the `trunc()` function, the `n1` column takes the values from 0 to 9, with 100 occurrences of each value. So the query will return 200 rows. Run this test under 8.1.7.4, and then 9.2.0.6, and check the **cardinality** reported by **autotrace**. I got the following results:

```
Execution Plan (8.1.7.4 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=190 Bytes=19570)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=3 Card=190 Bytes=19570)

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=200 Bytes=20400)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=4 Card=200 Bytes=20400)
```

In 8.1.7.4, the optimizer gives an estimate of 190 rows as the cardinality of the result set. In 9.2.0.6, the optimizer has an algorithm that produces an estimate of 200 as the cardinality of the result set. This happens to be correct—and it is the result of an error in the optimizer code being fixed.

Before 9*i* was released, my three-day seminar included the observation that the calculations in 8*i* seemed to allow for a possible benefit that could be obtained when mixing indexed access paths and in-lists. Then 9*i* came out, and I ran the test I've just described and realized that it wasn't a cunning calculation in 8*i*, it was a bug. (See Chapter 3 for more details.)

So if your experience contradicts my claims, you may have a configuration that I haven't seen—I've only got through a few hundred databases in my lifetime, and there must be millions of combinations of configuration and data distribution details that could make a difference. In most cases, the download includes scripts to allow you to reproduce the test cases in the book—so you can at least check whether my test cases behave as predicted when run on your system. Whatever the results, you may get some clues about why your observations differ from my observations.

It is left as an exercise to the reader to compare the results from 9.2.0.6 and (specifically) 10.1.0.4 when you change the predicate in the previous query from

```
where n1 in (1,2);
```

to

```
where n1 in (11, 12);
```

and then change the literal values to ever increasing values. The results are quite surprising—and will be mentioned further in Chapter 3.

Theory vs. Practice

It is important to keep reminding yourself as you read this book that there are two separate stages to running a query. First, the optimizer works out what it "thinks" is going to happen, then the run-time engine does what it wants to do.

In principle, the optimizer should know about, describe, and base its calculations on the activity that the run-time engine is supposed to perform. In practice, the optimizer and the run-time engine sometimes have different ideas about what they should be doing.

explain plan

In a later book in this series, I will describe the methods you can use to acquire execution plans, and list various reasons why the resulting plans may be wrong, or at least misleading. For the present, therefore, let me just say that most of the examples in this book use `autotrace` or the `dbms_xplan` package to display execution plans for reasons of simplicity.

However, when I am on-site investigating a real problem and have any reason to doubt these theoretical tools, I may choose to run any problem queries so that I can check the `10053 trace` files, the statistical information from the `10046 trace` files, and the contents of `v$sql_plan`. And for cases involving partitioned objects or parallel executions, I may use a couple of other trace events to pin down details that are not otherwise available in any form of the execution plan.

All the tools for generating or investigating execution plans have some deficiencies. Do not depend on `autotrace` or `dbms_xplan` simply because they seem to be the only things I use in this book.

Conclusion

When you've finished reading this book (the first time), there are three key things that I hope you will remember from it.

First, there is usually a good solution to your business problem somewhere in the execution paths available to the optimizer. You just have to work out what the path is, and how to make it happen.

Second, anyone can build good, incremental tests that start small and simple but grow in scale and subtlety when, *and only when*, the situation warrants it.

Finally, if you observe a performance problem and find yourself addressing it in terms of the data content, requirements for access, and physical storage strategies, then you've learned the most important lesson this book has to offer.

With Apologies to the Late Douglas Adams

There is a theory stating that if ever anyone discovers exactly what the cost based optimizer does and how it works, it will instantly disappear and be replaced by something even more bizarre and inexplicable.

There is another theory that states this has already happened ... twice.

Test Cases

The files in the download for the preface are as follows:

Script	Comments
`in_list.sql`	Demonstration of changes in the in-list iterator selectivity calculation
`setenv.sql`	Sets a standardized environment for SQL*Plus

CHAPTER 1

What Do You Mean by Cost?

You have to be a little careful when you talk about the cost of an SQL statement because it's very easy to use the word in two different ways simultaneously. On one hand, you may be thinking about the *magic number* generated by a tool such as `explain plan`; on the other, you may be thinking about the actual resource consumption as a statement executes. In theory, of course, there ought to be a nice, simple relationship between the two, and it shouldn't be necessary to be so fussy about the meaning.

In this book, the word *cost* will always mean the result of the calculation performed by the optimizer. The purpose of the book is to explain the main features of how the optimizer does the calculations that produce the cost figure that determines the **execution plan** for an SQL statement.

As a side effect of this explanation, I will also describe some of the problems that allow the optimizer to produce a cost that appears to have nothing to do with actual resource consumption. It is my fond hope that as you read this book, you will experience magic moments when you say, "So that's why query X keeps doing that ..."

Optimizer Options

The commonest type of statement is the `select` statement—but even though the rest of this book will focus almost exclusively on `select`s, it is worth remembering that any statement, be it a query, `DML` (such as an update), or `DDL` (such as an index rebuild) is subject to analysis by the **cost based optimizer** (**CBO**).

Oracle has three variants on its cost based optimizer. The three variants have different constraints built into their code, but they all follow the same strategy—which is to find the execution mechanism that uses the fewest resources to achieve the required result for a given statement. The variants are identified by the legal settings of the parameter `optimizer_mode`:

- `all_rows`: The optimizer will attempt to find an execution plan that completes the statement (typically meaning "returns all the rows") in the shortest possible time. There are no special constraints built into this code.
- `first_rows_N`: The number `N` can be 1, 10, 100, or 1000 (and as a further refinement there is the `first_rows(n)` **hint**, where the number `n` can be any positive whole number). The optimizer first estimates the number of rows that will be returned by completely analyzing just the first **join order**. This tells it what fraction of the total data set the query is supposed to fetch, and it restarts the entire optimization process with the target of finding the execution plan that minimizes the resources required to return that fraction of the total data. This option was introduced in 9*i*.

- `first_rows`: Deprecated in 9*i*, but maintained for backward compatibility. The optimizer will attempt to find an execution plan to return the first row of a result set as rapidly as possible. There are several high-level constraints built into the code. For example, one constraint appears to be *avoid merge joins and hash joins unless the alternative is a nested loop with a full tablescan on the inner (second) table*. The rules tend to push the optimizer into using indexed access paths, which are occasionally very inappropriate access paths. An example of and workaround to this particular issue can be found in the script `first_rows.sql` in the online code suite for this chapter, available from the Apress web site (`www.apress.com`) or from the author's web site (`www.jlcomp.demon.co.uk`).

There are two other options for the `optimizer_mode` (even in 10*g*): `rule` and `choose`. This book is going to ignore **rule based optimization** (**RBO**) completely because it has been deprecated for years and was finally desupported in 10*g* (even though some of the internal SQL still uses the `/*+ rule */` hint).

As for the `choose` option, this gave the optimizer a run-time choice between rule based optimization and `all_rows`. Since I am ignoring rule based optimization, nothing more needs to be said about `choose` mode. I will just mention that in 10*g*, if you use the **Database Configuration Assistant** (**DBCA**) to build a database, or call `catproc.sql` during a manual creation, you will automatically install a job (created by the `catmwin.sql` script and visible in view `dba_scheduler_jobs`) that will, every 24 hours, generate statistics on any tables with missing or *stale* statistics. So, if you set the `optimizer_mode` to `choose`, it will probably result in `all_rows` optimization, but you may find that any tables that missed the most recent pass of statistics generation will then be subject to **dynamic sampling**. This is because the default value in 10*g* for the parameter `optimizer_dynamic_sampling` is 2 (which means use dynamic sampling on any table without statistics), rather than 1, as it was in 9*i*.

Another parameter associated with the `optimizer_mode` is `optimizer_goal`. There seems to be no difference between these two parameters as far as optimization strategies go, although the `optimizer_goal` can only be set dynamically within a session, and is not available in the `spfile` (`init.ora`).

So What Is the Cost?

One of the commonest questions about the CBO on the Internet is *What does the cost represent?* This is usually followed by comments like *According to `explain plan`, the cost of doing a hash join for this query is 7 million and the cost of a nested loop is 42—but the hash join completes in 3 seconds and the nested loop takes 14 hours.*

The answer is simple: the cost represents (and has always represented) the optimizer's best estimate of the time it will take to execute the statement. But how can this be true when people can see oddities like this hash join/nested loop join example? The answer can usually be found in that good old acronym **GIGO**: **Garbage In, Garbage Out**.

The CBO makes errors for six main reasons:

- Some inappropriate assumptions are built into the cost model.
- The relevant **statistics** about the data distribution are available, but misleading.
- The relevant statistics about the data distribution are not available.

- The performance characteristics of the hardware are not known.
- The current workload is not known.
- There are bugs in the code.

We shall examine these issues, and the evolution of the optimizer to address them, in the course of the book. However, given the impact they make, I shall just mention briefly some changes to the CBO that have appeared in recent versions of Oracle (and may appear in future versions) that make it much harder to explain what the optimizer is up to.

In 8*i*, the optimizer simply counted the number of requests it expected to make to the I/O subsystem. The execution plan that required the smallest number of requests was the one that was chosen. This did not account for the fact that a tablescan might require vastly more CPU than an indexed access path. It did not account for the fact that a 128-block read might actually take more time than a single-block read. It did not account for the fact that a notional 128-block read might actually turn into, say, 25 separate read requests because a randomly scattered subset of the required blocks were already in the Oracle buffer. It did not account for the fact that an I/O request might be satisfied from an intervening cache rather than requiring an actual physical disk read.

In 9*i*, the optimizer introduced a feature referred to as **CPU costing**. You can store typical response times for single-block and multiblock I/O requests in the database along with an indicator of the typical size of a multiblock request, and the optimizer will factor these values into the cost equation. The optimizer will also convert a count of CPU operations (e.g., compare date column with constant) into CPU time, and factor that into the equation as well. These refinements ensure that the optimizer has a better estimate of the cost of tablescans and tends to produce more sensible execution plans and fewer anomalies like the hash join/nested loop join noted earlier in this section.

In 10*g*, an **offline optimizer** appeared. This allows you to generate and store critical statistical information (in the form of a **profile**) that helps the **online optimizer** deal with the problems of correlated data distributions. In effect, you can enhance a query by adding a hint that says, "At this point, you will have 15 times as much data as you expect." Both 9*i* and 10*g* collect **cache statistics** at the object level and, looking to the future, 10*g* has a couple of **hidden parameters** that look as if they will enable high-precision, cache-aware calculations to be used. If (or when) this feature goes into production, the optimizer may be able to produce plans that better reflect the number of actual I/O requests needed based on recent cache success.

Moreover, both 9*i* and 10*g* collect run-time statistics in the views `v$sql_plan_statistics` and `v$sql_plan_statistics_all`. These statistics could, in theory, be fed back to the optimizer to give it a second chance at optimizing a query if the actual statistics differ too much from the assumptions made by the optimizer.

One day, perhaps within the next couple of minor releases, you will be able to look at the cost of a query and convert it confidently into an approximate run time, because the optimizer will have produced exactly the right execution plan for your data, on that machine, at that precise moment in time. (Of course, an optimizer that changes its mind every five minutes because of the ongoing activity could be more of a threat than a benefit—I think I might favor predictability and stability to intermittently failing perfection.)

In the meantime, why am I so sure that cost is supposed to equal time? Check the *9.2 Performance Guide and Reference* (Part A96533), pages 9-22:

```
According to the CPU costing model:
      Cost = (
            #SRds * sreadtim +
            #MRds * mreadtim +
            #CPUCycles / cpuspeed
      ) / sreadtim

where
      #SRDs - number of single block reads
      #MRDs - number of multi block reads
      #CPUCycles - number of CPU Cycles

      sreadtim - single block read time
      mreadtim - multi block read time
      cpuspeed - CPU cycles per second
```

Translated, this says the following:

The cost is the time spent on single-block reads, plus the time spent on multiblock reads, plus the CPU time required, all divided by the time it takes to do a single-block read. Which means the cost is the total predicted execution time for the statement, expressed in units of the single-block read time.

REPORTING CPUSPEED

Although the manual indicates that the `cpuspeed` is reported in cycles per second, there are two possible errors in the statement.

The simple error is that the values that appear suggest the unit of measure is supposed to be millions of cycles per second (i.e., CPU speed in MHz). Even then, the number always seems to fall short of expectations—in my case by a factor of anything between 5 and 30 on various machines I've tested.

The more subtle error then is that the value may actually be a measure of millions of *standardized oracle operations* per second, where a "standardized oracle operation" is some special subroutine designed to burn CPU. (A `10053 trace file` from 10.2 offers corroborative evidence for this.)

Whether the number represents cycles per second or operations per second, the difference is only a simple scaling factor. The mechanism involved in using the `cpuspeed` is unchanged.

Why does Oracle choose such an odd time unit for the cost, rather than simply the number of centiseconds? I think it's purely for backward compatibility. The cost under 8*i* (and 9*i* before you enabled full CPU costing) was just the count of the number of I/O requests, with no distinction made between single-block and multiblock I/Os. So, for backward compatibility, if the new code reports the time in units of the single-block read time, the number produced for the cost for a typical (lightweight, index-based) **OLTP** query will not change much as you upgrade from 8*i* to 9*i*.

A little extra thought about this formula will also tell you that when you enable CPU costing, the cost of a tablescan will tend to go up by a factor that is roughly (`mreadtim` / `sreadtim`). So 9*i* with CPU costing will tend to favor indexed access paths a little more than 8i did because 9i

recognizes (correctly) that multiblock reads could take longer than single-block reads. If you are planning to upgrade from 8*i* to 9*i* (or 8*i* to 10*g*), make sure you enable CPU costing from day one of your regression testing—there will be some surprises waiting for you.

A final consideration when examining this formula is that there is no explicit mention of any components relating to the time spent on the I/O that can result from merge joins, hash joins, or sorting. In all three cases, Oracle uses **direct path** writes and reads with sizes that usually have nothing to do with the normal multiblock read size—so neither `mreadtim` nor `sreadtim` would seem to be entirely appropriate.

Transformation and Costing

There is an important aspect of optimization that is often overlooked and can easily lead to confusion, especially as you work through different versions of Oracle. Before doing any cost calculation, Oracle may *transform* your SQL into an equivalent statement—possibly one that isn't even legally acceptable SQL—and then work out the cost for that equivalent statement.

Depending on the version of Oracle, there are transformations that (a) cannot be done, (b) are always done if possible, and (c) are done, costed, and discarded. Consider, for example, the following fragments of SQL (the full script, `view_merge_01.sql`, is available with the online code suite for this chapter):

```
create or replace view avg_val_view
as
select
       id_par,
       avg(val)      avg_val_t1
from   t2
group by
       id_par
;

select
       t1.vc1,
       avg_val_t1
from
       t1,
       avg_val_view
where
       t1.vc2 = lpad(18,32)
and    avg_val_view.id_par = t1.id_par
;
```

You will note that `avg_val_view` is an **aggregate view** of the table `t2`. The query then joins `t1` to `t2` on the column that is driving the aggregation. In this case, Oracle could use one of two possible mechanisms to produce the correct result set: instantiate the aggregate view and then join the view to table `t1`, or merge the view definition into the query and transform it. From a 9*i* system, here are the two possible execution plans:

```
Execution Plan (9.2.0.6 instantiated view)
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=15 Card=1 Bytes=95)
  HASH JOIN (Cost=15 Card=1 Bytes=95)
    TABLE ACCESS (FULL) OF 'T1' (Cost=2 Card=1 Bytes=69)
    VIEW OF 'AVG_VAL_VIEW' (Cost=12 Card=32 Bytes=832)
      SORT (GROUP BY) (Cost=12 Card=32 Bytes=224)
        TABLE ACCESS (FULL) OF 'T2' (Cost=5 Card=1024 Bytes=7168)

Execution Plan (9.2.0.6 merged view)
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=14 Card=23 Bytes=1909)
  SORT (GROUP BY) (Cost=14 Card=23 Bytes=1909)
    HASH JOIN (Cost=8 Card=32 Bytes=2656)
      TABLE ACCESS (FULL) OF 'T1' (Cost=2 Card=1 Bytes=76)
      TABLE ACCESS (FULL) OF 'T2' (Cost=5 Card=1024 Bytes=7168)
```

As you can see from the execution plans, my example allows Oracle to aggregate table `t2` and then join it to `t1`, but it also allows Oracle to join the two tables and then do the aggregation. The "equivalent" code for the merged view would look something like this:

```
select
       t1.vc1,
       avg(t2.val)
from
       t1, t2
where
       t1.vc2 = lpad(18,32)
and    t2.id_par = t1.id_par
group by
       t1.vc1, t1.id_par
;
```

So which of the two paths is better and which execution plan will the optimizer choose? The answer to the first part of the question depends on the data distribution.

- If there is an efficient way to get from `t1` to `t2`, and if there are only a couple of rows in `t2` for each row in `t1`, and if the extra volume of data added to each row by `t2` is small, then joining before aggregating will probably be the better idea.
- If there is no efficient way to get from `t1` to `t2`, and there are lots of rows in `t2` for each row in `t1`, and the extra volume of data added to each row by `t2` is large, then aggregating before joining will probably be the better idea.
- I can tell you about the extremes that make the options possible, but I can't give you an immediate answer about all the possible variations in between. But the optimizer can make a pretty good estimate.

The answer to the second part of the question depends on your version of Oracle. If you are still running 8*i*, then Oracle will aggregate the view and then perform the join—*without considering the alternative*. If you are running 9*i*, Oracle will open up the view, join, and then aggregate—*without considering the alternative*. If you are running 10*g*, Oracle will work out the cost of both alternatives separately, and take the cheaper one. You can see this if you rerun script `view_merge_01.sql`, but set `event 10053` (which will be discussed in later chapters) to generate a **trace file** of the optimizer's cost calculations.

EVOLUTION OF OPTIMIZATION CODE

As you go through the versions of Oracle, you will notice many examples of mechanisms that can be enabled or disabled at the system or session level by hidden parameters; there are also many mechanisms that can be enabled or disabled by hints at the SQL level.

A common evolutionary path in the optimizer code seems to be the following: hidden by undocumented parameter and disabled in the first release; silently enabled but not costed in the second release; enabled and costed in the third release.

In this specific example, the option to open up the aggregate view and merge it into the rest of the query depends on the hidden parameter `_complex_view_merging`, which defaults to false in 8*i*, but defaults to true from 9*i* onwards. You can force 8*i* to do **complex view merging** by changing this parameter—although you may find some cases where you also need to use the `merge()` hint to make merging happen. You could also stop 9*i* and 10*g* from doing complex view merging by changing the value of this parameter, but it might be more sensible to use the `no_merge()` hint selectively—which is what I did to get the first of the two execution plans shown previously.

There are many features available to the optimizer to manipulate your query before optimizing it—**predicate pushing**, **subquery unnesting**, and **star transformation** (perhaps the most dramatic example of query transformation) have been around for a long time. Predicate generation through **transitive closure** has also been around years, and predicate generation from **constraints** has come and gone across the versions. All these possibilities (not to mention the explicit **query rewrite** feature), and perhaps some others that I haven't even noticed yet, make it much harder to determine precisely what is going on with complex SQL unless you examine the details very closely—ultimately wading through a `10053 trace`.

Fortunately, though, you won't often need to get down to this extreme, as the detail offered by `explain plan` is often enough to tell you that a transformation has occurred. A couple of hints, or a check of the optimizer-related parameters, will usually tell you whether a transformation is mandatory or optional, costed or uncosted, and how much control you have over it.

WYSIWYG?

One more problem faces you as you struggle to unravel the details of how the cost based optimizer works: what you see is not always what you get.

There are three different layers of operational complexity:

- First, the execution plan tells you what the optimizer thinks is going to happen at run time, and produces a cost based on this model.
- Second, the execution engine starts up and executes the model dictated by the optimizer—but the actual mechanism is not always identical to the model (alternatively, the model is not a good description of what actually happens).
- Finally, there are cases where the resources required to execute the model vary dramatically with the way in which the incoming data happens to be distributed.

In other words, the optimizer's execution plan may not be exactly the run-time execution path, and the speed of the run-time execution path may be affected by an unlucky choice of data. You will see examples of this in Chapter 9.

Summary

The main purpose of this very short chapter is to warn you that there are no quick and snappy answers to questions like *Why did Oracle do X?* At the very least, you have three optimizer modes and two (possibly three) major versions of Oracle to check before you start to formulate an answer.

Oracle can do all sorts of transformations to your SQL before optimizing it. This means that an execution plan you were expecting to see may have been eliminated by a transformation before the optimizer started to do any costing.

On the positive side, though, the basic arithmetic is always the same. Cunning refinements, little tweaks, new features, and dirty tricks are thrown in to confuse the issue—but 95% of everything that the optimizer does can be described and explained reasonably accurately in just a few chapters.

Test Cases

The files in the subdirectory for this chapter are shown in Table 1-1.

Table 1-1. *Chapter 1 Test Cases*

Script	Comments
`first_rows.sql`	Demonstration for the problems of `first_rows` optimization
`view_merge_01.sql`	Demonstration of the changes in complex view merging
`setenv.sql`	Sets the standardized test environment for SQL*Plus

CHAPTER 2

Tablescans

At first sight, you might think that there couldn't be a lot to say about **tablescans**, but you may be in for a surprise. Since the cost calculations for tablescans are nice and simple, I thought I'd use them in this early chapter to demonstrate the four different strategies for cost based optimization that have evolved over the years, briefly outlined here:

- **Traditional**: Simple counting of read requests
- **System statistics (1)**: Accounting for size and time of read requests
- **System statistics (2)**: Accounting for CPU costs, and size and time of read requests
- **System statistics (3)**: Accounting for caching, CPU costs, and size and time of read requests

The traditional method appeared in Oracle 7, and has been enhanced continuously since. Some of the more serious drawbacks, particularly the failure to account for caching and variable I/O costs, were partially addressed in 8*i*. I will be saying more about those fixes in Chapter 4.

The use of **system statistics** (**CPU costing**) appeared in 9*i*, and is still undergoing enhancement in 10*g*. The "normal" variant of system statistics is number 2 in our list—accounting for CPU costs and variable I/O costs.

The option to disable just the CPU component of system statistics is available through an undocumented parameter whose effects seem to vary with version of Oracle. And the potential for including caching effects appears in 10*g* but has to be enabled through an undocumented parameter, and is therefore an assumption I am making about future directions, not a fact.

As you can see, the Oracle kernel code has been extended to become more aware of the environment that it is running in, which should make it better able to produce the most appropriate execution plan for the moment that the query is optimized.

Once you've worked through all four variations that Oracle has to offer, you might think you know all there is to know about the different methods of calculating the costs of a tablescan. But you've only just begun—remember that an **index fast full scan** is really just a variant on the tablescan idea; think about the fact that there may be some anomalies introduced by **automatic segment space management** (**ASSM**); and consider that tablescans can be performed using **parallel execution**, and might involve **partitioned tables**.

Getting Started

Before running any tests, you will need to do a little preparation work to be able to examine the details of the execution plans that the optimizer is going to produce.

On my 9*i* system I've used the script `$ORACLE_HOME/rdbms/admin/utlxplan.sql` to build a `plan_table`, but I've modified the standard script to create the table as a `global temporary table` with the option `on commit preserve rows`, owned by the `system` account. Then I've granted all privileges on this table to `public`, and given it a `public synonym`. (Oracle 10*g* has also adopted this strategy.)

Oracle supplies a couple of scripts (`$ORACLE_HOME/rdbms/admin/utlxplp.sql` and `utlxpls.sql`) to ensure that you can produce a plan that probably includes all the latest features, but I've also prepared my own scripts for reporting execution plans: `plan_run81.sql` and `plan_run92.sql`, which are available in the online code suite.

In 9*i* you may also need to execute the script `dbmsutil.sql` to create the package `dbms_xplan` that Oracle calls to produce a formatted execution plan. You may want to get familiar with Oracle's scripts simply for the sake of consistency of appearance.

■ **Note** Oracle Corp. has been supplying scripts for standardizing the presentation of execution plans for a long time. In 9*i*, these scripts changed from complex SQL statements to a simple call to a packaged procedure. In 10*g*, the packaged procedure was enhanced in several ways to allow you to report an execution plan for a single statement still in the view `v$sql`, or even to report a set of plans for all the statements returned by a query made against either `v$sql` or the **Automatic Workload Repository** (**AWR**) tables. Keep an eye on the script `dbmsutl.sql` and the `dbms_xplan` package if you want to keep abreast of the latest developments.

It's also critical to work on a stable platform when investigating the optimizer. In my case, the most important features of my starting test environment are

- Block size 8KB
- `db_file_multiblock_read_count = 8`
- Locally managed tablespaces
- Uniform extents of size 1MB
- Freelist space management—not automatic segment space management
- `optimizer_mode = all_rows`
- System statistics (`cpu_costing`) initially disabled

There are various parameters that I might then adjust for different test cases, but I always start with this baseline.

So let's build a test case—and for this experiment we will go back briefly to 8*i*. The full script is `tablescan_01.sql` in the online code suite:

```
execute dbms_random.seed(0)

create table t1
pctfree 99
pctused 1
```

```
as
select
        rownum                                  id,
        trunc(100 * dbms_random.normal)         val,
        rpad('x',100)                           padding
from
        all_objects
where
        rownum <= 10000
;
```

With this script, I've created a table of 10,000 rows that also spans 10,000 blocks because of the unusual value I've chosen for `pctfree`. (It's a trick I often use in test cases to waste lots of space without generating lots of data—occasionally it causes surprises because one row per block is a very special boundary condition.) Using view `all_objects` if you are running 8*i* with Java installed is a convenient way of getting about 30,000 rows in a scratch table without resorting to cunning tricks.

REPEATABLE RANDOM DATA

Oracle Corp. supplies a lot of useful packages that rarely see the light of day. The `dbms_random` package is one of these. It includes several procedures for generating pseudo-random data, including a procedure for generating uniform random numbers, a procedure for generating normally distributed random numbers, and a package for creating random strings of a specified size. I frequently use the `dbms_random` package as a quick, reproducible method for generating test data that approximates to a particular type of production situation.

After generating statistics on the table, I just enable `autotrace` and run a simple query. Since there are no indexes on the table, Oracle has to perform a tablescan, and under Oracle 8*i* you will get an execution plan like this one:

```
select   max(val)
from     t1
;

Execution Plan (8.1.7.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1518 Card=1 Bytes=4)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=1518 Card=10000 Bytes=40000)
```

From `autotrace` we get three critical numbers on most lines of the execution plan—the cumulative cost for a line (`Cost=`), the number of rows that each line of the plan will generate (`Card=`), and the total volume of data that each line will generate (`Bytes=`).

In this case, we can see that the `max()` function will return just one row of 4 bytes, at a cost of 1,518 (the result will be slightly different in 9*i* and 10*g*); but note that most (in fact all) of the

cost comes from the full tablescan in line 2, which will pass 10,000 rows and 40,000 bytes up to the `sort (aggregate)` line, and the `sort (aggregate)` in line 1 seems to have been free of charge.

Before investigating the arithmetic, let's see how we can persuade Oracle to reevaluate the cost: let's change the size of the `db_file_multiblock_read_count`.

```
alter session set db_file_multiblock_read_count = 16;
```

When we double the value of this parameter, we find the cost of the tablescan drops—but doesn't halve. We can automate the process of recording cost against size of `db_file_multiblock_read_count`, and Table 2-1 shows a few results this will produce (still from Oracle 8*i*). Note especially the column headed "adjusted dbf_mbrc"; I have calculated this as (10,000 / cost), that is, the number of blocks in my table divided by the calculated cost of scanning the table.

Table 2-1. *Effects of Changing the Multiblock Read Count*

db_file_multiblock_read_count	Cost	Adjusted dbf_mbrc
4	2,396	4.17
8	1,518	6.59
16	962	10.40
32	610	16.39
64	387	25.84
128	245	40.82

The *adjusted dbf_mbrc* is significant, because a little further experimentation shows that the cost of a tablescan when using the traditional costing methods is (blocks below **high water mark** / *adjusted dbf_mbrc*).

If you do a tablescan of a 23,729 block table with a `db_file_multiblock_read_count` of 32, then the cost of the tablescan will be reported as `ceil(23,729 / 16.39)`; if you do a tablescan of a 99-block table with a `db_file_multiblock_read_count` of 64, then the cost of the tablescan will be `ceil(99 / 25.84)`.

There will be small rounding errors, of course—if you want a more accurate set of figures, create a very large table (or fake it by using the procedure `dbms_stats.set_table_stats` to claim that your test table is 128,000,000 blocks).

It doesn't matter what your standard block size is (although, as you will see, you have to fiddle with the formula when you start working with nonstandard block sizes), there is just one reference set of values that Oracle uses to calculate the cost of a tablescan. The online code suite includes a script called `calc_mbrc.sql` that can generate the full set of values for the *adjusted dbf_mbrc*. If you run this script, you will find that there is a point beyond which the *adjusted dbf_mbrc* will not change. When Oracle starts up, it negotiates with the operating system to find the **largest physical read size** your operating system will allow, and silently uses that to limit whatever value you set for the `db_file_multiblock_read_count`.

HIGH WATER MARK

In general, when a table or index segment is first created, space for that segment will be preallocated from a data file but very little of the space will be formatted for use. As data arrives, blocks will be formatted a few at a time.

In the simplest setup, Oracle would format "the next five" blocks from the preallocated space as the need arose, and the object's high water mark (HWM) would be adjusted to show how many blocks had been formatted and were available for use.

With the arrival of ASSM in 9*i*, Oracle formats groups of adjacent blocks (typically 16, it seems) at a time. The high water mark still identifies the highest formatted block in the segment, but ASSM randomizes the allocation slightly, so that unformatted holes (16 blocks, or multiples thereof) can appear in the middle of the object.

ASSM also allocates one or two **bitmap space management blocks** per extent. So the object is larger, and the cost of a tablescan is pushed upwards—for large objects the difference is not huge, but there are other side effects to consider, as you will see in later chapters.

Even in this very simple case, it is important to recognize that there can be a difference between the optimizer's calculations and the run-time activity. The *adjusted dbf_mbrc* is used purely for calculating the cost. At run time Oracle will try to use the value of the `db_file_multiblock_read_count` parameter to perform a tablescan—although odd glitches such as extent boundaries blocks already being cached and therefore not subject to being read usually mean that a serial tablescan will do a fairly random selection of read sizes from one through to the full `db_file_multiblock_read_count`. (This typical run-time variation is probably the rationale behind the way the *adjusted dbf_mbrc* gets more pessimistic as the `db_file_multiblock_read_count` gets larger and larger.)

Before moving on, we should take a look at a slightly more complex query:

```
select
        val,
        count(*)
from
        t1
group by
        val
;

Execution Plan (8.1.7.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1540 Card=582 Bytes=2328)
   1    0   SORT (GROUP BY) (Cost=1540 Card=582 Bytes=2328)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=1518 Card=10000 Bytes=40000)
```

Note in this case that the cost of the tablescan is unchanged (1,518 in line 2), but the total cost of the query is 1,540 (line 1)—the extra 22 must be due to the cost of doing the `sort (group by)`. Whether this is a reasonable estimate or not remains to be seen—but it does seem a little expensive for sorting just 40KB of data.

■ **Note** We will examine sort costs in detail in Chapter 13, but one detail you might note now is that the `sort (aggregate)` reported in the simpler query doesn't really do any sorting, which is why the marginal cost on the `sort (aggregate)` line was zero. Oracle simply checks each incoming row against the current maximum. (This concept has been enhanced to produce the hash (group by) in 10*g* Release 2.) There are lots of shortcuts that the run-time engine takes that aren't necessarily reflected perfectly in execution plans.

The other important detail to pick up from this plan is the final **cardinality**: the optimizer has estimated that there will be 582 rows returned by this query. In fact, this is exactly right. Given the simplicity of the query, the optimizer could use one of the stored statistics for the `val` column (`user_tab_columns.num_distinct`) to produce the right answer. It only takes a slightly more complex query and the cardinality starts to drift from the correct answer though. As you will see in Chapter 10, it is very important that the optimizer gets an accurate estimate of the cardinality every step of the way.

Onwards and Upwards

We now move on to 9*i*, and the important enhancements made to the optimizer. Before we look at the most critical enhancements, though, this is a good moment to cast an eye over the multiple block size feature of 9*i* and repeat our test case using different block sizes.

Effects of Block Sizes

We start by retesting our baseline on the newer version of Oracle—still using a tablespace built with an 8KB block size—and note that the cost has gone up from 1,518 to 1,519. This is a minor variation in 9*i* controlled by parameter `_tablescan_cost_plus_one`, which is set to `false` in 8*i* and true in 9*i*. A tablescan in 9*i* is automatically a tiny bit more expensive than the equivalent tablescan in 8*i* (all other things being equal).

This is the type of minor tweak that appears surprisingly frequently in Oracle, and makes it very difficult to produce any sort of documentation about the optimizer that is both concise and complete. In this case, the change may represent the requirement for the necessary access to the table's segment header block at the start of a tablescan; alternatively it may simply be one of several tricks built into the optimizer to bias it towards using indexed access paths on very small tables.

After spotting the *plus 1* in the cost, we can try a few variations in block sizes (`tablescan_01a.sql` and `tablescan_01b.sql` in the online code suite) and see what happens when we fix the `db_file_multiblock_read_count` at 8 and leave CPU costing disabled. The three figures for each block size shown in Table 2-2 are

- The cost of a tablescan on a fixed number of blocks (10,000) as the block size changes
- The *adjusted dbf_mbrc* (10,000 / (cost – 1)) for each of the different block sizes
- The cost of a tablescan when the size (in MB) of the table is fixed and the block size changes

Table 2-2. *How the Block Size Affects the Cost*

Block Size	Cost of 10,000 Block Scans	Adjusted dbf_mbrc	Cost for 80MB Scan
2KB	611	16.39	2,439
4KB	963	10.40	1,925
8KB	1,519	6.59	1,519
8KB ASSM	1,540	n/a	1,540
16KB	2,397	4.17	1,199

We can draw two significant conclusions from this table. First, check the values for the *adjusted dbf_mbrc* column—where have you seen those figures (plus or minus small rounding errors) before?

- For the 2KB row, the value of 16.39 is the value we got in our first test with a `db_file_multiblock_read_count` of 32.

  Note: 2KB * 32 = 64KB

- For the 4KB row, the value of 10.40 is the value we got in our first test with a `db_file_multiblock_read_count` of 16.

  Note: 4KB * 16 = 64KB

- For the 16KB row, the value of 4.17 is the value we got in our first test with a `db_file_multiblock_read_count` of 4.

  Note: 16KB * 4 = 64KB

That 64KB that keeps on appearing is significant—it's 8KB * 8, our standard database block size multiplied by our setting of `db_file_multiblock_read_count`. So we infer that Oracle handles nonstandard block sizes by restating the `db_file_multiblock_read_count` as a directive about the read size based on the standard block size. For example, if the default block size is 8KB and the table to be scanned is in a tablespace with a 4KB block size, then the optimizer effectively doubles the `db_file_multiblock_read_count` to keep the read size constant before doing the normal cost calculation.

Moreover, if you generate **extended trace** files (**event 10046**) while doing this experiment on a tablespace with a 4KB block size, you find that the same adjustment occurs at run time. If you had set `db_file_multiblock_read_count` to 8 with a default block size of 8KB, you would find multiblock reads of 16 blocks for tablescans in a tablespace using 4KB blocks.

Our second conclusion comes from examining the cost of a tablescan where we keep the physical size of the data segment constant. Notice that the cost of the tablescan decreases as the block size increases. If you move an object from one tablespace to another with a different block size, the cost of scanning it with multiblock reads changes significantly—you may find that a surprising number of SQL statements change their execution plan as a result, and the change may not be for the better.

TUNING BY CHANGING BLOCK SIZES

Be very cautious with the option for using different block sizes for different objects—the feature was introduced to support transportable tablespaces, not as a tuning mechanism.

You may be able to find a few special cases where you can get a positive benefit by changing an object from one block size to another; but in general you may find that a few side effects due to the optimizer changing its arithmetic may outweigh the perceived benefits of your chosen block size.

One final observation—I have included the original results for the 8KB block size in the table; but I have also listed the costs when the table was in a tablespace that used ASSM. Notice that the cost of a tablescan has increased by about 1.5%. Every extent in my table had a couple of blocks taken out for space management bitmaps—2 blocks out of 128 since my extents were 1MB; the extra cost is largely due to the effect of those blocks on the high water mark of the table. Again, if you are advised to move an object into an ASSM tablespace for performance reasons (specifically to avoid contention on inserts), be just a little cautious—this is just one of the irritating little side effects of ASSM.

CPU Costing

One of the most serious defects of the optimizer prior to 9*i* was its assumption that single-block reads and multiblock reads were equally cost effective. This assumption is flawed on two counts. First, multiblock reads often take longer to complete than single-block reads (especially on systems with too few disks, configured with too small a stripe size). Second, a tablescan can use a surprising amount of CPU as each row is tested against some predicate.

In 9*i*, both these flaws are addressed through system statistics. You can collect system statistics over representative periods of time, or you could simply calibrate your hardware (at least the I/O subsystem) for absolute performance figures and then write system statistics into the database.

For example, you could issue the following two statements at 9:00 a.m. and noon respectively one Monday morning:

```
execute dbms_stats.gather_system_stats('start')
execute dbms_stats.gather_system_stats('stop')
```

The `'start'` option takes a starting snapshot of various figures from `v$filestat` (actually the underlying `x$` including some columns that are not exposed in `v$filestat`) and `v$sysstat`; the `'stop'` takes a second snapshot, works out various statistics about disk and CPU activity over that three hours, and records them in the database. The nature of the data collected and stored is version dependent—and is probably still subject to change in 10*g*, but you should find results in table `sys.aux_stats$` that look something like the following from 9*i* (10*g* has a few extra rows):

```
select
        pname, pval1
from
        sys.aux_stats$
```

```
where
        sname = 'SYSSTATS_MAIN'
;

PNAME            PVAL1
---------- ----------
CPUSPEED            559
SREADTIM          1.299
MREADTIM         10.204
MBRC                 6
MAXTHR        13938448
SLAVETHR        244736
```

Accessing the values by querying the base table is not the approved method, of course, and Oracle supplies a PL/SQL **API** to query, set, and delete system statistics. For example, we can set the basic four system statistics with code like the following, which comes from script `set_system_stats.sql` in the online code suite:

```
begin
   dbms_stats.set_system_stats('CPUSPEED',500);
   dbms_stats.set_system_stats('SREADTIM',5.0);
   dbms_stats.set_system_stats('MREADTIM',30.0);
   dbms_stats.set_system_stats('MBRC',12);
end;
/

alter system flush shared_pool;
```

I've included the flushing of the *shared pool* in this code fragment as a reminder that when you change the system statistics, existing cursors are not invalidated (as they would be for **dependent cursors** when you gather statistics on a table or index). You have to flush the *shared pool* if you want to make sure that existing cursors are reoptimized for the new system statistics.

■ **Note** If you want to use a low-privilege account to collect system statistics, you will need to grant the role `gather_system_statistics` to the account. This role is defined in `$ORACLE_HOME/rdbms/admin/dbmsstat.sql`. There was a bug in many versions of 9.2 that would result in a nonfatal Oracle error if an account tried to modify the system statistics more than once in a session.

The figures in my anonymous PL/SQL block tell Oracle that

- A single CPU on my system can perform 500,000,000 *standard operations* per second.
- The average single-block read time is 5 milliseconds.
- The average multiblock read time is 30 milliseconds.
- The typical multiblock read size is 12 blocks.

The maxthr and slavethr figures relate to throughput for **parallel execution** slaves. I believe that the figures somehow control the maximum **degree of parallelism** that any given query may operate at by recording the maximum rate at which slaves have historically been able to operate—but I have not been able to verify this. These two statistics are allowed to have the value –1, but if any of the others end up with the value –1, then the CPU costing algorithms will not be invoked. (The situation changes with 10*g*, where there are two sets of statistics—one of which is the noworkload set. If the main statistics are invalid, Oracle will fall back to using the noworkload statistics.)

SYSTEM STATISTICS 10.2

Although the manuals state that the CPUSPEED figure represents the CPU speed in cycles per second, you will probably find that it always falls short of the real CPU speed. Christian Antognini has suggested that the measure represents the number of times per second that Oracle can perform some calibrating operation on your platform. The 10053 trace file from 10*g* release 2 (10.2.0.1) includes a section on system statistics that corroborates this; for example, reporting my earlier sample set as follows (note the loss of precision in reporting the read times):

```
Using WORKLOAD Stats
CPUSPEED: 559 million instructions/sec
SREADTIM: 1 milliseconds
MREADTIM: 10 milliseconds
MBRC: 6.000000 blocks
MAXTHR: 13938448 bytes/sec
SLAVETHR: 244736 bytes/sec
```

Of course, the CPUSPEED is just a number, and the optimizer is just doing arithmetic with that number, whatever it represents—but if you have a proper understanding of where the number comes from, you may be a little more cautious about fiddling with it.

So how does the optimizer use these statistics? Modify the original test case to include the system statistics shown previously and run it under 9*i* with the db_file_multiblock_read_count set to 4, 8, 16, and 32 in turn (this test is available as script tablescan_02.sql in the online code suite). Autotrace reported the cost of the query as 5,031 in the first three cases. Unfortunately, the cost in the case of the 32-block read size was 5,032, a small but unexpected variation. Under 10*g*, the costs were one unit less across the board. The rules about rounding, truncating, and so on are slightly different across the versions—a common problem that increases the difficulty of working out what's really happening.

So what do we infer from this new test? Apart from the tiny anomaly with the 32 block reads, the cost no longer changes with the value of db_file_multiblock_read_count.

The first detail to worry about is that autotrace is not up to doing a thorough job. We need the complete description of the execution plan to help us, so we need a proper explain plan script (see plan_run92.sql in the online code suite). The columns we are particularly interested in from the latest version of the plan_table are cpu_cost, io_cost, and temp_space (this last will only be used for sorts and hashes that are expected to overflow to disk). When we report a more complete execution plan, we see the following:

```
SELECT STATEMENT (all_rows) Cost(5031,1,4) New(5001,72914400,0)
  SORT (aggregate)
    TABLE ACCESS (analyzed) T1 (full) Cost(5031,10000,40000) New(5001,72914400,0)
```

The three figures reported as `Cost( , , )` are equivalent to the original cost, cardinality, and bytes reported by `autotrace`. The three figures reported as `New ( , , )` are the `cpu_cost`, `io_cost`, and `temp_space` of the new `cpu_costing` algorithm; and the final cost is `io_cost` + (scaled) `cpu_cost`.

If we apply the formula quoted in Chapter 1, we can reverse engineer the arithmetic that the optimizer is doing:

```
Cost = (
   #SRds * sreadtim +
   #MRds * mreadtim +
   #CPUCycles / cpuspeed
) / sreadtim
```

By dividing the `sreadtim` all the way through the equation, we can rearrange this to read as follows:

```
Cost = (
   #SRds +
   #MRds * mreadtim / sreadtim +
   #CPUCycles / (cpuspeed * sreadtim)
)
```

Since we are doing a tablescan, we have

- `SRds` = 0 (single-block reads)
- `MRds` = 10,000 / 12 (Remember our code earlier claimed the size of a multiblock read was 12.)

The I/O Bit

We'll worry about the CPU component in a little while, but slotting in the 30 milliseconds and 5 milliseconds that we set for the `mreadtim` and `sreadtim`, the formula gives us an I/O cost of (10,000 / 12) * (30 / 5) = 5,000. And, of course, we have to remember that `_tablescan_cost_plus_one` is set to `true`, and we get to our target 5,001.

So we can see that when system statistics are active, the I/O cost of a tablescan uses the actual value of `MBRC` instead of an adjusted `db_file_multiblock_read_count`, and then caters to the difference in speed between a single-block read and a multiblock read by multiplying up by (recorded multiblock read time / recorded single-block read time).

If you have not collected system statistics in 10g, you will find that the optimizer makes use of three other statistics from the `aux_stats$` table:

```
PNAME                 PVAL1
--------------- ----------
CPUSPEEDNW         913.641      -- speed in millions of operations per second
IOSEEKTIM               10      -- disk seek time in milliseconds
IOTFRSPEED            4096      -- disk transfer time in bytes per millisecond
```

If the optimizer uses these noworkload statistics, it takes the preceding values, the db_block_size, and the db_file_multiblock_read_count, and synthesizes some values for the sreadtim, mreadtim, and MBRC.

- MBRC is set to the actual value of db_file_multiblock_read_count.
- sreadtim is set to ioseektim + db_block_size/iotrfrspeed.
- mreadtim is set to ioseektim + db_file_multiblock_read_count * db_block_size/iotftspeed.

In other words, a read request takes one seek and then however many milliseconds of transfer to shift the volume of data off disk. Using the preceding example with an 8KB block size and a multiblock read of 8, the optimizer would set the sreadtim to 10 + 8192/4096 = 12 ms, and mreadtim to 10 + 8 * 8192/4096 = 26 ms.

Once these values have been synthesized (the values are not stored back into the aux_stats$ table), the rest of the calculation proceeds as previously discussed. But, inevitably, there are complications—for example, what happens when you change the db_file_multiblock_read_count? Table 2-3 compares the effect of noworkload system statistics with *normal* system statistics and traditional costing for different values of db_file_multiblock_read_count (see script tablescan_03.sql in the online code suite).

This example uses the same 10,000 row table of my first example, with one row per block. For the standard cpu_costing figures, I have deliberately set the MBRC, sreadtim, and mreadtim to mimic the values derived from the noworkload statistics.

Table 2-3. *Effects of the 10g Workload Statistics*

db_file_multiblock_read_count	Traditional	Standard cpu_costing	Noworkload cpu_costing
4	2,397	2,717	3,758
8	1,519	2,717	2,717
16	963	2,717	2,196
32	611	2,717	1,936
64	388	2,717	1,806
128	246	2,717	1,740

In the first column of results, we see the traditional costing and, as expected, the cost goes down as the size of the multiblock read goes up. In the second column, we see the effect of the standard cpu_costing mechanism—the cost is dictated by a fixed MBRC and therefore does not change as we modify the multiblock read size. Finally, we see the strange variation that appears from the noworkload cpu_costing—the cost changes with the size of the multiblock read, though less extremely as the read-size gets very large.

The most important point about this variation in costing in 10*g* is that you need to know about it before you migrate. If you aren't using cpu_costing before you get to 10g, then either variant of the mechanism will need careful testing for side effects. If you move into 10*g* without

realizing that you are automatically going to be using cpu_costing (of the noworkload variety), you will have a second round of testing when you start to gather system statistics and switch to normal cpu_costing.

The slightly less important question (in this case) is *Why?* Why do the costs change for noworkload statistics? Remember that I pointed out that Oracle does not store the synthesized statistics—it re-creates them for every query (or perhaps every session). Let's work through the 4-block read as an example:

- MBRC = 4
- sreadtim = 10 + 2 = 12 milliseconds
- mreadtim = 10 + 4 * 2 = 18 milliseconds

We have the standard formula—and for the purposes of getting approximately the right answer, we'll just pick the bit about multiblock reads:

```
Cost = (
   #SRds +                              -- zero in this case
   #MRds * mreadtim / sreadtim +
   #CPUCycles / (cpuspeed * sreadtim)   -- ignore this for the moment
)

cost =   (1000/4) * 18/12 =
      2,500    * 1.5 =
      3,750
```

So the multiblock read *size* is smaller, but the synthetic multiblock read *time* is also smaller and we have to do more of them. I think the I/O component of 3,750 that we get from the preceding working is close enough to the observed value of 3,758 that we can postpone pursuing the CPU component again.

The interference between noworkload system statistics and the use of different block sizes is left as an exercise for you to do on your own. (But you might like to read script tablescan_04.sql from the online code suite.)

Of course, there are still lots of little details left to examine—which we can do by running various test cases. But here are a few answers:

When Oracle executes the tablescan, how many blocks does it try to read in a multiblock read? Is it the value of MBRC, the value of db_file_multiblock_read_count, or something else? Answer: Oracle still tries to use the actual value for db_file_multiblock_read_count—scaled up or down if we are reading from a tablespace with a nondefault block size. I actually had my db_file_multiblock_read_count set to 8, so it was silly to set the MBRC to 12, but for the purposes of the arithmetic the optimizer believed me, and then the run-time engine read the table 8 blocks at a time.

Where did the extra 1 come from in the I/O cost when db_file_multiblock_read_count was set to 32? I don't know. But I have come across a couple of other places in the optimizer code where modifying the db_file_multiblock_read_count produces a change in the cost that *obviously* should not happen. For the moment I'm going to assume that the odd 1 is a strange rounding error or minor aberration—until I find an example where the difference is significant and merits more attention.

The CPU Bit

The next important question is *How do you convert the* `cpu_cost` *quoted in the* `plan_table` *before adding it to the* `io_cost` *to produce the final cost, and where does the* `cpu_cost` *come from anyway?* To answer the first part, we look at the formula, our recorded values, and the literal value in the execution plan for the `cpu_cost`:

- The formula gives the CPU cost as: `#CPUCycles / (cpuspeed * sreadtim)`
- `CPUSPEED` = 500MHz
- `sreadtim` = 5 milliseconds = 5,000 microseconds (standardizing units of time)
- `#CPUCycles` (called `cpu_cost` in the `plan_table`) = 72,914,400

Slotting the figures in place: 72,914,400 / (500 * 5,000) = 29.16576. And the optimizer rounds CPU costs up in 9.2 (but not, apparently, in 10*g*), giving us the CPU cost of 30 that we wanted to see.

ROUNDING

Although the optimizer always used to round costs up, you will find that a new parameter called `_optimizer_ceil_cost` has appeared in 10*g* with a default value of `true`. But it seems that in this case, true means false, and false means true; and it may only apply to the CPU costs anyway.

Finding out exactly where the original count of 72,914,400 operations came from is much harder. If you care to run through a set of extremely tedious experiments, you could probably track it down—approximately—to details like these:

- Cost of acquiring a block = X
- Cost of locating a row in a block = Y
- Cost of acquiring the Nth (in our case the 2nd) column in a row = (N – 1) * Z
- Cost of comparing a numeric column with a numeric constant = A

And when you've got it all worked out, the constants will probably change on the next point release anyway.

However, to give you an idea of how powerful CPU costing can be—and how you may find that some queries run faster for no apparent reason after upgrading to 9*i*—I'd like to present an example of CPU costing in action.

The Power of CPU Costing

This example started life as a production issue, before being stripped back to a simple test case. The original code was running under 8*i*, and displaying an odd performance problem, so I re-created the data on a 9*i* database to see if the problem would go away—and it did. I then discovered that the problem had gone away because CPU costing allowed the optimizer in 9*i* to do something that the 8*i* optimizer could not do.

```
create table t1(
      v1,
      n1,
      n2
)
as
select
        to_char(mod(rownum,20)),
        rownum,
        mod(rownum,20)
from
        all_objects
where
        rownum <= 3000
;

--      Collect statistics using dbms_stats here
```

This is a simple table of three thousand rows. The data has been constructed (`cpu_costing.sql` in the online suite) to demonstrate two points. Notice that the `v1` and `n2` columns are only allowed to hold 20 different values and, apart from issues of data type, hold the same values. Column `n1` is unique across the table.

We now run three separate queries, and pass them through a proper `explain plan` so that we can isolate the CPU costs. I have used the hint `ordered_predicates` in these queries to force Oracle to apply the predicates in the order they appear in the `where` clause. In this test, 8*i* would have no option to do otherwise, but 9*i* can take a cost-based decision to re-order the predicates. (Note: This test uses the values for the system statistics that I quoted earlier in the chapter.)

```
select
        /*+ cpu_costing ordered_predicates */
        v1, n2, n1
from
      t1
where
      v1 = 1
and   n2 = 18
and   n1 = 998
;

select
        /*+ cpu_costing ordered_predicates */
        v1, n2, n1
from
        t1
```

```
where
        n1 = 998
and     n2 = 18
and     v1 = 1
;

select
        /*+ cpu_costing ordered_predicates */
        v1, n2, n1
from
        t1
where
        v1 = '1'
and     n2 = 18
and     n1 = 998
;
```

As you would expect, the execution plan in all three cases is a full tablescan, and if you use `autotrace` to see what's going on, you will discover only that the cost of the query (in all three cases) is 6. But if you use a proper query against the `plan_table`, reporting the columns `cpu_cost` and `filter_predicates` (yet another column that appeared in 9*i*), you will see the results summarized in Table 2-4.

Table 2-4. *Predicate Order Can Affect the Cost*

Predicate Order	CPU Cost	filter_predicates
v1 = 1	1,070,604	TO_NUMBER("T1"."V1")=1
and n2 = 18		AND "T1"."N2"=18
and n1 = 998		AND "T1"."N1"=998
n1 = 998	762,787	"T1"."N1"=998
and n2 = 18		AND "T1"."N2"=18
and v1 = 1		AND TO_NUMBER("T1"."V1")=1
v1 = '1'	770,604	"T1"."V1"='1'
and n2 = 18		AND "T1"."N2"=18
and n1 = 998		AND "T1"."N1"=998

It is the column `filter_predicates` that tells you exactly what is going on, and why the CPU cost can change even though the plan structure is the same. Based on the information it has about the different columns, including the low, high, number of distinct values, and so on, the optimizer in 9*i* is capable of working out (to a reasonable degree of accuracy) that the first query will require the following operations:

- Convert column v1 to a number 3,000 times, and compare—to produce 150 rows.
- For those 150 rows, compare n2 with a number—to produce 8 (technically 7.5) rows.
- For those 8 rows, compare n1 with a number.

That's 3,000 coercions and 3,158 numeric comparisons. On the other hand, the second query will require the following:

- Compare n1 to a numeric 3,000 times—to produce just one row (probably).
- For that one row, compare n2 with a number—to produce just one row (probably).
- For that one row, coerce v1 to a number and compare.

That's 3,002 numeric comparisons and a single coercion—which is a big saving in CPU. And if you take the ordered_predicates hint out of the first query, the optimizer automatically chooses to rearrange the predicates to match the order of the second query. (The default setting of true for parameter _pred_move_around gives you a hint that this can happen.)

Just as a final detail, the third query repeats the predicate order of the first query, but uses the correct string comparison, instead of an **implicit conversion**. Eliminate those 3,000 conversions and the CPU cost of the query drops from 1,070,604 to 770,604, a difference of exactly 300,000. This tends to suggest that the to_number() function has been given a cost of 100 CPU operations.

The BCHR Is Dead! Long Live the BCHR!

The next big change isn't in production yet—but the signs are there in 10*g* that something dramatic is still waiting in the wings. (The section heading isn't intended to be serious, by the way.)

It has become common knowledge over the last few years that the **buffer cache hit ratio** (**BCHR**) isn't a useful performance **target**—although plotting **trend** lines from regular snapshots can give you important clues that something is changing, or that a performance anomaly occurred at some point in time.

One of the biggest problems of the buffer cache hit ratio is that it is a system-wide average, and one statistically bizarre object (or query) can render the value completely meaningless. (For a demonstration of this fact, you can visit Connor McDonald's web site, www.oracledba.co.uk, to download the "Set Your Hit Ratio" utility).

But what if you track the cache hit ratio for every individual object in the cache—updating the figures every 30 minutes, maintaining rolling averages and trend values? Could you do anything useful with such a precisely targeted data collection?

One of the defects in early versions of the CBO is that the arithmetic works on the basis that every block visit goes to a data block that has never been visited before and therefore turns into a physical disk read. There were a couple of parameters introduced in 8*i* to work around the problems this caused (optimizer_index_cost_adj—which you will meet in Chapter 4, and optimizer_index_caching—which you will also see in Chapter 4, and again in Chapter 11). But in 9*i*, Oracle collects statistics about logical block requests and physical disk reads for every data segment.

In fact, Oracle seems to have several ways of collecting cache-related statistics. 9*i* introduced the dynamic performance view v$segstat, and 10*g* expanded the range of statistics the

view holds. Here, for example, is a query to discover the activity against a particular data segment (the `t1` table used in earlier examples in this chapter) in a 10*g* database:

```
select
        *
from    v$segstat
where   obj# = 52799
;
```

```
  TS#       OBJ#   DATAOBJ# STATISTIC_NAME             STATISTIC#      VALUE
----- ---------- ---------- -------------------------- ---------- ----------
    4      52799      52799 logical reads                       0      21600
    4      52799      52799 buffer busy waits                   1          0
    4      52799      52799 gc buffer busy                      2          0
    4      52799      52799 db block changes                    3        240
    4      52799      52799 physical reads                      4      23393
    4      52799      52799 physical writes                     5      10001
    4      52799      52799 physical reads direct               6          0
    4      52799      52799 physical writes direct              7      10000
    4      52799      52799 gc cr blocks received               9          0
    4      52799      52799 gc current blocks received         10          0
    4      52799      52799 ITL waits                          11          0
    4      52799      52799 row lock waits                     12          0
    4      52799      52799 space used                         14          0
    4      52799      52799 space allocated                    15   82837504
    4      52799      52799 segment scans                      16          4
```

This wealth of information is invaluable for troubleshooting—note particularly the items identifying contention points (`buffer busy waits`, `gc buffer busy`, `ITL waits`, `row lock waits`) and the explicit finger-pointing at tablescans and index fast full scans (`segment scans`).

In passing, the missing values of 8 and 13 are deliberately excluded by the view definition, and refer to aging timestamp and service ITL waits, respectively—the latter may be quite interesting for index **block splits**.

V$SEGSTAT BUGS IN 9*i*

Although `v$segstat` is an extremely valuable view for performance monitoring (taking the usual snapshots over short time intervals, of course), it has a couple of bugs in 9*i*.

In earlier versions, when you `truncate` or `move` a table, the `data_object_id` (physical ID) of a segment changes—however, the statistics for the object are not cleared from `v$segstat`. In later versions, the table statistics will be cleared, but the defunct statistics for the corresponding indexes are not.

Moreover, in current versions (9.2.0.6 and 10.1.0.4 as I write), virtually any query against this view or its underlying `x$` seems to result in an irretrievable memory leak of around 15KB from the SGA—unless the query contains an `order by` clause—which could eventually result in the instance effectively stopping with chronic `ORA-04031` errors.

For the purposes of the optimizer, though, the interesting statistics appear to be `logical reads` and `physical reads`. I have to admit that I was a little surprised by the fact that my example showed more `physical reads` than `logical reads`. The `logical reads` statistic is sampled, though, and this may explain the discrepancy, especially since my use of this object had been a little extreme. (Another alternative is that the `sample` clause used by the `dbms_stats` package when estimating statistics on a table can result in a variant of the tablescan run-time mechanism that literally does report more physical I/Os than logical I/Os.)

If we have data that tells us what fraction of `logical reads` against an object usually turn into `physical reads`, and we derive a sensible metric from that information, perhaps we can fold that metric back into the next pass when we optimize a query against that object.

There is just one little problem with this suggestion. How do you figure out what Oracle is doing (or going to do) with this information? There are various bits of evidence lying around—in fact, there is too much evidence.

Consider the tables `sys.tab_stats$`, and `sys.ind_stats$`. Appearing in 10*g*, they both include columns called `obj#` and `cachehit`—apparently some sort of cache hit ratio at the (logical) object level. Neither of these tables seems to be populated by default (yet), although they do appear in various recursive queries.

Consider also the table `sys.cache_stats_1$`, with the following interesting columns picked out of its description (notice particularly the `inst_id`—different **RAC** instances could have different, localized cache hit ratios).

```
SQL> desc cache_stats_1$
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 DATAOBJ#                                  NOT NULL NUMBER
 INST_ID                                   NOT NULL NUMBER
 CACHED_AVG                                         NUMBER
 CACHED_SQR_AVG                                     NUMBER
 CHR_AVG                                            NUMBER
 CHR_SQR_AVG                                        NUMBER
 LGR_SUM                                            NUMBER
 LGR_LAST                                           NUMBER
 PHR_LAST                                           NUMBER
```

This looks much more like a sensible attempt to keep some sort of rolling metric about caching and segment-based cache hit ratios—and we find in support of this table a complex `merge` statement being executed at regular intervals by `mmon` (the **manageability monitor process**) to update this table.

A quick check of the `dbms_stats` package also shows that an option to gather cache statistics has been added to many of the stats gathering procedures. And you will find that these procedures access the `cache_stats_1$` table if you run something like this:

```
execute dbms_stats.gather_table_stats(user,'t1',stattype => 'cache')
```

Finally, note the two hidden parameters:

`_cache_stats_monitor` (Default value `TRUE`)

`_optimizer_cache_stats` (Default value `FALSE`)

The first one enables collection of cache statistics, the second one enables the use of cache statistics for optimization—and since it can be set at the session level, it can't really hurt to try fiddling with it (on a test database).

```
set autotrace traceonly explain

alter session set "_optimizer_cache_stats" = true;
select count(*) from t1;

alter session set "_optimizer_cache_stats" = false;
select count(*) from t1;

set autotrace off
```

In this particular test, the cost of the tablescan with `_optimizer_cache_stats` set to false was 5,030 (this was following on from the 2KB block test earlier). When I switched `_optimizer_cache_stats` to `true`, the cost dropped to 5,025.

Unfortunately, I couldn't see any SQL accessing `cache_stats_1$` during the optimization phase when I enabled cache stats. On the other hand, the optimizer always accessed the empty `tab_stats$` and `ind_stats$` tables to optimize a new statement.

I'm still in the dark about what this scattered collection of observations means—but there seem to be at least two groups in Oracle Corporation who are working on ideas about including localized cache hit ratios into the optimizer. If or when the feature goes into production, it's sure to fix some existing problems; it's also likely to create some new ones. How, for example, do you work out what the execution plan was 24 hours after a performance problem has come and gone, when the localized cache hit ratios have changed or you're only allowed to run diagnostics on the UAT system where the localized cache hit ratios have nothing to do with the figures you get from production? (You may have to license the AWR whether you like it or not.)

Parallel Execution

Parallel query (or **parallel execution** as it became in 8*i*) is another feature of the Oracle database where a seemingly small change in the costing strategy produces a dramatic change in the cost calculations. The best place to see this is in our nice simple tablescan; and the change for this simple operation is so dramatic that you won't have any trouble believing how much difference it could make to a more complex query.

Re-create the table from our first test, and run the following queries against it with `autotrace` enabled. Repeat the following queries in 8*i*, 9*i*, and 10*g*, first with system statistics disabled (script `parallel.sql` in the online code suite) and then using the system statistics defined earlier in the chapter (see script `parallel_2.sql` in the online code suite):

```
select /*+ parallel(t1,1) */ count(*) from t1;
select /*+ parallel(t1,2) */ count(*) from t1;
select /*+ parallel(t1,3) */ count(*) from t1;
select /*+ parallel(t1,4) */ count(*) from t1;
select /*+ parallel(t1,5) */ count(*) from t1;
select /*+ parallel(t1,6) */ count(*) from t1;
select /*+ parallel(t1,7) */ count(*) from t1;
select /*+ parallel(t1,8) */ count(*) from t1;
```

Assuming you have parameter `parallel_max_servers` set to at least 8, you should get results similar to those in Table 2-5 for the cost of the query.

Table 2-5. *Effects of different degrees of parallelism*

Degree	8*i*	9*i* (I/O)	10*g* (I/O)	9*i* (CPU)	10*g* (CPU)
Serial	1,518	1,519	1,519	5,031	5,030
2	1,518	760	844	2,502	2,779
3	1,518	507	563	1,668	1,852
4	1,518	380	422	1,252	1,389
5	1,518	304	338	1,002	1,111
6	1,518	254	282	835	926
7	1,518	217	242	716	794
8	1,518	190	211	627	695

Ignore, for the moment, the last two columns, which are reporting the results with CPU costing enabled, and focus on the first three sets of costs. You will notice that the values for 9*i* and 10*g* don't quite agree with each other, but at least they are reasonably close to (serial cost / degree of parallelism). But the most obvious feature is that the cost in 8*i* never changes, no matter what the degree of parallelism.

Essentially, 8*i* costs and optimizes your query for the best serial path, and then runs it in parallel. On the other hand, 9*i* assumes that it can execute a completely collision-free 100% parallel run, effectively optimizing for a set of data downsized by a factor of (degree of parallelism). Allowing for small rounding errors, the numbers from 10*g* suggest that it has introduced a *parallel efficiency factor* of 90% in the arithmetic. The actual figures suggest the following version-dependent formulae for a parallel tablescan:

8*i* Cost at degree N = serial cost

9*i* Cost at degree N = `ceil(serial cost / N )`

10*g* Cost at degree N = `ceil(serial cost / (0.9 * N))`

This leaves you with three issues to consider. First, 8*i* is not optimizing parallel queries properly. Second, as you upgrade, costs of parallel execution can change dramatically, and you may not be able to predict the side effects. Third, the arithmetic used by 9*i* assumes that there will be absolutely no interference between parallel execution slaves—and that's not going to be very realistic in most cases.

The change between 8*i* and 9*i* is controlled by parameter `optimizer_percent_parallel` (hidden from 9*i* onwards as `_optimizer_percent_parallel`). In 8*i*, the default value was zero—in other words, calculate as serial. In 9*i* the default value is 101—which causes Oracle to perform its calculations on the basis of 100% parallelism.

You can, in theory, set the parameter to any value between 0 and 101. If you do, then the optimizer does a straight-line interpolation between the serial cost (the `Resc` figure in a

10053 trace) and the parallel cost at full parallelism (the Resp figures in a 10053 trace), and then picks the point along the line from serial to parallel that represents your required percentage. (The same interpolation mechanism applies to sorting and to some aspects of hash joins, but the simple tablescan is the best way to demonstrate the arithmetic.)

For example, from the table of 9*i* results, the cost of parallel degree 4 is 380, and the cost of a serial scan is 1519. If we set _optimizer_percent_parallel to 75, then the final cost would be calculated as 25% of 1519 plus 75% of 380, which comes to 664.75. You might note from the table of results that running parallel 4 at 75% does not give you the same cost as running parallel 3 at 100%.

Yet another consideration comes into play in 9*i* and 10*g* if you set the parameter parallel_adaptive_multi_user to true (and this is the default value for 10*g*). When this is set, only a limited number of users are allowed to run at the default degree of parallelism—and on my test systems, this limit was one in 9*i*, and two in 10*g*. So the optimizer is effectively costing parallel execution on the basis that no more than one or two parallel queries are going to be running at any one moment. The limit is set by the hidden parameter _parallel_adaptive_max_users, but don't fiddle with it as it's one of the parameter values used by Oracle at startup time to calculate the value of parallel_max_servers, so it will have side effects.

Other oddities occur with the costing of parallel tablescans. When you have not enabled system statistics (CPU costing), the calculation uses the same *adjusted dbf_mbrc* as the serial scan. But parallel scans are **direct path** reads—in other words, reads that bypass the **data buffer**—so they don't have to worry about the side effects of catering to blocks that are already in the cache, and therefore it is almost inevitable that they will be the full db_file_multiblock_read_count size. The calculation does not cater to this different mechanism.

PARALLEL SCANS AND DIRECT PATH READS

Parallel scans use direct path reads to bypass the data buffer and read blocks directly into local (PGA) memory. This helps to reduce the impact on the data buffer (but might mean you want a small Oracle buffer and a large file system buffer in some special cases).

But if the block in the data buffer is dirty (newer than the block on disk), then you might think a direct read would not see the latest version, and may therefore get the wrong result. To solve this problem, a parallel query will first issue a **segment checkpoint** to get all dirty blocks for the segment written to disk before it reads. (The cost is exposed through statistic DBWR parallel query checkpoint buffers written in 10*g*, and otherwise indicated by **enqueues** of type TC.)

This could lead to a performance problem in rare cases that mixed a large data buffer, a busy **OLTP** system, and parallel execution for reports—the work done by the database writer (DBWR) walking the **checkpoint queue** to find the relevant dirty blocks could have an undesirable impact on the OLTP activity.

Now take a closer look at columns 5 and 6 in the table, the set of results with system statistics (CPU costing) enabled. The thing that stands out most clearly is that the optimizer is using the stored value for the MBRC statistic in the calculation. And again we can see that 10*g* has introduced a 90% factor into the calculation.

But this raises another issue. Again, Oracle will use the actual value of parameter `db_file_multiblock_read_count` to do direct reads against the table, so the cost based optimizer is apparently using an unsuitable value in the cost calculation. In this case, though, there is a saving grace. If all you do are large, parallel tablescans, then the code that generates the `MBRC` will see nothing but the effects of your direct path multiblock reads, so the value gathered by `dbms_stats.gather_system_stats` will be the correct value. On the downside, of course, if you are running a mixed OLTP/**DSS** system with lots of multiblock reads that aren't parallel, then the value for `MBRC` is likely to be unsuitable for both the serial and parallel queries, falling somewhere between the two ideal values.

There is one other cunning little detail to CPU costing that can be seen most clearly in the 9*i* figures for parallel two. The cost drops from 5,031 to 2,501 as we go from serial to parallel two—but as we saw earlier on, 5,031 is the sum of 5,001 for the I/O cost and 30 for the CPU cost. When we go parallel, the optimizer "loses" the CPU cost. In fact, examination of the trace files for 9*i* and 10*g* show that there is a small component of CPU cost in place. Is this a bug? Probably not. A large fraction of the CPU cost of accessing a row comes from the CPU cost of locating, latching, and pinning a buffered block—but parallel queries don't use the cache, they do direct reads, so perhaps most of the CPU cost should disappear.

Index Fast Full Scan

The most important point to remember about the index fast full scan is simply that it exists as a possible execution plan. It doesn't appear very often in the current versions of the optimizer, but it is a path that can appear without being hinted.

In effect, for a query that references just a set of columns in an index, Oracle can decide to treat an index like a skinny table with a few bits of garbage (such as stored **rowids** and meaningless **branch blocks**) mixed in. This means Oracle can read the index segment in physical block order, using multiblock reads, throwing away the branch blocks as it goes. The index entries will not be returned in index order as Oracle will not be walking from **leaf block** to leaf block following the usual pointers, but in theory the cost of any sorting that may subsequently be needed will be outweighed by the benefit of getting the data off disk more quickly.

■ **Note** Index fast full scans are just like tablescans. They use multiblock reads and "large" indexes (defined by the same 2% of the block buffer count at startup) that are loaded into the discard end of the LRU list. They also suffer from a bug that affects tablescans: the buffer **touch counts** are not incremented (even for "small" indexes/tables) if the block has been loaded by an index fast full scan. This bug has been fixed in 10*g*.

For example, re-creating the first data set, we could do the following (see script `index_ffs.sql` in the online code suite):

```
create index t1_i on t1(val);
execute dbms_stats.gather_table_stats(user,'t1',cascade=>true);

set autotrace traceonly explain

select
        count(*)
from
        t1
where
        val > 100;

Execution Plan (9.2.0.6)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=5 Card=1 Bytes=4)
1  0   SORT (AGGREGATE)
2  1     INDEX (FAST FULL SCAN) OF 'T1_I' (INDEX) (Cost=5 Card=3739 Bytes=14956)
```

Unsurprisingly, you will find that the cost of an index fast full scan uses the same arithmetic as the cost of a full tablescan. However, the results often seem to be just slightly wrong; which might make you ask where the optimizer finds the number that it uses for the block count that is needed for the tablescan formula.

When you generate statistics for a table, one of the results is the number of blocks below the high water mark. When you generate the statistics for an index, you get the number of leaf blocks and the index `blevel`; but you don't get any information about the number of branch blocks, or the index segment's HWM.

So what number does the optimizer use as the basis for the cost of the index fast full scan? The answer seems to be the number of leaf blocks—which is fairly reasonable, because in a nice, clean randomly generated index, with no catastrophic updates and deletes, the number of leaf blocks is probably within 1% of the total number of blocks below the high water mark. Strangely, if you have not collected statistics on an index, Oracle uses its knowledge of the high water mark from the index's segment header block to get the right answer.

It is possible though that in some scenarios you could push an index into an unusual state where the number of populated leaf blocks was much smaller than the number of blocks below the high water mark (unlucky or inappropriate use of the `coalesce` command could have this effect—sometimes it really is a good idea to rebuild an index), resulting in an inappropriate index fast full scan cost when a range scan would be more efficient.

INVESTIGATIVE TECHNIQUES

You may wonder how I came to the conclusion that the index fast full scan cost was dictated by the statistic `leaf_blocks`.

Initially, instead of continually re-creating the same index with different values for `pctfree`, I used packaged procedure `dbms_stats.set_index_stats` to change the `leaf_blocks` value for an index to see what happened to the cost of the query. Then I changed the `blevel`, and so on. The script `hack_stats.sql` in the online code suite demonstrates the method.

This type of problem can have surprising side effects. If you have "emptied" a range of leaf blocks, then the `analyze` command reports the leaf block count as the number of leaf blocks currently in the index structure, whereas the procedure `dbms_stats.gather_index_stats` reports the count as the number of leaf blocks that actually have data in them. (When all the rows have been deleted from a leaf block, it is temporarily left in place in the tree, but also attached to the index segment's **freelist**.) When you switch from using the deprecated `analyze` to the strategic `dbms_stats`, you may find that some SQL statements suddenly start using index fast full scans "for no apparent reason."

This problem could only appear in queries (or execution plan lines) that could be satisfied completely within an index, and so probably won't be very common at present. But you never know what exciting new feature the next release of the optimizer might introduce and whether a problem might suddenly appear. It's surprising how often an enhancement in the optimizer helps 99 people out of 100—and causes major headaches for the odd one out. Here, for example, is an execution plan that is perfectly reasonable (in the right circumstances) but is not yet available to the optimizer:

```
Execution Plan (? 11.1.0.0 ?)
-----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=30 Card=18 Bytes=144)
1  0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=30 Card=18 Bytes=144)
2  1     SORT (ORDER BY)
3  2       INDEX (FAST FULL SCAN) OF 'T1_I' (INDEX) (Cost=12 Card=18)
```

The plan (a *complete fiction*, I must stress) starts with a fast full scan of an index to find a list of rowids based on nonleading columns, sorts by block ID, and then accesses the table. If you had an index that was much smaller than its base table, and you were after just a few rows meeting some criteria that could be tested in the index (but not identified by the leading edge of the index), then this could, in theory, be a better execution plan than an **index skip scan**.

At present, you could get quite close to emulating this path with SQL like the following (also in script `index_ffs.sql` in the online code suite):

```
select
        /*+ ordered no_merge(tb) use_nl(ta) rowid(ta) */
        *
from
        (
          select /*+ index_ffs(t1) */
                 rowid
          from t1
          where val > 250
          order by rowid
        )    tb,
        t1   ta
where
        ta.rowid = tb.rowid
;

Execution Plan (9.2.0.6)
```

```
------------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1834 Card=1818 Bytes=209070)
1  0  NESTED LOOPS (Cost=1834 Card=1818 Bytes=209070)
2  1    VIEW (Cost=16 Card=1818 Bytes=12726)
3  2      SORT (ORDER BY) (Cost=16 Card=1818 Bytes=19998)
4  3        INDEX (FAST FULL SCAN) OF 'T1_I' (INDEX) (Cost=5 Card=1818 Bytes=19998)
5  1     TABLE ACCESS (BY USER ROWID) OF 'T1' (TABLE) (Cost=1 Card=1 Bytes=108)
```

If the optimizer ever acquires this new access path, then some people may suddenly have problems on an upgrade because they fall into that unhappy band of DBAs who have indexes where the number of leaf blocks in use is always far less than the number of blocks below the high water mark. Suddenly some people really could have a good reason for regularly rebuilding a few indexes.

Partitioning

The critical issue with partitioned objects can best be demonstrated with a simple example where we only have to look at the cardinality (row estimates), rather than the cost of the execution plan (see script `partition.sql` in the online code suite):

```
create table t1 (
      part_col   not null,
      id         not null,
      small_vc,
      padding
)
partition by range(part_col) (
      partition   p0200 values less than ( 200),
      partition   p0400 values less than ( 400),
      partition   p0600 values less than ( 600),
      partition   p0800 values less than ( 800),
      partition   p1000 values less than (1000)
)
nologging as
with generator as (
      select   --+ materialize
               rownum    id
      from     all_objects
      where    rownum <= 5000
)
select
        trunc(sqrt(rownum-1)),
        rownum-1,
        lpad(rownum-1,10),
        rpad('x',50)
from
        generator   v1,
```

```
        generator   v2
where
        rownum <= 1000000
;

--      Collect statistics using dbms_stats here
```

I have used **subquery factoring** (a specific 9*i* feature) to generate a large table, in this case with the `materialize` hint to stop Oracle from rewriting the subquery named `generator` as an in-line view and then optimizing a big messy statement. Even a minimalist database install will have more than 3,000 rows in view `all_objects`, so this type of trick could be used to produce a table of up to 9,000,000 rows (900,000,000 or more, possibly, on a full Java install) with just two copies of the factored subquery.

By generating the partitioning column with the `trunc(sqrt())` function, I have conveniently managed to build a partitioned table where the number of rows in each partition gets larger and larger as you move up the partitions.

```
select
        partition_name,
        num_rows
from
        user_tab_partitions
order by
        partition_position
;

PARTITION_NAME         NUM_ROWS
-------------------- ----------
P0200                    40,000
P0400                   120,000
P0600                   200,000
P0800                   280,000
P1000                   360,000
```

After building the table and calling `dbms_stats.gather_table_stats` to compute table statistics, I ran three queries against the table, and checked their execution plans carefully. The first query used literal values and had a predicate that restricted it to one specific partition. The second query used literal values, but the predicate ranged across two consecutive partitions. The last query was the same as the second query but used **bind variables** instead of literals.

■ **Note** `Autotrace` is often sufficient to do a quick check on what the optimizer is doing, but it does have limitations. One of the most serious limitations appears with partitioned tables, when `autotrace` fails to report three of the critical `plan_table` columns that help you to understand how effective your partitioning strategy has been. This issue has been addressed in 10*g* release 2 where `autotrace` has been recoded to call the `dbms_xplan` package.

These are the queries—each followed by the execution plans provided by dbms_xplan from 9.2.0.6, with a description of how the optimizer has worked out the cardinality (Rows) column.

```
/*+ Query 1 - one partition */

select  count(*)
from    t1
where   part_col between 250 and 350
;

-------------------------------------------------------------------------------------
| Id  | Operation            |  Name      | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |            |     1 |     4 |   193 |       |       |
|   1 |  SORT AGGREGATE      |            |     1 |     4 |       |       |       |
|*  2 |   TABLE ACCESS FULL  | T1         | 61502 |   240K|   193 |     2 |     2 |
-------------------------------------------------------------------------------------

   2 - filter("T1"."PART_COL">=250 AND "T1"."PART_COL"<=350)
```

The optimizer has identified partition 2 as the single partition that will be hit—note the values for pstart and pstop. If we check the statistics for the part_col for this one partition, we find that there are 120,000 rows in the partition, the column data varies from 200 to 399, and it has 200 distinct values.

Of the available data range, we want the range 250 to 350, which is about 120,000 * (350 – 250) / 199, plus an extra 120,000 * 2 / 200 that Oracle adds in because our range is *closed* at both ends. (I will cover this in detail in Chapter 3.) Total row count is therefore 120,000 * ((100/199) + 1/100) = 61,502 (rounding up from 61,501.5).

So, with a known single partition at **parse** time, Oracle has used the partition-level statistics.

```
/*+ Query 2 - multiple partitions */

select  count(*)
from    t1
where   part_col between 150 and 250
;

-------------------------------------------------------------------------------------
| Id  | Operation                |  Name | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |       |     1 |     4 |   257 |       |       |
|   1 |  SORT AGGREGATE          |       |     1 |     4 |       |       |       |
|   2 |   PARTITION RANGE ITERATOR|       |       |       |       |     1 |     2 |
|*  3 |    TABLE ACCESS FULL     | T1    |   102K|   398K|   257 |     1 |     2 |
-------------------------------------------------------------------------------------

3 - filter("T1"."PART_COL">=150 AND "T1"."PART_COL"<=250)
```

We have crossed a partition boundary—Oracle has noted that we will be hitting partitions 1 and 2. Checking the table and column statistics, we find that between them the partitions have 160,000 rows, and the column has 400 distinct values with a range from 0 to 399, from which we want the range 150 to 250. Let's apply the same formula as last time: 160,000 * ((250 – 150)/399 + 2/400) = 48,100. The result is not even close.

We could check to see whether Oracle has done two sets of arithmetic, one for the range 150 <= part_col < 200 and one for 200 <= part_col <= 250, and then added them. It hasn't—this would give the result 40,800.

The answer comes from the table-level statistics. There are 1,000,000 million rows, with 1,000 different values, and a range of 0 to 999, giving us 1,000,000 * ((250 –150) / 999 + 2/1,000) = 102,100.

With multiple-known partitions at parse time, Oracle uses the table-level statistics.

```
/*+ Query 3 - multiple partitions, using binds */

variable v1 number
variable v2 number

execute :v1 := 150; :v2 := 250

select  count(*)
from    t1
where   part_col between :v1 and :v2
;

-------------------------------------------------------------------------------------
| Id  | Operation                  | Name | Rows  | Bytes | Cost  | Pstart| Pstop |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |      |     1 |     4 |  1599 |       |       |
|   1 |  SORT AGGREGATE            |      |     1 |     4 |       |       |       |
|*  2 |   FILTER                   |      |       |       |       |       |       |
|   3 |    PARTITION RANGE ITERATOR|      |       |       |       |   KEY |   KEY |
|*  4 |     TABLE ACCESS FULL      | T1   |  2500 | 10000 |  1599 |   KEY |   KEY |
-------------------------------------------------------------------------------------

   2 - filter(TO_NUMBER(:Z)<=TO_NUMBER(:Z))
   4 - filter("T1"."PART_COL">=TO_NUMBER(:Z) AND "T1"."PART_COL"<=TO_NUMBER(:Z))
```

An estimate of 2,500 is not good, especially since we know the right answer is in the region of 40,000—so where has it come from? Starting from a position of incomplete knowledge (note in particular the `KEY - KEY` option for the partition start and stop in the plan), the optimizer has used table-level statistics (i.e., 1,000,000 rows). In this case, though, there is no information about actual values, so the optimizer has fallen back on some hard-coded constants—namely 0.25% for "between :bind1 and :bind2"; the 2,500 rows is 0.0025 * 1,000,000.

OVERUSING BIND VARIABLES

Oracle users have been known to go through fads from time to time. One of the more recent ones is the constant advice to use bind variable instead of literals because of the parsing and **latching** overhead you otherwise get. Enthusiasm for this strategy can go a little too far. In some cases, the optimizer will do a very bad job if it can't see literal values. As a guideline, heavy-duty reports that hit big tables for a lot of data probably should use literal values—the overhead of parsing is likely to be tiny compared to the work done by the query.

Even in OLTP systems, it would be perfectly reasonable to help the optimizer by the judicious use of literals occasionally, perhaps to the extent of having half a dozen versions of the same query that differed only in the value of one critical input.

Just to make things more complicated though, 9*i* and 10*g* employ bind variable peeking in most circumstances (but certainly not when running `explain plan` or `autotrace`). At run time the optimizer would have been able to peek at the incoming bind variables and use their values to produce a cardinality that was likely to be a little more appropriate to that set of values. Unfortunately, the very next use of the query could supply a completely different set of values—but the optimizer doesn't peek at them, it just runs with the plan it got from the first optimization. This could be an expensive error—and there's not a lot you can do about it, except know about it and code your way around it.

■ **Caution** I was recently sent a `10053 trace` file from 10.2 that suggested that the optimizer does not peek at the bind variable for prepared statements coming through the JDBC thin driver. MetaLink note 273635.1 also makes reference to this issue, but only with regard to the 8*i* version of the driver.

You will have noticed the strange **filter predicate** on line 2 of the last execution plan. If you try running this test in 8*i* and 9*i*, you will see the important difference that that predicate makes. Supply the bind variables as `:v1 = 300` and `:v2 = 250` (in other words, the wrong way around) for 8*i*, and if both predicates come from the same partition, Oracle will scan that partition—despite the fact that there will obviously be no data found. Repeat the test on 9*i*, and the extra predicate will automatically be false, and Oracle will not run the next line of the execution plan.

Returning to the issue of statistics, the problem of partition statistics and table-level statistics is a difficult one. The commonest use of partitioned tables involves a process that Oracle Corp. now calls **Partition Exchange Loading** (or **rolling partition maintenance**) where you load an empty table, index it and collect statistics on it, and then do a **partition exchange** using SQL like the following:

```
alter table pt1
    exchange partition p0999 with table load_table
    including indexes
    without validation
;
```

The problem is that this doesn't bring table-level statistics up to date (in fact, there have been many reports in the past about table-level statistics vanishing when partition maintenance took place). After you've done a partition exchange, you need to have a mechanism that brings the table-level statistics up to date—preferably without using an excessive amount of machine resources. You really have to know your data, and the most up-to-date version of the `dbms_stats` package, to do this efficiently.

The problem of partitions and table-level statistics echoes on down the chain to subpartitions. If you want to query exactly one subpartition of one partition, then the optimizer uses the statistics for that one subpartition. If you want to query several subpartitions from a single partition, the optimizer switches to the partition-level statistics for that one partition. If your query gets any messier in its selection, the optimizer will use the table-level statistics.

Summary

The cost of a tablescan is largely the cost of assumed multiblock reads. To calculate the cost, the optimizer divides the number of used blocks in the table (below the high water mark) by a number representing the assumed size of the multiblock read. (It was only when I wrote this line for the summary that I thought how nice it would be if the parameter included the word *size* instead of *count* and was expressed in KB.)

In 8*i*, this calculated count of the required number of multiblock reads is the cost.

In 9*i*, the introduction of system statistics allows the result to be adjusted by factors representing typical sizes and relative speeds of multiblock reads, and adding in the CPU cost of visiting the blocks and acquiring data from every row in the blocks.

And 10*g* holds some clues that at some future date the optimizer will also factor in recent cache history, which is likely to result in a reduced value for the cost.

For index fast full scans, the number of leaf blocks in the index, rather than the number of blocks below the high water mark, seems to be the driving value used in the calculation. In many cases, this will give a reasonable result. There are a couple of scenarios, though, where the number of leaf blocks can be much smaller than the number of blocks below the high water mark, and this would result in the cost of an index fast full scan being seriously underestimated.

Partitioned tables are likely to be a problem. The optimizer can use the statistics from a single partition if the partition can be identified at parse time; otherwise it uses the table-level statistics. And the same type of strategy appears with subpartitions of a single partition. In many cases, you may find that the only way to get the optimizer to produce sensible plans is to refresh the table-level statistics as you do **partition maintenance**. This may prove to be a resource-intensive approach.

Test Cases

The files in the download for this chapter are shown in Table 2-6.

Table 2-6. *Chapter 2 Test Cases*

Script	Comments
plan_run81.sql	Script to report all columns of the plan_table in 8.1
plan_run92.sql	Script to report all columns of the plan_table in 9.2
tablescan_01.sql	Builds simple test data set for tablescan costs
calc_mbrc.sql	Script to generate *adjusted dbf_mbrc* values
tablescan_01a.sql	Impact of block size on cost for a table fixed at 10,000 blocks
tablescan_01b.sql	Impact of multiple block sizes on cost for a table fixed at 80MB
set_system_stats.sql	Sample of code to set system statistics
tablescan_02.sql	Original test case modified to investigate normal CPU costing
tablescan_03.sql	Original test case modified to investigate noworkload CPU costing
tablescan_04.sql	Effects of noworkload CPU costing with multiple block sizes
cpu_costing.sql	Basic demonstration of predicates moving
parallel.sql	Reuses the first tablescan example for parallel testing
parallel_2.sql	Version of parallel.sql with system statistics enabled
index_ffs.sql	Simple demonstration of fast full scan costs on an index
hack_stats.sql	General-purpose script for modifying statistics on an object
partition.sql	Demonstration of how partitioned tablescans are costed
setenv.sql	Sets the standardized test environment for SQL*Plus

CHAPTER 3

Single Table Selectivity

After the chapter on **tablescans**, you may have expected a chapter on indexed access paths. But the predicted number of rows (**cardinality**) generated by an operation plays a crucial part in selecting initial join orders and optimum choice of indexes, so it is useful to have a good understanding of how the optimizer estimates the number of rows that are going to be produced at each step of a plan.

The reason why this chapter's title includes the term **selectivity**, rather than cardinality, is that the optimizer's calculations of cardinality are based on estimating the expected **fraction** of the rows in the current data set that would pass a particular test. That fraction is the number we call the selectivity. After you've worked out the selectivity, the cardinality is simply selectivity * (number of input rows).

Depending on the circumstances, one concept is sometimes more convenient, or intuitive, to use than the other, so I will switch between them fairly freely in the course of the book.

I have made a couple of comments about **histograms** in this chapter, but in most cases this chapter describes the calculations that the optimizer uses when there are no histograms in place. The effects of histograms are examined in Chapter 7.

Getting Started

At a recent conference, I managed to draw an audience of 1,200 people. How many of them do you think were born in December? If you've decided that the answer is about 100, then you've just performed a perfect imitation of the **CBO**. (I lied about the audience size to keep the numbers simple, by the way.)

```
There are 12 possible months in the year.                    -- known reference
Dates of birth are (probably) evenly scattered through the year.  -- assumption
One-twelfth of the audience will be born in any one month.   -- month's selectivity
The request was for one specific month.                      -- predicate
The requested month does actually exist in the calendar.     -- boundary check
There are 1,200 people in the audience.                      -- base cardinality
The answer is one twelfth of 1,200, which is 100.            -- computed cardinality
```

Let's turn the question into an SQL statement, and rerun the steps from the optimizer's perspective. We'll have a table called `audience`, with a column called `month_no` that uses the

numbers 1 to 12 for the months of the year. Being good DBAs, we have generated statistics for the data. Our query reads like this:

```
select  count(*)
from    audience
where   month_no = 12
;
```

Allowing for a little poetic license on my part, the optimizer will perform the following steps for the `month_no` column. Note how these steps follow the human line of thinking very closely, although they add in one check that human intuition allows us to forget. We examine figures from the view `user_tab_col_statistics` (or `user_tab_columns`) and check view `user_tab_histograms` to find the following details:

- `user_tab_col_statistics.num_distinct` is equal to 12.
- `user_tab_histograms` shows just the low (1) and high (12) values, so assume the values are evenly spread.
- `user_tab_col_statistics.density` is equal to 1/12; one month gives one twelfth of the data.
- `month_no` is equal to 12, single column, equality, so `user_tab_col_statistics.density` is usable.
- 12 is between the `low_value` and `high_value` of `user_tab_col_statistics`.
- `user_tab_col_statistics.num_nulls` is equal to 0 (everyone was born some time—the computer has to consider it, even though it was intuitively obvious to you).
- `user_tables.num_rows` is equal to 1,200.
- The answer is one-twelfth of 1,200, which is 100.

The audience is modeled in the script `birth_month_01.sql` in the online code suite, and the query for December birthdays produces the following execution plan:

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'AUDIENCE' (Cost=2 Card=100 Bytes=300)
```

Line 2 is the one we are interested in—this shows `Card=100`: Oracle has correctly inferred that there will be 100 rows in the base table that match our test. (The `Card=1` in line 0 reflects the fact that the final output we get from counting those rows is, indeed, a one-line answer.)

You might note a little oddity when you look at the statistics stored in the data dictionary.

```
select
        column_name,
        num_distinct,
        density
from
        user_tab_col_statistics
```

```
where
        table_name = 'AUDIENCE'
;

COLUMN_NAME          NUM_DISTINCT    DENSITY
-------------------- ------------ ----------
MONTH_NO                       12 .083333333
```

Oracle appears to be storing the same piece of information twice—the column called `num_distinct` (the number of distinct nonnull values) and the column called `density` (the fraction of data—ignoring rows with nulls—that would be returned by a query of the type `column = constant`). In our example, `num_distinct` is equal to 12 and `density` is equal to 1/12; and in general, you would probably notice that `density = 1 / num_distinct`. So why does Oracle appear to store the same information twice?

The two numbers are related in our example, but this is not always true. When you create a histogram on a column you will (usually) find that the density is no longer `1 / num_distinct` and, when histograms are in place, different versions of Oracle behave in slightly different ways. Use a different version to run the example, and you find the optimizer in 10*g* uses the `num_distinct` column to work out the result: `cardinality = num_rows / num_distinct`. If there had been a histogram in place, the optimizer would have used the `density` column: `cardinality = num_rows * density`.

To confirm this little detail, I used the packaged procedure `dbms_stats.set_column_stats` to change the `num_distinct` and `density` between two executions of the same query (see script `hack_stats.sql` in the online code suite). This showed that 8*i* always uses the `density`, but 9*i* (like 10*g*) uses the `num_distinct` if there is no histogram in place, and `density` if there is a histogram—although 9*i* does not pick up the changed values unless you flush the **shared pool** (a quirky little oddity and not really relevant unless you've implemented scripts to load business-specific statistics directly into the data dictionary).

ENHANCEMENTS AND PROBLEMS

Lots of little traps are always waiting for you when you upgrade. Here's one I found with the simple test scripts I wrote for the month of birth example.

When you use the packaged procedure `dbms_stats.gather_table_stats()`, it has a default value for the parameter `method_opt`. In 8*i* and 9*i*, the default was *for all columns size 1*, which translated into *Do not collect histograms*; in 10*g* the default is *for all columns size auto*.

Since my standard for testing code prior to the launch of 10g was to execute `dbms_stats.gather_table_state( user, 't1', cascade=>true )`, I found that some of my test results suddenly went "wrong" when I upgraded.

The default behavior is actually table-driven in 10*g*. The code in the `dbms_stats` package uses a procedure called `get_param` to look up the default value for some of the collection options.

You can alter the default behavior by using calls to the associated procedure `dbms_stats.set_param`. But I would be a little cautious about doing this. It would be easy to forget that you had done it, and this might cause a lot of confusion on the next upgrade, or on the next installation you did.

Null Values

Let's enhance our example—imagine that 10% of the members in our audience don't remember which month their birthday is in (and don't want to switch their PDAs on in the middle of an interesting presentation). How many people will put their hands up for December?

Assuming that birthday-aphasia is randomly distributed, there will now be 120 people who can't answer the question. These will be represented by a uniform scattering of nulls for the `month_no` column of our `audience` table. How does this affect the statistics and calculations? These are the figures we see when we query the data dictionary:

```
user_tables.num_rows is equal to 1,200                    -- no change
user_tab_col_statistics.low_value is equal to 1           -- no change
user_tab_col_statistics.high_value is equal to 12         -- no change
user_tab_histograms shows no histogram                    -- no change
user_tab_col_statistics.num_distinct is equal to 12       -- no change
user_tab_col_statistics.density  is equal to 1/12         -- no change
user_tab_col_statistics.num_nulls is equal to 120         -- new data item
```

The most important point to note is that the presence of null values does not change `num_distinct` (or `density`); the null values are simply ignored for the purposes of generating statistics. So what do we get as our result for people born in December?

A human argument would go something like this: if 100 people were born in December, but 10% of the audience don't remember when they were born, then, assuming a uniform distribution of amnesia, 90 of the 100 will remember that they were born in December.

The equivalent "argument" from the optimizer is

- Base selectivity = 1/12 (from `density` or from 1/`num_distinct`)
- `num_nulls` = 120
- `num_rows` = 1200
- Adjusted selectivity = Base selectivity * (`num_rows - num_nulls`) / `num_rows`
- Adjusted selectivity = (1/12) * ((1200 - 120)/1200) = 0.075
- Adjusted cardinality = Adjusted selectivity * `num_rows`
- Adjusted cardinality = 0.075 * 1200 = 90

So, we would expect the cardinality to be 90 for our query—and sure enough, when we run the test, the cardinality shows up as 90 (script `birth_month_02.sql` in the online code suite).

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'AUDIENCE' (Cost=2 Card=90 Bytes=270)
```

Using Lists

Once we know how to deal with the simple case of `column = constant`, we can move on to slightly more complex cases, such as queries involving lists, queries involving lists with nulls, queries involving two columns, queries involving ranges, and queries involving **bind variables**. There are plenty of cases to investigate before we worry about indexes and joins.

Let's start with the easiest option, **in-lists**. Sticking with the table that represents our audience of 1,200 people, we can write a query like the following:

```
select  count(*)
from    audience
where   month_no in (6,7,8)
;
```

Given that we've selected three months, and we expect 100 people per month, we shouldn't be too surprised if the calculated cardinality came out as 300. But I'm going to start my investigation with 8*i*, and unfortunately, the following execution plan is what we get when we build the test with that version of Oracle (see `in_list.sql` in the online suite):

```
Execution Plan (8.1.7.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'AUDIENCE' (Cost=1 Card=276 Bytes=828)
```

Where did that 276 in the full scan come from? (Using 9*i*, or 10*g*, the cardinality is reported at 300, as expected.) There are two points to consider here. First, the calculation is *obviously* wrong. Secondly, and more importantly, the calculation changes as you upgrade your version of Oracle—and as I pointed out at the start of this chapter, the correct cardinality is crucial to getting the correct join order and optimum choice of indexes. You may see some of your execution plans change "for no reason" when you upgrade.

IN-LIST ERRORS

Internally, the optimizer will convert a predicate like `month_no in (6,7,8)` to `month_no = 6 or month_no = 7 or month_no = 8`. If you add the `use_concat` hint to the query, the optimizer will then transform the plan into a `union all` of its three component parts—at which point 8*i* suddenly produces the correct cardinality.

The error that 8*i* suffers from is that after splitting the list into three separate predicates, it applies the standard algorithm for multiple **disjuncts** (the technical term for **OR**'ed predicates). This algorithm generically corrects for rows double-counted where the predicates overlap—but of course, the predicates generated from an in-list are guaranteed not to overlap. 8*i* transforms the SQL to produce a very special case of disjuncts, and then fails to process it correctly.

When an oddity like this shows up, you need to think of two things. First: is the example a special case? (Our column has only 12 distinct values—would the problem occur, or be so apparent, if there were more distinct values?) Second: are there closely related areas that might produce other oddities?

Let's spend a little time experimenting around these two questions to see what happens. The code to create the base table was as follows:

```
create table audience as
select
        trunc(dbms_random.value(1,13))        month_no
from
        all_objects
where
        rownum <= 1200
;
```

Let's change this to generate 1,000 possible "months" across 12,000 people (so that the optimizer should calculate 12 rows per "month number").

```
create table audience as
select
        trunc(dbms_random.value(1,1001))        month_no
from
        all_objects
where
        rownum <= 12000
;
```

We can then create a simple script that queries the tables with longer and longer in-lists (the online code suite holds separate scripts for the two data sets: `in_list.sql` uses the original 1,200 rows and `in_list_02.sql` uses 12,000 rows with 1,000 distinct values):

```
select count(*) from audience where month_no in (1,2);
select count(*) from audience where month_no in (1,2,3);
select count(*) from audience where month_no in (1,2,3,4);
. . .
select count(*) from audience where month_no in (1,2,3,4,5,6,7,8,9,10,11,12,13,14);

select count(*) from audience where month_no in (
         1, 2, 3, 4, 5, 6, 7, 8, 9,10,
        11,12,13,14,15,16,17,18,19,20,
        21,22,23,24,25,26,27,28,29,30
);
```

When we produce a table of results, as shown in Table 3-1, you can see that the extreme divergence that appears in the base case where we had only 12 values is much less obtrusive in the example with 1,000 different values. In fact, we see no divergence at all in the large example until the in-list has 14 items in it. Moreover, when you check the values for 9*i* and 10*g*, you see that the cardinalities always come out as N * number of entries in the list until the number of entries in the list exceeds the number of distinct values.

Table 3-1. *Small Lists of Values Can Give Big Cardinality Errors*

Size of List	Cardinality—12 Values 8*i* (9*i*, 10*g*)	Cardinality—1,000 Values 8*i* (9*i*, 10*g*)
1	100 (100)	12 (12)
2	192 (200)	24 (24)
3	276 (300)	36 (36)
4	353 (400)	48 (48)
5	424 (500)	60 (60)
6	489 (600)	72 (72)
7	548 (700)	84 (84)
8	602 (800)	96 (96)
9	652 (900)	108 (108)
10	698 (1,000)	120 (120)
11	740 (1,100)	132 (132)
12	778 (1,200)	144 (144)
13	813 (1,200)	156 (156)
14	846 (1,200)	167 (168)
30	1,112 (1,200)	355 (360)

So, there is an issue with in-lists, and this may cause problems for systems where the number of distinct values in a column is small—and when you upgrade those systems, you may see execution plans changing. But, in the general case, the error in the calculations may be insignificant. Let's move on to the issue of other oddities that we might consider after seeing this one. Will the optimizer do anything unexpected with the following predicates for the table where the `month_no` takes only values from 1 to 12 (see `oddities.sql` in the online code suite)?

```
where month_no = 25                 -- outside high_value
where month_no in (4, 4)            -- repeated values
where month_no in (3, 25)           -- mixed set of in and out values
where month_no in (3, 25, 26)       -- ditto
where month_no in (3, 25, 25, 26)   -- ditto with repeats
where month_no in (3, 25, null)     -- does the optimizer spot the null ?
where month_no in (:b1, :b2, :b3)   -- with or without bind-variable peeking ?
```

The results of running these tests against 8*i*, 9*i*, and 10*g* are shown in Table 3-2.

Table 3-2. *Boundary Cases with In-Lists*

Predicate	Cardinality (8i)	Cardinality (9i/10g)	
`month_no = 25`	100	100	Ouch
`month_no in (4, 4)`	100	100	Good
`month_no in (3, 25)`	192	200	Ouch, but consistent
`month_no in (3, 25, 26)`	276	300	Ouch, but consistent
`month_no in (3, 25, 25, 26)`	276	300	Ouch, but consistent
`month_no in (3, 25, null)`	276	300	Ouch, ouch
`month_no in (:b1, :b2, :b3)`	276	300	Ouch, ouch, but consistent

So, apart from the odd "double-counting" factor introduced by 8*i*, all three versions behave the same (sometimes odd) way with in-lists. They don't notice that values are outside the high/low range, they don't notice that the list may include nulls. They do notice when you have explicit duplicates in the list—but don't notice if those duplicates are hidden inside bind variables.

As a final observation on in-lists, you might like to experiment with `not in`, using queries such as

```
select  count(*)
from    audience
where   month_no NOT in (1,2)
;
```

Allowing for slight rounding errors, you will find that 8*i* is self-consistent. The cardinality for `month_no in {specific list}` plus the cardinality for `month_no not in {specific list}` comes to 1,200—the total number of rows in the table.

However, 9*i* and 10*g* are not self-consistent—although the cardinality of `month_no in {specific list}` changes as you migrate from 8*i* to 9*i*, the mechanism for calculating `month_no not in {specific list}` has not.

Further experimentation is left as an exercise for you to do on your own—but scripts `in_list_03.sql` and `pv.sql` are available in the online code suite as a starting point for investigation. The latter includes a note on the variations that appear between versions when you start to use in-lists to create partitioned views.

PARTITIONED VIEWS—DEPRECATED BUT IMPROVED

Partition views are a deprecated feature—but this is one of those strange cases where the term is deprecated but the technology is not. The code that used to handle partition views is no longer special, it's just a bit of code that handles views—and does it better in 10*g* than it ever did in Oracle 7, even when parameter `_partition_view_enabled` is set to `false`.

10*g* Update

It is a hopeless task trying to write a definitive book about the cost based optimizer—it changes faster than you can write (at least, faster than I can write). Since starting this volume, I have gone through the 10*g* beta, 10.1.0.2, 10.1.0.3, and as I write today, I am busy retesting everything on 10.1.0.4 (and 10.2 came out before the book went to print) and in-lists have changed. In fact, any selectivity outside the low/high range has changed.

Go back to the data set (see script `in_list_10g.sql` in the online code suite for a special variant) and try the following queries:

```
where month_no = 13          -- outside low/high
where month_no = 15          -- outside low/high
where month_no in (13,15)    -- two values outside low/high
where month_no in (16,18)    -- different 2 values outside low/high
```

The results you get from 10.1.0.4 are not the same as the results you get from 10.1.0.2, as demonstrated by Table 3-3.

Table 3-3. *Out-of-Bounds Behavior Changes Within Versions of 10g*

Predicate	Cardinality (10.1.0.2)	Cardinality (10.1.0.4)
`month_no = 13`	100	91
`month_no = 15`	100	73
`month_no in (13,15)`	200	164
`month_no in (16,18)`	200	109

The calculation that Oracle uses can be represented graphically, as shown in Figure 3-1—the further away you get from the known low/high range, the less likely you are to find data. Oracle uses a straight-line decay to predict the variation, decaying to zero when you exceed the range by the difference between low and high.

We have a range of 11—running from 1 to 12. The right-hand edge hits zero at `month_no = 23` (high value + 11); the left-hand edge hits zero at `month_no = -10` (low value – 11).

This will be bad news to some people. If you have sequence, or time-based, values in a commonly queried column and haven't been keeping the statistics up to date, then queries that use an equality on that column will have been using a flat line to give you the right answer as time passes. Now, those same queries are going to give you cardinalities that keep dropping as time passes—until suddenly, the cardinality gets so low that plans may change dramatically.

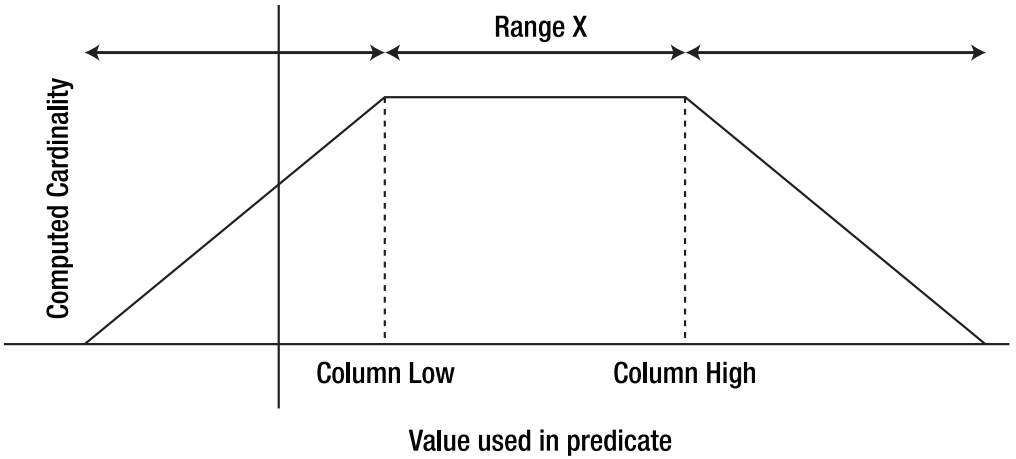

Figure 3-1. *10g enhancement of selectivity*

Range Predicates

You can put together plenty of variations for testing ranges, as a range can be bounded or unbounded, open or closed. Using the same basic test data of 100 rows for each of 12 months, I'll start with a table of examples (see script `ranges.sql` in the online code suite) and results from `explain plan`, Table 3-4, and discuss the arithmetic afterwards.

Table 3-4. *Variations in Cardinality for Range-Based Predicates*

Case	Predicate	Card (8*i*)	Card (9*i* /10*g*)	
1	`month_no > 8`	437	436	Unbounded, open
2	`month_no >= 8`	537	536	Unbounded, closed
3	`month_no < 8`	764	764	Unbounded, open
4	`month_no <= 8`	864	864	Unbounded, closed
5	`month_no between 6 and 9`	528	527	Bounded, closed, closed
6	`month_no >= 6 and month_no <= 9`	528	527	Bounded, closed, closed
7	`month_no >= 6 and month_no < 9`	428	427	Bounded, closed, open
8	`month_no > 6 and month_no <= 9`	428	427	Bounded, open, closed
9	`month_no > 6 and month_no < 9`	328	327	Bounded, open, open
10	`month_no > :b1`	60	60	Unbounded, open
11	`month_no >= :b1`	60	60	Unbounded, closed

Table 3-4. *Variations in Cardinality for Range-Based Predicates*

Case	Predicate	Card (8*i*)	Card (9*i*/10*g*)	
12	`month_no < :b1`	60	60	Unbounded, open
13	`month_no <= :b1`	60	60	Unbounded, closed
14	`month_no between :b1 and :b2`	4	3	Bounded, closed, closed
15	`month_no >= :b1 and month_no <= :b2`	4	3	Bounded, closed, closed
16	`month_no >= :b1 and month_no < :b2`	4	3	Bounded, closed, open
17	`month_no > :b1 and month_no < :b2`	4	3	Bounded, open, open
18	`month_no > :b1 and month_no <= :b2`	4	3	Bounded, open, closed
19	`month_no > 12`	100	100	Unbounded, open
20	`month_no between 25 and 30`	100	100	Bounded, closed, closed

As you can see, the results from 8*i*, 9*i*, and 10*g* are very similar to each other. The only differences are due to rounding that is sometimes introduced by **computational issues** and is sometimes due to something that appears to be a code change in the optimizer's choice of rounding.

The cases for bind variables (14–18) where 8*i* reports the cardinality as 4, rather than 3, are due to computational errors. The answer should be exactly 3, but in the translation from decimal to binary and back, a spurious digit has appeared somewhere past the 15th decimal place. 8*i* has rounded up; 9*i* and 10*g* have rounded to the closest integer.

The other differences occur where the arithmetic produces fractions like 0.36 or 0.27, and again 8*i* has rounded up, while 9*i* and 10*g* have simply rounded.

CARDINALITY

Any error (or change) in cardinality can have significant side effects in the join order and choice of indexes. Counterintuitively, a change in cardinality from 4 to 3 is much more likely to have a significant impact on an execution plan than a change from 537 to 536. Precision for smaller tables (more specifically, smaller result sets) is very important.

Glancing through the table of results, it is possible to pick out a couple of patterns.

- When we use literals, the difference between **open** (greater than [>], less than [<]) and **closed** (greater than or equal to [>=], less than or equal to [<=]) is exactly 100.
- When we go right outside the legal range of values for the columns (`user_tab_col_statistics.low_value`, `user_tab_col_statistics.high_value`), the cardinality seems to fix itself at 100.

- The values reported for bind variables seem to be rigidly fixed (no difference between open and closed) and have little to do with realistic possibilities (a range returning 60 rows seems unlikely when an individual value returns 100 rows).

With a little work (and a few follow-up experiments), we can guess the calculations that the optimizer is probably doing. Taking the second half of the table of results first:

- **Cases 10 through 13 (bind variables with unbounded ranges)**: The optimizer simply sets the selectivity at 5%. With 1,200 rows (and no nulls), we get 0.05 * 1200 = 60 rows.
- **Cases 14 through 18 (bind variables with bounded ranges)**: The optimizer simply sets the selectivity at 0.25% (which is actually 5% of 5%). With 1,200 rows (and no nulls), we get 0.0025 * 1200 = 3 rows.
- **Cases 19 and 20 (ranges outside the recorded low/high range)**: The optimizer detects that the query falls outside the known range, and seems to return a selectivity, hence cardinality that would be correct for `column = constant`. (There is a nasty boundary case, though, that appears from 9*i* onwards when every row holds the same value. The script `selectivity_one.sql` in the online code suite gives an example.)

BIND VARIABLES AND RANGES

One oddity with bind variables and ranges: you might expect `character_col like :bind` to be treated the same way as a `between` clause, after all, `colX like 'A%'` looks as if it should be treated like `colX >= 'A' and colX < 'B'`—which it nearly is. In fact (see script `like_test.sql` in the online code suite), when the optimizer sees this comparison with a bind variable, it uses the same 5% selectivity as it does for an unbounded range—with the usual caveat about the effects of **bind variable peeking**.

The predicates that use literal values need a little more explanation. The informal, and approximate, version of the optimizer's algorithm reads as follows:

Selectivity = "required range" divided by "total available range"

By looking at (`user_tab_col_statistics.high_value – user_tab_col_statistics.low_value`), we can calculate that the total range in our example is 11. And as soon as you see the number 11, you know that something is going to go wrong with our test case. We know that the test case is about 12 discrete measurements, but the optimizer uses arithmetic that treats the data as if it were continuously variable values with a total range of 11.

Anything that the optimizer does with small numbers of distinct values is going to have a critical error built in. (It's a bit like the problem that young children have with fence posts and fence panels—why are there 11 posts, but 10 panels? Putting it the grown-up way—how come there are 11 integers between 30 and 40 when 40 – 30 = 10?)

But how should we apply the algorithm in each of the slightly different cases?

Case 1

`month_no > 8`: This is an **unbounded** (no limit at one end), **open** (8 is excluded) range.

- Selectivity = (`high_value` – `limit`) / (`high_value` – `low_value`) = (12 – 8) / (12 – 1) = 4/11
- Cardinality = 1,200 * 4 / 11 = 436.363636 ... hence 437 or 436 depending on whether your version of Oracle rounds, or rounds up.

Case 2

`month_no >= 8`: This is an unbounded, **closed** (includes 8) range, so adjust for closure. The adjustment is made by *including* the rows for the closing value—in other words, adding `1/num_distinct`. (Again, 8*i* seems to use the `density` rather than `1/num_distinct`, but this isn't something you would notice unless you've been hacking the statistics or have a histogram in place.)

- Selectivity = (`high_value` – `limit`) / (`high_value` – `low_value`) + `1/num_distinct` = 4/11 + 1/12
- Cardinality = 1,200 * (4/11 + 1/12) = 536.363636 ... hence 537 or 536 depending on whether your version of Oracle rounds, or rounds up.

Cases 3 and 4

Analogous to cases 1 and 2:

- Selectivity (3) = (`limit` – `low_value`) / (`high_value` – `low_value`) = (8 – 1) / (12 – 1) = 7/11
- Selectivity (4) = (8 – 1) / (12 – 1) + 1/12

Cases 5 and 6

`month_no between 6 and 9`: Both are **bounded (limited at both ends)**, closed ranges—the `between` clause of case 5 is just a convenient shorthand for the two separate predicates of case 6. This gives us two closing values, so two occurrences of the adjustment.

Selectivity = (9 – 6) / (12 – 1) + 1/12 + 1/12 (>= , <=)

Cases 7, 8, 9

As for case 5, but with one, one, and zero adjustments for the closing value.

Selectivity (7) = (9 – 6) / (12 – 1) + 1/12 (>= , <)
Selectivity (8) = (9 – 6) / (12 – 1) + 1/12 (> , <=)
Selectivity (9) = (9 – 6) / (12 – 1) (> , <)

Although I've written out the exact formulae for each case, when I'm working on a problem I rarely do more than use the quick approximation of `requested range / total range`. However, you will appreciate that when there are only a few distinct values for a column, this estimate could be a long way out—it is a flaw that can have serious repercussions.

Looking at the significance of the *total range*, you will also appreciate that you could get some unexpected results if you have an application that uses silly values to avoid nulls. Consider an application that holds five years' worth of data (say 1 Jan 2000 to 31 Dec 2004). What will the optimizer's calculation for selectivity give for the predicate

```
where data_date between '01-Jan-2003' and '31-dec-2003'
```

Ignoring the adjustments for leap years and boundary details, the answer is going to be roughly 1/5. If you allow proper nulls in the `data_date` column, the result is unchanged. But if your application vendor decides that nulls should not be allowed, and decides to use 31-Dec-9999 instead, what happens? As soon as a single pseudo-null value goes into the table, the optimizer calculates the selectivity as 1/8,000. (You want one year out of a range of 8,000.) What do you think is going to happen to your execution plan when the optimizer's estimate is out by a factor of 1,600? We will return to this, and similar problems, in Chapter 6.

BIND VARIABLE PEEKING

Bind variables are good for OLTP systems, because they maximize sharable SQL and minimize CPU usage and latch contention during optimization (see Tom Kyte's *Expert Oracle Database Architecture*, Apress, September 2005). But we have just seen that bind variables make a complete nonsense of cardinality calculations. So 9*i* introduced bind variable peeking to address this problem.

The first time a piece of SQL is optimized, the optimizer (usually) checks the actual values of any incoming bind variables, and uses those values to do the optimizer calculations, which means the optimizer has a chance of picking the best plan for that first execution.

But on every subsequent occasion that a parse call is issued against that statement, and the text is found to be sharable, the same execution plan will generally be used regardless of any change in the value of the bind variables. (There is an exception relating to large variations in the lengths the character bind variables.)

In an OLTP system, this is likely to be a good thing, as OLTP activity tends to result in the same high-precision statements being repeated thousands of times per day, doing the same amount of work each time. But there are cases, even in an OLTP system, when this can cause problems.

In a **DSS** system, you usually need to avoid bind variables at all times so that people don't accidentally share executions plans for statements that look identical but perform enormously different amounts of work. And in a mixed environment, you want to make sure that you have a method for avoiding this bind variable trap for any heavy-duty SQL that you may need to run.

Just to confuse the issue: if you pass a statement with bind variables through `explain plan`, the optimizer doesn't know about any values (or even the **bind types**), so at best it will use the fixed selectivities to produce an execution plan. At worst, it will do an incorrect **implicit conversion** because it is treating something as a character that might be recognized as, say, numeric at run time, and give you a completely misleading execution plan.

10*g* Update

As with equalities, 10.1.0.4 suddenly changes what happens when you fall outside the low/high limits. The arithmetic, or picture, used for ranges outside the limits is just the same as the new mechanism we saw for equalities. So you get the sort of effects shown in Table 3-5 (see script `ranges_10g.sql` in the online code suite).

Table 3-5. *Out-of-Bounds Behavior Changes Within Versions of 10g (Reprise)*

Predicate	Cardinality (10.1.0.2)	Cardinality (10.1.0.4)
`month_no between 6 and 9`	527	527
`month_no between 14 and 17`	100	82
`month_no between 18 and 21`	100	45
`month_no between 24 and 27`	100	1

As you can see, the result you get inside the limits doesn't change, but as you move outside the range of possible values, the cardinality tapers off instead of using a constant dictated by the number of distinct values.

Two Predicates

Before you read on, look back at Table 3-4. The predicate `month_no > 8` gives a cardinality of 437, and the predicate `month_no <= 8` gives a cardinality of 864. Since every row in the table must meet exactly one or other of these two predicates (given that we have no nulls), what cardinality do you think the optimizer will return for the following clause?

```
where
        month_no > 8
or      month_no <= 8
```

There are three possible guesses:

- It will return 1301, because that's what you get by adding the cardinalities of the two individual predicates that obviously don't overlap.
- It will return 1200, because the combined predicate clearly describes all the rows in the table, and there are 1,200 rows in the table.
- It will return 986, because that was the last test I did in the script `ranges.sql`, and that's what actually happened.

The third argument is always compelling, even when the result is apparently irrational. And just to make things a touch more confusing, Table 3-6 shows the cardinalities you get on this query as you change the constant (see script `ranges_02.sql` in the online code suite).

Table 3-6. *Cardinality Varying When You Know It Shouldn't*

Constant	Cardinality
1	1,108
2	1,110
3	1,040

Table 3-6. *Cardinality Varying When You Know It Shouldn't (Continued)*

Constant	Cardinality
4	989
5	959
6	948
7	957
8	986
9	1,035
10	1,103
11	1,192
12	1,200

Isn't there a wonderfully tantalizing hint of symmetry centered on the 6 line; and isn't it aesthetically irritating that it's not quite in the middle of the table? We'll work out where these numbers came from in a couple of pages' time; at the moment I just want to remind you that the optimizer is not intelligent—it is just a piece of software.

To the human mind, it is obvious that `month_no > 8 or month_no <= 8` must return all the (non-null) entries in the table, but the optimizer code does not include that specific bit of pattern recognition; all it can see is two predicates with an **OR** between them.

The fact that the optimizer can recognize the slightly different `month_no >= 6 and month_no <= 9` as a single range on the same column is the result of a deliberate coding choice that stops it from being interpreted as two predicate clauses with an **AND** between them. It's a special case, and that's why the optimizer can calculate the (nearly) correct result.

To calculate general combinations of predicates, you need three basic formulae; and for these formulae, you do need to talk in terms of selectivities, rather than cardinalities.

- The selectivity of (`predicate1` **AND** `predicate2`) = selectivity of (`predicate1`) * selectivity of (`predicate2`).
- The selectivity of (`predicate1` **OR** `predicate2`) = selectivity of (`predicate1`) + selectivity of (`predicate2`) minus selectivity of (`predicate1` **AND** `predicate2`) ... otherwise, you've counted the overlap twice.
- The selectivity of (**NOT** `predicate1`) = 1 – selectivity of (`predicate1`) ... except for bind variable problems.

Let's go back to our audience of 1,200 as a concrete example of what these statements mean. Imagine the audience has traveled from all over the European Union (as of 1 September 2003) to hear me speak. That means they come from 15 different countries—and we will assume that the nationalities are randomly distributed (see script `two_predicate_01.sql` in the online code suite).

Instead of asking for everyone born in December to raise their hands as I did in the first example, I ask only the Italians born in December to raise their hands. How many should I see? I expect 100 people to be born in December (1 in 12); but that group of 100 comes from

15 countries, so only 1 in 15 of that group is likely to come from Italy. Since 100 / 15 = 6.5, I expect 6 or 7 people to raise their hands.

In SQL terms, I have `predicate1` as `month_no = 12` (selectivity = 1/12) and `predicate2` as `eu_country = 'Italy'` (selectivity = 1/15). The formula says that the combined selectivity is (1 / 12) * (1 / 15). The formula is simply stating algebraically the arithmetic we've just worked out intuitively for our audience.

What about the `or` case? Let's ask for people born in December or born in Italy to raise their hands. Of these, 1 in 12 were born in December (100), and 1 in 15 were born in Italy (80), so a first approximation says that 180 hands go up.

But if we stop there, we are going to count some people twice—the people who were born in December in Italy—so we have to subtract them from our total. That's easy because we've just worked out that the fraction of the audience born in December in Italy is (1 / 12) * (1 / 15). So the fraction of the audience we want is (1 / 12) + (1 / 15) – (1 / 12) * (1 / 15): just as the formula says.

Finally, how about the people who were *not* born in December? Since 1/12 of the audience was born in December, it's obvious that 11/12 of the audience was not born in December. And again, our formula is simply stating the normal mental arithmetic: fraction not born in December = 1 – fraction born in December.

Those of you who are familiar with probability theory will have recognized the three formulae as the standard formulae for calculating the combined probabilities of **independent events**:

- Probability(A **AND** B occur) = Probability(A occurs) * Probability(B occurs)
- Probability(A **OR** B occurs) = Probability(A occurs) + Probability(B occurs) – Probability (A **AND** B occur)
- Probability(**NOT**(A occurs)) = 1 – Probability(A occurs)

This equivalence is not really surprising. The probability of an event occurring is (loosely) the fraction of times it has occurred in previous tests; the selectivity of a predicate is (loosely) the fraction of rows in the table that match the predicate.

STRANGE BIND SELECTIVITIES

You may have wondered why the selectivity of `month_no between :b1 and :b2` is fixed at 0.0025. It's because the optimizer treats it as two predicates with an **AND**.

The selectivity invented for `month_no > :b1` is fixed at 0.05, the selectivity invented for `month_no < :b2` is the same; so the selectivity of both predicates being true is: 0.05 * 0.05 = 0.0025 (see script `bind_between.sql` in the online code suite).

The selectivity of `not (column > :b1)` is a special case as well: it is 0.05, not 0.95. The optimizer respects the *intention* that any test of an unbounded range scan on a column should return 5% of the data.

This problem gets even stranger when you use `not(month_no between :b1 and :b2)`. The value that the optimizer uses is 9.75% because the predicate is equivalent to `(month_no < :b1 or month_no > :b2)`—which should be 5% plus 5%, except the optimizer subtracts 0.25%, emulating the "subtract the overlap" strategy of a more general **OR** clause.

Problems with Multiple Predicates

I'd like to finish this chapter by doing two things with the general formulae for combining predicates. First I'd like to walk through the arithmetic that lets the optimizer calculate a cardinality of 986 on a clause that clearly covers exactly 100% of the rows in our 1,200 row table. Then I'd like to show you why the formulae that the optimizer uses are guaranteed to produce the wrong figures in other, more realistic, cases.

Let's dissect the `where` clause:

```
where
        month_no >  8               -- (predicate 1)
or      month_no <= 8               -- (predicate 2)
```

- From our single selectivity formula, (`required range / total range`), we calculate the selectivity for `predicate1`, `month_no > 8`, as (12 – 8) / (12 – 1) = 4 / 11 = 0.363636.
- Similarly, the selectivity for `predicate2`, `month_no <= 8`, is (8 – 1) / (12 – 1) + 1/12 = 7/11 + 1/12 = 0.719696.
- Our formula for selectivity(P1 **OR** P2) is selectivity(P1) + selectivity(P2) – selectivity(P1 **AND** P2), so the combined selectivity is 0.363636 + 0.719696 – (0.363636 * 0.719696) = 0.8216.
- Multiply this selectivity by the 1,200 rows in the table, round up, and you get 986, exactly as we saw in Table 3-6. It's clearly the wrong answer for reasons that are intuitively obvious to the human mind. But the machine is not human—it follows the code, it doesn't understand the situation.

In fact, this error is just a special case of a more general problem, which I will introduce by going back one last time to my audience of 1,200 people.

Assume everyone in the audience knows which star sign they were born under (I know that astrology is not a topic that should appear in a book that's trying to be scientific, but it does provide a convenient demonstration).

If I ask all the people born under Aries to raise their hands, I expect to see 100 hands—there are 12 star signs, and we assume uniform distribution of data—selectivity is 1/12, cardinality is 1,200/12 = 100.

If I ask all the people born in December to raise their hands, I expect to see 100 hands—there are 12 months, and we assume uniform distribution of data—selectivity is 1/12, cardinality is 1,200/12 = 100.

How many people will raise their hands if I ask for all the people born under Aries and in December to raise their hands? What about all the people born under Aries in March? What about all the people born under Aries in April?

According to Oracle, the answer will be the same for all three questions:

- Selectivity (month **AND** star sign) = selectivity (month) * selectivity (star sign) = 1/12 * 1/12 = 1/144
- Cardinality = 1,200 * 1/144 = 8.5 (rounded to 8 or 9 depending on version of Oracle)

But the star sign Aries extends from 21 March to 19 April—so there can't be any people born in December under Aries; about 35 of the 100 people born in March will be under Aries, and about 65 of the 100 people born in April will be under Aries. Star signs and calendar dates

are *not independent*, but Oracle's formulae for calculating combinations of predicates assume that the predicates are independent.

As soon as you apply multiple predicates to a single table, you need to ask whether there is some dependency between the columns you are testing. If there is a dependency, the optimizer will produce incorrect selectivities, which means incorrect cardinalities, which could easily mean inappropriate execution plans.

IN-LISTS REVISITED

Remember how 8*i* underestimated the cardinality of an in-list. We expected a cardinality of 300 from `month_no in (6,7,8)` but got a cardinality of 276. When you expand the formula for the selectivity of two predicates **OR**'ed together to cover three predicates, it looks like this:

sel(A or B or C) =

sel(A) + sel(B) + sel(C) – Sel(A)sel(B) – Sel(B)sel(C) – sel(C)sel(A) + Sel(A)Sel(B)Sel(C)

The individual selectivity for each month is 1/12, but put 1/12 into the preceding formula, and the answer is: 3/12 – 3/144 + 1/1728 = 0.22975. Multiply this by the 1,200 rows we started with and the answer is 275.69—the cardinality given by 8*i*.

Oracle's error in 8*i* was that it used the general method for handling **OR**'ed predicates when it expanded in-lists; it didn't recognize the special case. It's a common feature in the optimizer—many of the enhancements that appear in newer versions are the result of special cases being recognized, and being treated more appropriately. Enhancements are a good thing—but anything that changes the cardinality of an operation (even to correct it) may cause an execution plan to change unexpectedly.

Notice particularly that the same query with different inputs could mean that the standard cardinality estimate is too high (Aries and December) or too low (Aries and April). This means the problem of dependencies, particularly between columns in a single table, is not one that can be solved automatically. 9*i* offers **dynamic sampling** as a partial solution, and 10*g* offers **profiles**—we will take a further look at both options in Chapter 6.

But since the error in cardinality could go either way on exactly the same input text, the only complete solution requires Oracle to optimize for each set of input values as they appear by sampling the data. This may be feasible in a data warehouse environment, but is not viable in a high-performance OLTP system because of the extra resource consumption and contention issues that would inevitably appear.

Summary

To estimate the number of rows returned by a set of predicates, the optimizer first calculates the selectivity (fraction of data to return) and then multiplies it by the number of input rows.

For a predicate on a single column, the optimizer uses either the number of distinct values or the density as the basis for calculating the selectivity of an equality predicate. For range-based predicates on a single column the optimizer bases the selectivity on the fraction `range required / total available range` with some adjustments for end-point values.

For range predicates involving bind variables, the optimizer uses hard-coded constants for the selectivity—5% (0.05) for unbounded ranges, 0.25% (0.0025) or 9.75% (0.975) for bounded ranges.

The optimizer combines predicates by using the formulae for calculating the probability of independent events. This can lead to errors in selectivity (hence cardinality) when the predicates involve columns containing data sets that are not independent of each other.

Queries involving in-lists display some idiosyncratic behavior. The treatment of "in" and "not in" are self-consistent (and wrong) in 8*i*. The treatment of "in" is corrected in 9*i* and 10*g*, but will give misleading results for a list with bind variables, or values outside the expected range of values for the column. The treatment of "not in" is still incorrect in 9*i* and 10*g*, using the same calculation that 8*i* uses.

Test Cases

The files in the download for this chapter are shown in Table 3-7.

Table 3-7. *Chapter 3 Test Cases*

Script	Comments
`birth_month_01.sql`	Coded equivalent of baseline example
`hack_stats.sql`	Framework script to allow you to modify some object-level statistics
`birth_month_02.sql`	Adds null values to the baseline example
`in_list.sql`	Queries with different-sized in-lists against the baseline example
`in_list_02.sql`	Queries with different-sized in-lists against the modified example
`oddities.sql`	Collection of in-list queries that produced some undesirable results
`in_list_03.sql`	Queries with `in` and `not in` against the baseline example
`pv.sql`	Partition views and in-lists (versions vary significantly)
`in_list_10g.sql`	Demonstration of changes in 10.1.0.4 specifically
`ranges.sql`	Queries with different types of ranges against the baseline example
`selectivity_one.sql`	Demonstration of a boundary case for tests outside the low/high range
`like_test.sql`	Example of predicate `character like :bind`
`ranges_10g.sql`	Demonstration of changes in 10.1.0.4 specifically
`ranges_02.sql`	Oddity with `column < X or column >= X`
`two_predicate_01.sql`	Test case for two independent predicates
`bind_between.sql`	Anomalies when handling range predicates with bind variables
`setenv.sql`	Sets a standardized environment for SQL*Plus

CHAPTER 4

Simple B-tree Access

In this chapter, we will examine the arithmetic the optimizer uses to calculate the cost of using a simple B-tree index to access a single table. The investigation will not be exhaustive, and will focus only on the general principles, skimming over the numerous special cases where the optimizer "tweaks" the numbers a little.

I shall basically be covering range scans, including full scans and index-only scans, with a brief mention of unique scans. There are other uses of indexes, of course—we have already seen fast full scans in Chapter 2, and we will be exploring skip scans, and index joins in volumes 2 and 3.

This chapter requires you to be familiar with the concept of selectivity and how the optimizer calculates it, so you may want to read Chapter 3 before you start in on this one.

Basics of Index Costing

As you have seen in the earlier chapters, the cost of a query is a measure of time to completion, and one of the most significant factors in the time required is the number of real, physical I/O requests that have to be satisfied. This shows up very clearly in the basic formula for the single-table indexed access path.

Before quoting the formula, though, it is worth creating a mental image of what you would do to walk through a typical index access path.

- You have predicates on some of the columns that define the index.
- You locate the **root block** of the index.
- You descend through the **branch levels** of the index to the **leaf block** that is the only place for the first possible entry (the **start key**) that could match your predicates.
- You walk along a chain of leaf blocks until you overshoot the entry that is the last possible entry (the **stop key**) that could match your predicates.
- For each index entry, you decide whether or not to visit a table block.

So the formula for the cost of accessing a table by way of an index ought to cover three block-related components: the number of branch levels you descend through, the number of leaf blocks you traverse, and the number of table block visits you make. And as we know, the optimizer assumes that a single block visit equates to a real I/O request—so the number of block visits is the cost (if you have not enabled CPU costing).

SKIP SCANS

Because index leaf blocks contain forward and backward pointers, the standard *range scan* mechanism for using an index only has to descend through the branch levels once to find a starting block (whether the range scan is ascending or descending).

The mechanics of a skip scan (which will appear in a later volume) require Oracle to go up and down the branch levels. Not only is the arithmetic different, but the strategy for pinning buffers changes too. Every block in the path from the root block to the current leaf seems to be pinned as the leaf is scanned—after all, you don't want someone else to split your buffered copy of a branch block if you're going to go back up to it almost immediately.

The thing I call the *baseline formula* for index costing was first made public in a paper by Wolfgang Breitling (`www.centrexcc.com`) at IOUG-A in 2002. This formula is made up of exactly the three components I've just described:

```
cost =
        blevel +
        ceiling(leaf_blocks * effective index selectivity) +
        ceiling(clustering_factor * effective table selectivity)
```

- The first line of the formula represents the number of block visits needed to descend through the index (excluding the cost of actually hitting the first leaf block you want). Oracle implements a version of **Balanced B-trees**, so every leaf block is the same distance from the root block, and we can talk about *the* height of the index quite safely. The number of layers of branch blocks you have to descend through is the same whichever leaf block you want to get to.
- The second line of the formula represents the number of leaf blocks that you will have to walk along to acquire all the **rowids** matching a given set of input values. The *effective index selectivity* corresponds to the entry labelled `ix_sel` in the `10053 trace` file.
- The third line represents the number of visits to table blocks that you will have to make to pick up the rows by way of the selected index. The *effective table selectivity* corresponds to the thing that used to be labelled `tb_sel` in the `10053 trace` file, but ends up being labelled (more accurately) as `ix_sel_with_filters` in the 10*g* trace file. This line often generates the biggest component of the cost, and introduces the biggest error in the calculation of the cost of using a B-tree index. We will examine various causes and corrections for the errors in Chapter 5.

INDEX HEIGHT

There are still several documents and presentations doing the rounds that talk about sequence-based indexes *spawning extra levels* on the right-hand side. This does not happen. When leaf blocks split, the split travels upwards, if necessary, not downwards. All the leaf blocks in a Balanced B-tree will be at the same distance from the root.

I recently came across a paper that described a mechanism used by RDB for adding *overflow nodes* for repeated nonunique keys. If this description is actually correct, it's possible that the misunderstanding about Oracle's implementation arrived as a side effect of individuals migrating from one platform to another and assuming that their previous knowledge was still relevant.

The `blevel`, `leaf_blocks`, and `clustering_factor` are available in the view `user_indexes` once you've collected statistics. The *effective index selectivity* and *effective table selectivity* are calculated at optimization time based on the incoming predicates. We will investigate how the two selectivities are calculated in a worked example.

So the optimizer estimates the number of block visits, assumes that each block visit will turn into an I/O request—and that's basically the cost. There are numerous special cases, lots of side effects from tweaking parameters and hacking statistics, and some anomalies due to bugs and enhancements in different versions of the code. And with 9*i* and its CPU costing, we also have to add in a CPU component for each block visited and yet another CPU component for examining the data.

Getting Started

Let's build an example so that we can see the different bits of this formula in action. As usual, my demonstration environment starts with an 8KB block size, 1MB extents, locally managed tablespaces, manual segment space management, and system statistics (CPU costing) disabled. This sample is taken from script `btree_cost_01.sql` in the online code suite:

```
create table t1
as
select
        trunc(dbms_random.value(0,25))        n1,
        rpad('x',40)                          ind_pad,
        trunc(dbms_random.value(0,20))        n2,
        lpad(rownum,10,'0')                   small_vc,
        rpad('x',200)                         padding
```

```
from
        all_objects
where
        rownum  <= 10000
;

create index t1_i1 on t1(n1, ind_pad, n2)
pctfree 91
;
```

A couple of oddities about this data may need a little explanation. First, the `pctfree` setting on the index is unusually large; I've done this to force the index to spread itself across a large number of leaf blocks when it is first created. But `pctfree` does not apply to branch blocks—which is why I have introduced `ind_pad` as the second column of the index. Because this holds the same value for all rows, it doesn't affect the overall statistics and distribution, but it does stop Oracle from being able to pack lots of rows into each branch block, which conveniently pushes the index up to a `blevel` of 2.

PCTFREE FOR INDEXES

When applied to indexes, the `pctfree` storage parameter has a somewhat different meaning than it has for tables. For indexes, `pctfree` is only relevant as an index is created, rebuilt, or coalesced; and it only applies to leaf blocks.

For a *table,* the `pctfree` storage parameter tells Oracle when to stop inserting new rows into a block, so that some space in each block can be left for updates to existing rows in that block. But entries in indexes are never updated—when you change an index entry it (usually) belongs somewhere else in the index, thus an *update* to an index is really a delete followed by an insert—so you don't reserve space for updates, you reserve space for new rows.

Note that `n1` is specified in a way that will produce 25 different values (0 to 24), and `n2` is specified to produce 20 different values (0 to 19). Because of the random distribution, we are likely to see all 500 different possible combinations of numbers, with about 20 rows per combination (10,000 rows divided by 500 combinations).

After using the `dbms_stats` package to compute statistics on the table and index, you should get the results shown in Table 4-1.

Table 4-1. *Statistics for the Sample Table and Its Index*

Statistic	Value
Table rows	10,000
Table blocks (below the high water mark)	371
Index rows (entries)	10,000
Index leaf blocks	1,111

Table 4-1. *Statistics for the Sample Table and Its Index*

Statistic	Value
Index `blevel` (number of branch levels)	2
Index distinct keys	500
Index clustering factor	9,745
Index average leaf blocks per key	2
Index average data (table) blocks per key	19
Column `n1` distinct value	25
Column `n1` number of nulls	0
Column `n1` density	0.04
Column `n2` distinct values	20
Column `n2` number of nulls	0
Column `n2` density	0.05
Column `IND_PAD` distinct values	1
Column `IND_PAD` number of nulls	0
Column `IND_PAD` density	1.00

Now switch on `autotrace` and run the following query. `Autotrace` under 9*i* or 10*g* should report the execution plan that follows this query.

```
select
        small_vc
from
        t1
where
        n1      = 2
and     ind_pad = rpad('x',40)
and     n2      = 3
;

Execution Plan (Autotrace 9.2.0.6 & 10.1.0.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=25 Card=20 Bytes=1160)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=25 Card=20 Bytes=1160)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=5 Card=20)
```

In this query, the optimizer chooses an index and reports a cost of 25, with a cardinality of 20. We know that the cardinality happens to be a good estimate, but where does the cost come from? The first part of the cost comes from line 2 (`Cost = 5`) for visiting the index, and the next part (`Cost = 25`, an increment of 20) comes from line 1 for visiting the table.

Informally, we shouldn't be surprised that the cost of visiting the table was 20. We were expecting about 20 rows and knew that those rows were scattered all around the table. So we can accept that the incremental cost of 20 in the table line relates to the 20 different blocks that we will have to visit to acquire those 20 different rows.

Similarly, when we see the cost of using the index was 5, we might allocate two units to walking down the branch levels (the `blevel` of the index was 2). But that leaves three more units to account for—perhaps the optimizer has estimated that we will walk three leaf blocks to get the 20 rowids we need.

An informal argument is fine as a starting point, but if we want to investigate the formula and understand the optimizer's use of indexes a little better, we now need to know the significance of the two selectivities and the `clustering_factor`.

Effective Index Selectivity

In Chapter 3, I described the selectivity as the fraction of the rows that the optimizer expected to find based on the predicates supplied. But now I have raised the stakes to two selectivities, the *effective index selectivity* and the *effective table selectivity*.

The principle is exactly the same, though. Forget about the table for a moment; what fraction of the row entries in the index will we return if we use the three predicates from the query? All three predicates are on columns in the index, so we investigate all three of them, starting with their density or `num_distinct`:

`n1 = {constant}`	(Target: 1 row in 25, or 4% of the rows, or 0.04 * number of rows)
`ind_pad = {constant}`	(Target: 100% of the rows)
`n2 = {constant}`	(Target: 1 row in 20, or 5% of the rows, or 0.05 * number of rows)

Apply the formula from Chapter 3 for combining selectivities:

```
selectivity (P and Q) = selectivity(P) * selectivity(Q)
```

But we have three events—so we have to extend the formula a little:

```
selectivity (X and Y and Z) =
selectivity((X and Y) and Z) =
selectivity(X and Y) * selectivity(Z) =
selectivity(X) * selectivity(Y) * selectivity(Z)
```

We simply multiply together all three selectivities. In this case, the *effective index selectivity* is 0.04 * 1 * 0.05 = 0.002 (remember this figure for later). Our predicates will require us to visit 0.2% of the entries in the index. The thing about indexes, though, is that their leaf blocks are packed in sorted order. If we work out that we are going to examine X% of the index rows, then we are going to walk along X% of the leaf blocks to do so. This is why one component of the cost is `leaf_blocks` * *effective index selectivity*.

At this point, you may want to stop me and point out an oddity. The view `user_indexes` has a column called `distinct_keys`—which is where I got the value of 500 for my list of statistics a couple of pages ago. The optimizer "knows" that this index has 500 distinct entries, and I've just put in a query for

```
(every index column) = {constant}
```

Why doesn't the optimizer simply say there are 500 possible sets of values in the index, you want one possible set of values, so the selectivity is 1/500 = 0.002?

Good question, and a perfectly sensible suggestion. But generally the optimizer really does multiply up the separate selectivities, rather than looking at the combined index selectivity. (There seems to be just one special case that we will examine in Chapter 11.) Don't forget, though, that Oracle records both a `density` and `num_distinct` for each column. Perhaps the strategy of multiplying the separate column selectivities is a generic solution that allows for skewed data when there are histograms in place and the number of distinct values in the index cannot be trusted.

Effective Table Selectivity

We already know how to calculate selectivities on tables; and it's especially easy for the case where all the table-related predicates are **AND**'ed together. We just multiply the individual selectivities together.

But there is a refinement you have to consider when you are evaluating indexes. Imagine my sample query includes the extra predicate `small_vc = '0000000001'`. If you choose to visit the table by way of the existing index, you cannot check this final predicate until after you have reached the table—so this predicate does not affect the fraction of the data that you are going to *visit*, only the fraction of data that you are finally going to *return*.

When working out the cost of using an index, the *effective table selectivity* should be based only on those predicates that can be evaluated in the index, before you reach the table. (This is why I have termed it *effective table selectivity* rather than simply *table selectivity* and why 10*g* release 2 has relabelled it as the `ix_sel_with_filters`.)

In this case, the predicates we have on the table can all be resolved in the index, so we can safely say that the *effective table selectivity* is (also) 0.04 * 1 * 0.05 = 0.002.

clustering_factor

The `clustering_factor` is a measure that compares the order of the index with the degree of disorder in the table. The optimizer appears to calculate the `clustering_factor` by walking the table in index order and keeping track of how many times the walk jumped from one table block to another. (Of course, it doesn't really work like this; the code simply scans the index extracting table block addresses from rowids.) On every jump, a counter is incremented—the final value of the counter is the `clustering_factor`. Figure 4-1 illustrates the principle.

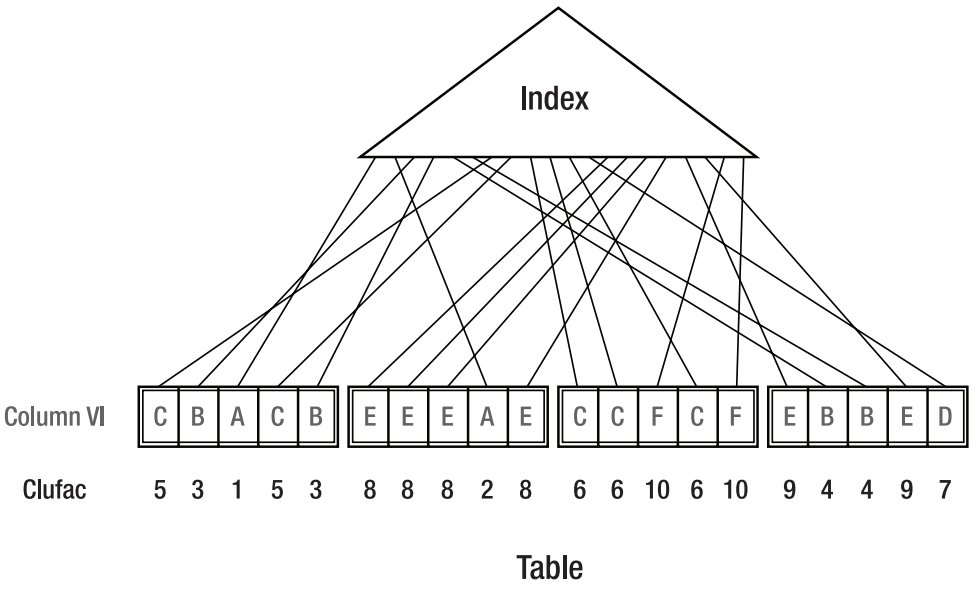

Figure 4-1. *Calculating the clustering_factor*

In Figure 4-1, we have a table with four blocks and 20 rows, and an index on the column `V1`, whose values are shown. If you start to walk across the bottom of the index, the first rowid points to the third row in the first block. We haven't visited any blocks yet, so this is a new block, so we count 1. Take one step along the index, and the rowid points to the fourth row of the second block—we've changed block, so increment the count. Take one step along the index, and the rowid points to the second row of the first block—we've changed block again, so increment the count again. Take one step along the index, and the rowid points to the fifth row of the first block—we haven't changed blocks, so don't increment the count.

In the diagram, I have put a number against each *row* of the table—this is to show the value of the counter as the walk gets to that row. By the time we get to the end of the index, we have changed table blocks ten times, so the `clustering_factor` is 10.

Notice how small clumps of data stop the `clustering_factor` from growing—look at block 2 where the value 8 appears four times because four consecutive entries in the index point to the same block; the same effect shows up in block 3 to give three rows the value 6.

The table doesn't have to be completely sorted for this type of thing to happen; it only needs to have little clumps (or clusters) of rows that are nearly sorted—hence the term *clustering_factor*, rather than *sort_factor*.

Given the way the `clustering_factor` is calculated, you will appreciate that the smallest possible value has to be the same as the number of blocks in the table, and the largest possible value has to be the same as the number of rows in the table—provided you have computed statistics.

If there are lots of blocks like block 2 in the table, the `clustering_factor` will turn out to be quite close to the number of blocks in the table, but if the data in the table is randomly scattered, the `clustering_factor` will tend to come out close to the number of rows in the table.

INDEXES AND THE CLUSTERING FACTOR

Historically it has been common practice to say that a *good* index has a *low* `clustering_factor`, and a *bad* index has a *high* `clustering_factor`.

There is obviously a degree of truth in this comment, especially in the light of what the `clustering_factor` represents. However, I have always had an aversion to words like *low*, *high*, *small*, *large*, and expressions like *close to zero*, when talking about Oracle. After all, is 10,000 a low `clustering_factor` or a high `clustering_factor`? It's low if you have 10,000 blocks in your table, and high if you have 100 blocks in your table. So you might want to write a couple of little scripts that join `user_tables` to `user_indexes` (and other scripts for partitioned tables, etc.) so that you can compare the critical figures.

In fact, for reasons I describe in Chapter 5, I often use the column `avg_data_blocks_per_key` to get an idea of how good Oracle thinks the index is.

So why does the `clustering_factor` feature in the formula for costing? When you acquire more than one row from a table through an index (in other words, when you are going to do an index range scan), you walk a section of the index—call it X% of the index. As you walk the index, you will be hopping around the table from row to row. If the `clustering_factor` is truly representative of the way that the data is scattered around the table, then as you walk X% of the index, the number of times you change table block will be X% of `clustering_factor`.

The optimizer behaves as if every change of block takes you to a block that you have not previously visited, which will therefore require an I/O request. This may be a fairly reasonable assumption in some cases, and explains why `clustering_factor` * *effective table selectivity* (rounded up) appears as the final component of the cost.

There are flaws in the argument, of course, and we shall examine them in more detail in Chapter 5. Consider, for example, the first table block in Figure 4-1. If we execute a query that uses the index to visit all five rows in that block, the optimizer will have allowed for three physical I/O requests to that specific block in its cost calculation (there are three *visit numbers* in the block). But the block will probably be buffered (and possibly pinned) for all visits after the first one. It is quite common for this part of the formula to produce a value that does not reflect reality. In fact, thanks to a *table pre-fetch* mechanism that appeared in 9*i* (see Chapter 11), there is likely to be an ever-increasing difference between the calculated costs and actual block visits in increasing numbers of cases.

Putting It Together

We have the formula for the cost of an index-driven access path as

```
cost =
        blevel +
        ceiling(leaf_blocks * effective index selectivity) +
        ceiling(clustering_factor * effective table selectivity)
```

We know what each of the terms represents, so we can check our simple example to see whether the formula gives us the right answer—in other words, whether it provides the numbers we see in the `autotrace` output. Let's just slot in the numbers from our first test:

`blevel` = 2
Effective index selectivity = 0.002 (As calculated previously)
`leaf_blocks` = 1,111
Effective table selectivity = 0.002 (As calculated previously)
`clustering_factor` = 9,745
Table rows = 10,000

So, according to the formula, the cost should be

```
2 + ceiling(1,111 * 0.002) + ceiling(9,745 * 0.002) =
2 + ceiling(2.222) + ceiling(19.49) =
2 + 3 + 20 =
5 + 20 =
25
```

Compare this with the execution plan, where the cost on the index line was 5 (which we hypothesized was 2 for descending the branches and 3 for walking the leaf blocks) and the total cost was 25. The correspondence is perfect.

So the formula seems to be sound for this simple case; the results of the arithmetic match the figures produced by the execution plan listing. So what does it tell us specifically about the optimizer's perceived cost of using B-tree indexes?

Practical experience tells us that the `blevel` is typically 3 or less, often 2, occasionally 4; that indexes tend to be densely packed; that index entries tend to be smaller than table rows, so leaf blocks usually hold far more entries than the corresponding table blocks; and that the number of leaf blocks in an index is often small compared to the number of blocks in the table. So, for any set of predicates targeting more than three or four rows in the table, the most significant component in the cost calculation is quite likely to depend on the `clustering_factor`—i.e., how randomly scattered the target data appears to be.

If the optimizer thinks the data in your table is well clustered, then the cost will be low, favoring the use of indexes. If the optimizer thinks the data in the table is very scattered, then the cost will be higher, favoring the use of alternative execution plans (such as tablescans). Counterintuitively, the state of your table can have a much bigger impression on the optimizer's calculations than the state of the index itself.

Of course, if your data really is distributed the way the optimizer thinks it is, then the cost calculation will often come close to a realistic run-time use of resources, and you will think the optimizer has done a good job. If the `clustering_factor` doesn't represent the real distribution, then the cost calculation will be wrong, and the optimizer is likely to choose an unsuitable execution plan.

REBUILDING INDEXES

You can often reduce the number of `leaf_blocks` (and, very occasionally, the `blevel`) of an index by rebuilding the index; but rebuilding an index has no effect on the `clustering_factor`.

Rebuilding an index may make that index appear more desirable to the optimizer, but the side effects may be good or bad. On the plus side, rebuilding an index may introduce a caching benefit for that index at query time—but the downside is a possible contention penalty on DML, combined with an increase in **leaf blocks splits** and **redo** generated as the index "tries" to get back to its equilibrium state. Counterintuitively, an index may end up "needing" regular rebuilds because you've started to rebuild it regularly (see script `rebuild_test.sql` in the online code suite for an artificial example demonstrating the concept).

If you want to get the optimizer to behave properly, it is usually more important to check whether or not the `clustering_factor` represents the real data scattering and do something about that, rather than simply wedging the index into a tighter hole.

Extending the Algorithm

Let's take a look at some more general examples of costing. For example:

- How do you handle range-based tests (e.g., `n2 between 1 and 3`)?
- How do you handle partial use of a multicolumn index?
- What effects do in-lists have?
- How special is an index full scan?
- What does the optimizer do when you don't have to visit the table?

How Do You Handle Range-based Tests (e.g., n2 between 1 and 3)?

Let's try it and see. Try running the following query (`btree_cost_02.sql` in the online code suite) against our base data set with `autotrace` enabled:

```
select
        /*+ index(t1) */
        small_vc
from
        t1
where
        n1      = 2
and     ind_pad = rpad('x',40)
and     n2      between 1 and 3
;
```

```
Execution Plan (9.2.0.6 and 10.1.0.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=93 Card=82 Bytes=4756)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=93 Card=82 Bytes=4756)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=12 Card=82)
```

You'll note that I've had to include an index **hint** with this statement, as the cost of using the index is rather higher than the cost of a tablescan. (The number of blocks below the table's high water mark was 371, and my setting for `db_file_multiblock_read_count` was 8, so in the absence of CPU costing, 9*i* calculated the tablescan cost as 1 + 371 / 6.588 = 58.)

As ever, we need to think about selectivity. The individual selectivities on columns `ind_pad` and `n1` are unchanged at 1 and 0.04 respectively, but we now have a range-based predicate on `n2`. Referring to Chapter 3, we can pull out the formula for the `between` predicate:

```
selectivity (n2 between 1 and 3) =
        required range / total range + 1/num_distinct + 1/num_distinct =
        (3 - 1) / (19 - 0)  + 1/20 + 1/20  = 0.205263
```

As with the previous example, every predicate applies to both the index and the table, so this number can be used in both places in the formula. Slotting the numbers in, then, we get the following:

```
        Effective index selectivity = 1 * 0.04 *  0.205263 = 0.0082105
        Effective table selectivity = 1 * 0.04 *  0.205263 = 0.0082105
        Cost =  2 +
                    ceiling(1,111 * 0.0082105) +                    -- 10
                    ceiling(9,745 * 0.00082105)                     -- 81
                =  12 (index line of plan) + 81 = 93                -- as required
```

There was one little glitch in this test that I haven't disclosed. I labelled the execution plan with "(9.2.0.6 and 10.1.0.4)". When I repeated the test under 8*i*, there was a small difference—the cardinality changed from 82 to 83.

The calculated cardinality is defined as

```
Input rows * calculated selectivity
10,000 * 0.00882105 = 82.105  -- should this be 82, or 83?
```

This is actually something we have seen before in Chapter 3. 8*i* always seems to round cardinality up (the `ceiling()` function, or as SQL puts it, the `ceil()` function), but 9*i* and 10*g* round to the nearest whole number (the `round()` function).

More on Range-based Tests

We took the easy option, and did a range-based test on the last column in the index. What happens if we do a range-based test on an earlier column in the index? Try this, for example:

```
alter session set "_optimizer_skip_scan_enabled"=false;

select
        /*+ index(t1) */
        small_vc
from
        t1
where
        n1      between 1 and 3
and     ind_pad = rpad('x',40)
and     n2      = 2
;

Execution Plan (8.1.7.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=264 Card=82 Bytes=4756)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=264 Card=82 Bytes=4756)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=184 Card=82)

Execution Plan (9.2.0.6 and 10.1.0.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=264 Card=82 Bytes=4756)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=264 Card=82 Bytes=4756)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=184 Card=1633)
```

The `alter session` command is there for the benefit of 9*i* and 10*g*. The unhinted execution plan was a full tablescan, but when I first put in an index hint, the optimizer insisted on using the **index skip scan** mechanism, and in 10*g* if I then included the `no_index_ss()` hint, the index was disabled and the plan went back to a tablescan (I would be inclined to call this behavior a bug—it seems perfectly reasonable to me to say, "Use this index, but don't do a skip scan," but it is possible that it's the specified behavior).

An important point to note is that the cardinality of this execution plan matches what we saw in the previous query (82), but the cost is very much greater (264 instead of 93). Let's try to use the formula in the standard way, and see what happens.

- The selectivity for column `n1` is (3 – 1) / (24 – 0) + 2/25 = 0.1633333.
- The selectivity of `ind_pad` is still 1.
- The selectivity of `n2` is still 0.05.

So we can produce a combined selectivity of 1 * 0.1633333 * 0.05 = 0.0081667 (and multiplying this by the number of rows in the table and rounding we get 82, as the plan says).

If we slot this value for the selectivity into both places in the formula, what results do we get?

```
Cost =  2 +                                     -- blevel
            ceiling(0.0081667 * 1,111) +        -- 10
            ceiling(0.0081667 * 9745)           -- 80
     =  92                                      -- what's gone wrong,
                                                -- the answer should be 264!
```

Looking back at the execution plan, we see that the *index line* reports a cost of 184, and the *table line* reports a cost of 264, a difference of 80, which is what we get from the clustering portion of the formula. This suggests that it is the bit involving the index leaf block factor that has gone wrong.

The cardinality on the index line of the 9*i*/10*g* plan gives us the clue—the optimizer is telling us that it is going to examine 1,633 entries from the index, a huge fraction of the index. Why is this? Because as soon as we have a range scan on a column used somewhere in the **middle** of an index definition or fail to supply a test on such a column, the predicates on later columns don't restrict the selection of index leaf blocks that we have to examine.

This is why we have two selectivities in our formula, the *effective table selectivity* (which combines all predicates available on the index's columns) and the *effective index selectivity* (which may have to use a subset of the predicates based on the index's leading columns).

In this case, the *effective index selectivity* has to be calculated from the predicate on just the `n1` column. Because the test on `n1` is range-based, the predicates on `index_pad` and `n2` do not restrict the number of index leaf blocks we have to walk. Of course, when we finally get to examine an index leaf row, we can use all three predicates to check whether we should go to the table, so the *effective table selectivity* still includes all three individual column predicates.

So the *effective index selectivity* is 0.1633333 (the selectivity we calculated previously for column `n1`), and the final cost formula is

```
Cost =  2 +                                     -- blevel
             Ceiling(0.1633333 * 1,111) +       -- 182
             Ceiling(0.0081667 * 9745)          -- 80
     =   184 + 80 = 264                         -- as expected
```

This result confirms a well-known guideline for arranging the columns in a multicolumn index. Columns that usually appear with range-based tests should generally appear later in the index than columns that usually appear with equality tests. Unfortunately, changing the column ordering in an index can have other contrary effects, which we will examine in Chapter 5.

EXPLAIN PLAN ENHANCEMENTS

9*i* introduced two very important columns to the `plan_table` for supporting `explain plan`. These are the `filter_predicates` and `access_predicates`, which tell you exactly how, and where, the optimizer thinks it is going to use the components of your `where` clause.

If you are not making effective use of an index, the index line of an execution plan will highlight the problem very clearly. The `access_predicates` column will list the predicates being used to generate the start and stop keys for the index, but the `filter_predicates` column will list the predicates that cannot be used until the leaf blocks have been reached (in other words, the ones that should not be used in the calculation of the *effective index selectivity*).

Ranges Compared to In-Lists

In Chapter 3, we found an oddity with **in-lists** and tablescans under 8*i*. It is worth checking whether this oddity shows up when the expected access path is through an index. So let's run a simple test (`btree_cost_04.sql` in the online code suite) against our base data:

```
select
        /*+ index(t1) */
        small_vc
from
        t1
where
        n1      = 5
and     ind_pad = rpad('x',40)
and     n2      in (1,6,18)
;

Execution Plan (8.1.7.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=65 Card=58 Bytes=3364)
   1    0   INLIST ITERATOR
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=65 Card=58 Bytes=3364)
   3    2       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=9 Card=58)

Execution Plan (9.2.0.6 and 10.1.0.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=68 Card=60 Bytes=3480)
   1    0   INLIST ITERATOR
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=68 Card=60 Bytes=3480)
   3    2       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=9 Card=60)
```

Sure enough, the cost for the execution plan with the in-list varies between 8*i* and 9*i*. This shouldn't really be a surprise; the error in 8*i* relates directly to the selectivity of the in-list, and only indirectly to the cost. The error has been introduced before we even get to working out the cost. So if you use in-lists with indexes in 8*i*, the cost for the execution plan may go up, and the optimizer may switch to a different index or even a tablescan when you upgrade to 9*i* or 10*g*.

OPTIMIZER_INDEX_CACHING AND IN-LISTS

The `optimizer_index_caching` parameter was introduced in 8*i* to allow some room for correcting the optimizer's assumption that all reads are physical reads. It is usually mentioned as having an impact on the cost calculation for index block accesses for the inner (second) table of nested loop joins, but it also has an effect on the cost calculation for in-list iteration. Very specifically, it does not have an impact on the cost calculation for a simple, single-table indexed access path.

I don't have a complete understanding of how this works with in-lists, but as soon as the parameter is set to a non-zero value, the effective cost of the `blevel` component of the formula seems to be halved, after which the whole of the index component is adjusted by the cache percentage, with the usual oddities of

round() and ceil() confusing the issue. However, there also seems to be a lower limit relating to *number of leaf blocks* divided by *product of column selectivities*.

It would be nice to work out the whole algorithm, but it is probably sufficient in most cases to have this rough approximation.

How Do You Handle Partial Use of a Multicolumn Index?

After the example of the range scan at the start of the index, I don't think I need to go through all the workings for this one, because it's really just a special case of the previous problem. Take a table with an index (col1, col2, col3, col4), and consider this query:

```
select
        *
from
        t1
where
        col1 = {const}
and     col2 = {const}
-- and  no predicate on col3
and     col4 = {const}
```

We calculate the *effective index selectivity* from just col1 and col2, and we calculate the *effective table selectivity* from col1, col2, and col4. Then we apply the basic formula and the job is done.

What About Index Full Scans?

There are occasions when the optimizer will decide that the optimum access path is to read the entire index in correct index order, starting with the leaf block at one end of the index and following the leaf pointers until it gets to the leaf block at the other end of the index.

FIRST_ROWS OPTIMIZATION

One of the options for the optimizer_mode in 8*i* was first_rows. This option still exists for backward compatibility in 9*i* and 10*g*, but is deprecated. One of the critical features of first_rows optimization (compared to the newer first_rows_N optimization) was the existence of a few rules for overriding the normal costing behavior.

One such rule was that if there was an index that could be used to avoid a sort, then the optimizer would use it—apparently regardless of how much more expensive the path might be. So a query that acquired five rows and sorted them under all_rows might switch to acquiring 1,000,000 rows, and discarding all but five of them under first_rows if this meant the sort could be avoided.

So one side effect of first_rows optimization was the relatively frequent appearance of full scans of indexes. (In fact, you could change this behavior by adjusting the hidden parameter _sort_elimination_cost_ratio, as the default value made the behavior rather extreme.)

One possible cause for this behavior would be to avoid a sort for an order by clause, another might be that almost all the data in the table had been deleted and an indexed access to just the

few rows remaining would be quicker than scanning a very large number of empty table blocks. Consider this example (`btree_cost_03.sql` in the online code suite):

```
alter session set "_optimizer_skip_scan_enabled"=false;

select
        /*+ index(t1) */
        small_vc
from
        t1
where
        n2 = 2
order by
        n1
;

Execution Plan (8.1.7.4, 9.2.0.6 and 10.1.0.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1601 Card=500 Bytes=8500)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=1601 Card=500 Bytes=8500)
   2    1     INDEX (FULL SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=1113 Card=500)
```

Notice how there is no clue in this example that the `order by` clause is present in the query. More complex execution plans may be a little more helpful, with explicit lines like `sort (order by) nosort`.

How does the cost work in this case? Informally, we have no restrictions on the index until after we have reached the leaf blocks, so we could expect the *effective index selectivity* to be 1.00 (100%). When we examine the leaf blocks, we can identify the entries where `n2 = 2`, a single predicate with a selectivity of 0.05, so we could expect this to be the *effective table selectivity*. So let's put these numbers into the formula and check:

```
Cost =
    2 + (1 * 1111) + (0.05 * 9745) =
    2 + 1111 + 487.25 =                         -- round() or ceil() ?
    1113 + 488 =                                -- I've chosen ceil()
    1601
```

The numbers do work out properly—but only because I've fiddled them a little bit. Usually it's looked as if `ceil()` is used on costs in 8*i*, and `round()` in 9*i* and 10*g*. The error in this example is small, so I'm not going to worry too much about it. (For commonly occurring cases, I might fuss to nail down the error; for the fringe examples I'm happy with a 99% fit to the model, especially since I know that the kernel code has a number of "adjustments" for special cases.)

And Index-only Queries?

What should we do with the baseline formula when the query doesn't need to visit the table at all? Again, we can start with an informal argument, and check to see if the arithmetic works. If we aren't going to visit the table, then perhaps we just ignore the last component—the bit that represents the visit to the table (`btree_cost_03.sql` in the online code suite).

```
select
        /*+ index(t1) */
        n2
from
        t1
where
        n1 between 6 and 9
order by
        n1
;

Execution Plan (9.2.0.6 and 10.1.0.4)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=230 Card=2051 Bytes=12306)
1  0   INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=230 Card=2051 Bytes=12306)
```

As usual, we have to use a little care to calculate the selectivity of the range-based predicate on `n1`. In this case it is (9 – 6) / (24 – 0) + 2/25 = 3/24 + 2/25 = 0.205. So if we forget about the table component in the baseline formula, we get

```
Cost =
        2 + 0.205 * 1111 =
        2 + 227.755 =
        230
```

Our assumption seems to be correct.

The Three Selectivities

Just to finish off, let's add one final refinement to our test case, to show that the general case of a single table access by B-tree index has three selectivities. Run the following query (script `btree_cost_02.sql` in the online code suite again) through `autotrace`:

```
select
        /*+ index(t1) */
        small_vc
from
        t1
where
        n1       between 1 and 3
and     ind_pad  = rpad('x',40)
and     n2       = 2
and     small_vc = lpad(100,10)
;

Execution Plan (9.2.0.6 and 10.1.0.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=264 Card=1 Bytes=58)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=264 Card=1 Bytes=58)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=184 Card=1633)
```

This query starts like the example in the section "More on Range-based Tests," but adds one more predicate: `small_vc = lpad(100,10)`. Note particularly the cost has not changed, but the cardinality has dropped from 82 to 1 (`Card=1` in the first and second lines). The optimizer has calculated that the query will return just one row.

The mechanical steps at execution time are as follows:

- Check all the index leaf blocks for the range `n1 between 1 and 3` (*effective index selectivity*).
- Check all the table rows where the index entry passes all three index predicates (*effective table selectivity*).
- Return the rows where the fourth predicate passes.

So we have

- *Effective index selectivity* (`n1`) = 0.1633333
- *Effective table selectivity* (`n1`, `ind_pad`, `n2`) = 0.163333 * 1 * 0.05
- Final table selectivity (`n1`, `ind_pad`, `n2`, `small_vc`) = 0.163333 * 1 * 0.05 * 0.0001

And we get these cardinalities (which are always some variant of `selectivity * input row count`):

```
Index access line =
                round(0.163333 * user_tables.num_rows (10,000)) = 1,633

Rowid source (not part of plan) =
                round(0.0081667 * user_tables.num_rows (10,000)) = 82

Table return =
                round(0.0000081667 * user_tables.num_rows (10,000)) =  0!
```

At first glance, this seems to be another place where my theory about how 9*i* and 10*g* use the `round()` function to produce the final cardinality seems to break. In this case, though, it isn't really surprising, and you will find several other occasions where an answer that technically looks as if it should be zero is changed to one.

It's an interesting point, of course, that 9*i* and 10*g* report 1,633 as the cardinality on the index line of this plan. In most cases, the reported cardinality is the *output* cardinality, not the *input* cardinality—in other words, the number of rows that will be passed on by this line of the plan, rather than the number of rows that will be examined by this line of the plan. Ideally, of course, we would like to see both 1,633 and 82 reported against the index line in the plan, as both numbers are relevant—unfortunately, there is only space for one value in the execution plan.

Looking again at the section "More on Range-based Tests," 8*i* appeared to be following the *output count* rule when it reported 82 against the index line where 9*i* and 10*g* reported 1,633. In this example, 8*i* went one step further and reported a cardinality of just one against the index line. There will always be little inconsistencies like this that cause endless confusion, especially as you migrate through different versions of Oracle.

Problems with 10.1.0.4

If you examine script btree_cost_02.sql closely, you will see that it calls a copy of my hack_stats.sql script, which has been set to change the value of user_indexes.num_rows for the critical index. Originally I had done this to check whether Oracle used the value of num_rows stored in user_tables or user_indexes when working with the less common uses of indexes. In 8*i* and 9*i*, the reported cardinalities did not change when I made this change, but in 10.1.0.4, I got the following results from the preceding query when I changed the num_rows value in user_indexes to 11,000 and 9,900 respectively:

```
Execution Plan (10.1.0.4, autotrace - user_indexes.num_rows = 11,000)
----------------------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=264 Card=1 Bytes=58)
1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (TABLE) (Cost=264 Card=1 Bytes=58)
2    1     INDEX (RANGE SCAN) OF 'T1_I1' (INDEX) (Cost=184 Card=1797)

Execution Plan (10.1.0.4, autotrace - user_indexes.num_rows = 9,900)
----------------------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=93 Card=1 Bytes=58)
1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (TABLE) (Cost=93 Card=1 Bytes=58)
2    1     INDEX (RANGE SCAN) OF 'T1_I1' (INDEX) (Cost=12 Card=82)
```

In the case where I have boosted the value of num_rows from 10,000 to 11,000, the cardinality on the index line has gone up from 1,633 to 1,797—the optimizer seems to have used user_indexes.num_rows to calculate the cardinality of this line (1,797 = 1,633 * 11,000 / 10,000). Slightly surprising, but perfectly rational, and the cost of the query has not changed significantly, so quite a safe step.

But look what has happened when I artificially lowered the value of user_indexes.num_rows—the cost of the index line drops from its original 184 down to 12, and the cardinality drops from 1,633 to 82. The change in cardinality is no big deal, as the error is self-correcting by the time we reach the table—but the dramatic drop in cost echoes on through the plan. Basically, the arithmetic has simply dropped the cost of the leaf block scan (possibly through a simple numerical error: trunc(9,000 / 10,000) = 0).

Of course, it isn't completely safe to draw absolutely firm conclusions from hacked statistics, so script btree_cost_02a.sql in the online code suite repeats the final test of btree_cost_02.sql, but includes one extra line in setting up the data:

```
update t1
set
        n1      = null,
        ind_pad = null,
        n2      = null
where
        rownum <= 100
;
```

This line changes the data so that there are 100 rows in the table that do not appear in the index. This has virtually no effect on the costs and cardinalities of the plans produced in 8*i* and 9*i*, but again, when we get to 10*g*, we get a nasty surprise—the results we got from hacking the

statistics still appear when the data is genuine. This is the execution plan for the last query with the (slightly) modified data—I've included the equivalent plan from 9.2.0.6 for comparative purposes:

```
Execution Plan (10.1.0.4, autotrace - 100 completely null entries)
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=90 Card=1 Bytes=58)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (TABLE) (Cost=90 Card=1 Bytes=58)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (INDEX) (Cost=11 Card=80)

Execution Plan (9.2.0.6, autotrace - 100 completely null entries)
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=260 Card=1 Bytes=58)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=260 Card=1 Bytes=58)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=182 Card=1633)
```

Tracking this down into the `10053 trace` file, we find the following critical details (note that the entry labelled `ix_sel_with_filters` used to be called the `tb_sel` in earlier versions of Oracle):

```
With no null entries:
ix_sel:  1.6333e-001    ix_sel_with_filters:  8.1667e-003

With 100 null entries
ix_sel:  8.0850e-003    ix_sel_with_filters:  8.0850e-003
```

Clearly, when things go wrong, something has caused the `ix_sel_with_filters` to get copied into the `ix_sel`—which has the effect of losing the cost of the `leaf_block` accesses. Just to confuse the issue though, if you run script `btree_cost_02a.sql` several times in a row, you will find that sometimes it produces the wrong cost, and sometimes it produces the right cost, with the following line appearing in the `10053 trace`:

```
ix_sel:  1.6333e-001    ix_sel_with_filters:  8.0041e-003
```

The underlying problem is that the call to `dbms_stats.gather_table_stats()`, with the cascade option set to `true`, sometimes fails to update the index statistics. Counterintuitively, when `dbms_stats` gets it wrong, the execution plan comes up with the right cost (because the value of `user_indexes.num_rows` stays the same as `user_tables.num_rows`), and when `dbms_stats` gets it right, the execution plan comes up with the wrong costs because `user_indexes.num_rows` is (correctly) recorded as being less than `user_tables.num_rows`.

For the moment, I am assuming that this is a bug, so I haven't experimented further with the issue, as it may be fixed by the time this book is in print. But bear in mind that on an upgrade from 9*i* to 10*g*, this bug means that some very simple queries could change from tablescans to indexed accesses because the cost of the indexed access path has suddenly become unrealistically low—and it seems that this could happen more or less randomly.

CPU Costing

When you get to 9*i*, you have the option of enabling **system statistics**, and this makes a difference to execution plans. We examined how the new costing mechanisms work in Chapter 2,

but we can now consider the impact the feature can have even on very simple index-based execution plans.

As far as indexes are concerned, the arithmetic is basically unchanged. As we saw in the earlier chapter, the critical features of CPU costing are the time component for multiblock I/Os and the CPU utilization. But typical indexed access is done in single-block I/Os so the time component is irrelevant, and the CPU utilization walking an index to locate and acquire a few rows from a table is usually relatively small.

The big difference that appears when you enable CPU costing is that tablescans suddenly become more expensive. I'll demonstrate this with an example. To start, here's some code to create some system statistics (the code was specifically designed for 9*i*, but is valid on 10*g*, where there are a couple of other statistics):

```
alter session set "_optimizer_cost_model" = cpu;
begin
        dbms_stats.set_system_stats('CPUSPEED',350);       -- MHz
        dbms_stats.set_system_stats('MREADTIM',20);        -- millisec
        dbms_stats.set_system_stats('MBRC',8);
        dbms_stats.set_system_stats('SREADTIM',10);        -- millisec
end;
/
```

In this code fragment, we are telling the optimizer to assume that our CPU speed is 350MHz (or probably 350 million *Oracle operations* per second), that a multiblock I/O will take 20 milliseconds and will typically be fetching 8 blocks, whereas a single block I/O takes 10 milliseconds.

The deliberate setting of the optimizer cost model to use CPU costing is to standardize the effect between 9*i* and 10*g*. The default value for this parameter is `choose`, and 9*i* will choose to use CPU costing only if the system statistics exist, but 10*g* will always choose to use CPU costing and then synthesize some statistics if there aren't any in place (and I can't get the `delete_system_stats` call to drop the system statistics in 10*g*).

If we then run the following query against our base data set, we can see the effects of CPU costing (see script `btree_cost_05.sql` in the online code suite):

```
select
        small_vc
from
        t1
where
        n1      = 2
and     ind_pad = rpad('x',40)
and     n2      in (5,6,7)
;
```

In my test script, I ran the query with an `index()` hint, a `full()` hint, and unhinted. The costs behaved as shown in Table 4-2.

Table 4-2. *Effects of System Statistics*

Test Case	Original Cost	Cost with System Statistics 9*i* / (10*g*)
With `index()` hint	68	69 (68)
With `full()` hint	58	96 (95)
Path when not hinted	Full tablescan	Index iterator

In the absence of the system statistics, the optimizer estimated that the index-based path would visit 68 blocks individually for a total of 68 I/O requests, compared to 1 + (371 / 6.59) for the tablescan. (Remember that for tablescans and index fast full scans, and ignoring ASSM tablespaces, the optimizer divides the high water mark by an adjusted value of the `db_file_multiblock_read_count`.)

When we enable system statistics, the optimizer decides that the cost for the indexed path is still 68 for the I/O component, plus a small extra cost to cater to the CPU cost of acquiring blocks, and making a few data comparisons.

SYSTEM STATISTICS VS. OPTIMIZER_INDEX_COST_ADJ

One of the most important features of system statistics (or CPU costing) is that it allows the optimizer to balance the costs of single-block and multiblock reads properly by scaling *up* the cost of a multiblock read. As an early fix to this imbalance, the parameter `optimizer_index_cost_adj` appeared in 8*i*. In effect, this parameter appeared to be a percentage that would be used to scale *down* the cost of single block reads compared to multiblock reads. (See Tim Gorman's paper "The Search for Intelligent Life in the Cost Based Optimizer" at `www.evdbt.com` for the first useful paper published about this parameter.)

The flaw built into this fix was that it reduced the cost of single block reads, rather than increasing the cost of multiblock reads. On the plus side, this did tend to stop the optimizer from using excessive tablescans. On the minus side, the rounding built into the calculations could cause the optimizer to switch from a good index to a bad index, because their costs became identical after adjustment and rounding. When you scale down, rounding errors become more significant.

The cost for the tablescan path changes more dramatically when we switch to CPU costing. To work out the I/O cost component, we divide the high water mark by the actual stored `MBRC`, and round up: `ceiling(371/8) = 47`. Then we double it, because our `mreadtim` is twice our `sreadtim`, to get 94. We then add a little for the amount of CPU that will be used examining every row of the table.

The upshot of this is that if your system statistics really do represent the typical behavior of your system, then the cost based optimizer can produce a cost that is a more realistic reflection of actual resource usage and time to completion before it starts running the query.

But as the last line of the table of results shows—upgrading (or enabling system statistics some time after an upgrade) could leave you with lots of execution plans changing spontaneously, with switches from tablescans to indexed access paths being quite likely. In theory, any changes should be better paths, but I wouldn't guarantee that this will be true in every case.

Loose Ends

When you start experimenting with indexes, you may find that the numbers don't work quite correctly. The optimizer might use various adjustments in different circumstances that result in the calculations being off by plus or minus one.

We have already noted that there are variations between the versions of the Oracle (and sometimes within a version) about the use of `round()` and `ceil()`. It took me a long time to notice this because all my test cases just happened to produce results where `round()` and `ceil()` gave the same result. The moral is that you should never assume that you have worked out exactly what's going on; a good approximation is as close as you can ever get.

But just when you think you're close enough, you discover that there really isn't a formula, there is only a decision tree, with different formulae at the end at each of the branches. Here are a couple of common cases that may cause a variation between the baseline formula and the actual result:

- For unique indexes, or nonunique indexes supporting unique or primary key `constraints`, the optimizer uses the standard formula, and then subtracts 1. However, in the case of a nonunique index supporting a unique or primary key constraint, this adjustment is not made if the constraint is `deferrable`. So be careful—you may be tempted to make all your constraints deferrable *just in case*, but there are small side effects that may cause execution plans to change.

- Indexes where the `blevel` is set to 1 (so the index goes straight from the root block to the leaf blocks). The optimizer effectively ignores the `blevel` if every column in the index appears in an equality predicate. This is an interesting case, as a root block split (which could happen because just one row in the table has been updated) would then push the cost of the index up by two—which could change the access path. This is just one of many clues that small tables can be more important than large tables when you want to solve optimizer problems.

INDEXES ON SMALL TABLES

A strategy for handling indexes and statistics on small tables can be quite important because some small indexes may (apparently) need regular rebuilds. If they are on the cusp between a `blevel` of 1 and a `blevel` of 2, and you keep collecting statistics, one extra row may be all it takes to move you from a `blevel` of 1 to a `blevel` of 2. It's not a dramatic change in the data size, or the number of blocks in the index, but it causes a sudden jump in the costing calculation.

If you do have a couple of candidates in this unusual state, one option is to stop collecting the statistics, another is to set the `blevel` by hand after collecting the statistics, and a third is to rebuild the index every time prior to collecting the statistics.

Summary

Although the chapter was only intended to be an introduction to how the optimizer assesses the cost of using a simple B-tree index, we have covered a lot of ground. Key points to remember though are as follows:

The typical cost of using a B-tree index comprises three components—the depth of the index, based on the `blevel`; the number of index leaf blocks that will be visited, based on `leaf_blocks`; and the number of visits to table blocks, based on the `clustering_factor`.

The `clustering_factor` of an index is a primary indicator of how desirable the index appears to be to the optimizer. The drawback is that you need to compare the `clustering_factor` to the number of rows and blocks in the table—the number by itself is effectively meaningless.

If you have columns in an index that are often omitted from the `where` clause or have range-based tests applied to them, then they should generally be pushed towards to the end of the index definition; otherwise your query could end up doing a lot of work with index leaf blocks. The optimizer's cost calculation will reflect this, and may result in the index being ignored in favor of alternative paths.

System statistics should be enabled as part of your migration to 9*i* (or 10*g*), and you need to be aware of the impact this will have on execution plans.

Test Cases

The files in the download for this chapter are shown in Table 4-3.

Table 4-3. *Chapter 4 Test Cases*

Script	Comment
`btree_cost_01.sql`	Basic script used in this chapter
`rebuild_test.sql`	Indicates that it is possible that rebuilding an index can cause a problem
`btree_cost_02.sql`	Examples of cost calculations for range scans and partial use of index
`btree_cost_04.sql`	Example of in-list error in 8*i*
`btree_cost_03.sql`	Index full scans, index-only scans
`hack_stats.sql`	Script to modify statistics directly on the data dictionary
`btree_cost_02a.sql`	Repeats the `btree_cost_02.sql` example, but allows for some completely null entries
`btree_cost_05.sql`	Example of impact of CPU costing
`setenv.sql`	Sets a standardized environment for SQL*Plus

CHAPTER 5

The Clustering Factor

In the previous chapter, I warned you that the `clustering_factor` was an important factor in the cost of using a B-tree index for a range scan and could easily be the biggest cause of errors in the calculation of cost. It's time to find out why things can go wrong.

This chapter is very important because it describes many of the sensible strategies that DBAs adopt to improve performance or avoid contention, only to discover side effects that can leave the optimizer ignoring indexes that it ought to be using. Almost invariably, the *sensible strategy* has caused problems for some queries because of the impact it has had on the `clustering_factor`.

The `clustering_factor` is a single number that represents the degree to which data is randomly distributed through a table, and the concept of creating a number to represent the data scatter in a table is a good one. Unfortunately, some recent, and not-so-recent, features of Oracle can turn this magic number into a liability.

In all the discussions that follow, we shall be focusing very specifically on traditional **heap-organized** tables, and you will find that my examples of problematic indexes tend to be ones that are broadly time-based or sequence-based.

Baseline Example

To see how badly things can go wrong, we will start with a test case that mimics a common real-life pattern and see what the `clustering_factor` looks like when things go right.

We start with a table that has a two-part primary key: the first part being a date, and the second being a sequence number. We will then run five concurrent processes to execute a procedure emulating end-user activity. The procedure inserts data for 26 days at the rate of 200 rows per day—but to keep the experiment short, an entire day's input gets loaded in just 2 seconds. The total volume of data will be 5 processes * 26 days * 200 rows per day = 26,000 rows. On a reasonably modern piece of hardware, you should expect the data load to complete in less than a minute.

As usual, my demonstration environment starts with an 8KB block size, locally managed tablespaces with 1MB uniform extents, manual segment space management, and system statistics (`cpu_costing`) disabled (see the script `base_line.sql` in the online code suite).

```
create table t1(
        date_ord    date         constraint t1_dto_nn not null,
        seq_ord     number(6)    constraint t1_sqo_nn not null,
        small_vc    varchar2(10)
)
pctfree 90
pctused 10
;

create sequence t1_seq
;

create or replace procedure t1_load(i_tag varchar2) as
        m_date        date;
begin
        for i in 0..25 loop                             -- 26 days
                m_date :=  trunc(sysdate) + i;
                for j in 1..200 loop                    -- 200 rows per day
                        insert into t1 values(
                                m_date,
                                t1_seq.nextval,
                                i_tag || j              -- used to identify sessions
                        );
                        commit;
                        dbms_lock.sleep(0.01);          -- reduce contention
                end loop;
        end loop;
end;
/

rem
rem     Now set up sessions to run multiple copies
rem     of the procedure to populate the table
rem
```

You will notice the unusual values for `pctused` and `pctfree` on the table; these are there so that I can create a reasonably large table without generating a lot of data.

To run the test, you have to create the sequence, table, and procedure, and then start up five different sessions to run the procedure simultaneously. In the supplied script, the procedure also uses the `dbms_lock` package to synchronize the start time of the concurrent copies of itself, but to keep the code sample short, I have not included the extra lines in the text.

When the five concurrent executions have completed, you need to create the relevant index, and then generate and inspect the relevant statistics. The following results come from a system running 9*i*.

```
create index t1_i1 on t1(date_ord, seq_ord);

begin
        dbms_stats.gather_table_stats(
                user,
                't1',
                cascade => true,
                estimate_percent => null,
                method_opt => 'for all columns size 1'
        );
end;
/

select
        blocks,
        num_rows
from
        user_tables
where
        table_name = 'T1'
;

    BLOCKS   NUM_ROWS
---------- ----------
       749      26000

select
        index_name,
        blevel,
        leaf_blocks,
        clustering_factor
from
        user_indexes
where
        table_name = 'T1'
;

INDEX_NAME               BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
-------------------- ---------- ----------- -----------------
T1_I1                         1          86              1008
```

Notice how the `clustering_factor` in this case is similar to the number of blocks in the table, and very much smaller than the number of rows in the table. You may find the `clustering_factor` varies by 1% or 2% if you repeat the test, but it will probably stay somewhere around either the 750 mark or the 1,000 mark depending on whether you are running a single or multiple CPU machine. This looks as if it may be a good index, so let's test it with a slightly unfriendly (but perfectly ordinary) query that asks for all the data for a given date.

```
set autotrace traceonly explain

select  count(small_vc)
from    t1
where   date_ord = trunc(sysdate) + 7
;

set autotrace off

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=44 Card=1 Bytes=13)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=44 Card=1000 Bytes=13000)
   3    2       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=5 Card=1000)
```

Note that our query uses only one column of the two-column index. Applying the Wolfgang Breitling formula (which I introduced in Chapter 4), we can see the following figures drop out:

```
cost =
        blevel +
        ceil(effective index selectivity * leaf_blocks) +
        ceil(effective table selectivity * clustering_factor)
```

In this case, we are after 1 day out of 26—a selectivity of 3.846% or 0.03846—and the two selectivities are identical. Putting these figures into the formula:

```
cost =
        1 +
        ceil(0.03846 * 86) +
        ceil(0.03846 * 1,008)
        = 1 + 4 + 39 = 44
```

We know, and Oracle can observe through the `clustering_factor`, that all the rows for a given date have arrived at about the same time, and will be crammed into a small number of adjacent blocks. The index is used, even though Oracle has to fetch 1,000 rows, or nearly 4% of the data. This is good, as our simple model is probably a reasonable representation of many systems that are interested in *daily activity*.

Reducing Table Contention (Multiple Freelists)

But there may be a problem in our setup. In a high-concurrency system, we might have been suffering from a lot of contention. Take a look at the first few rows in the sample data that we have just produced. You will probably see something like this:

```
select
        /*+ full(t1) */
        rowid, date_ord, seq_ord, small_vc
from
        t1
where
        rownum <= 10
;

ROWID              DATE_ORD     SEQ_ORD SMALL_VC
------------------ --------- ---------- ----------
AAAMJHAAJAAAAAKAAA 18-FEB-04          1 A1
AAAMJHAAJAAAAAKAAB 18-FEB-04          2 B1
AAAMJHAAJAAAAAKAAC 18-FEB-04          3 C1
AAAMJHAAJAAAAAKAAD 18-FEB-04          4 A2
AAAMJHAAJAAAAAKAAE 18-FEB-04          5 D1
AAAMJHAAJAAAAAKAAF 18-FEB-04          6 E1
AAAMJHAAJAAAAAKAAG 18-FEB-04          7 B2
AAAMJHAAJAAAAAKAAH 18-FEB-04          8 D2
AAAMJHAAJAAAAAKAAI 18-FEB-04          9 B3
AAAMJHAAJAAAAAKAAJ 18-FEB-04         10 E2
```

Remember that the **extended rowid** is made up of the following:

- `object_id` First six letters (`AAAMJH`)
- Relative `file_id` Next three letters (`AAJ`)
- Block within file Next six letters (`AAAAAK`)
- Row within block Last three letters (`AAA`, `AAB`, `AAC` ...)

All these rows are in the same block (`AAAAAK`). In my test run, I populated the column `small_vc` with a tag that could be used to identify the process that inserted the row. All five of our processes were busy hitting the same table block at the same time. In a very busy system (in particular, one with a high degree of concurrency), we might have seen lots of **buffer busy waits** for blocks of class **data block** for whichever one block was the current focus of all the inserts.

How do we address this issue? Simple: we read the advice in the *Oracle Performance Tuning Guide and Reference* and (for older versions of Oracle particularly) realize that we should have created the table with multiple `freelists`. In this case, because we expect the typical degree of concurrency to be 5, we might go to exactly that limit, and create the table with the extra clause:

```
storage (freelists 5)
```

With this clause in place, Oracle maintains five linked lists of free blocks hanging from the table's **segment header block**, and when a process needs to insert a row, it uses its **process ID** to determine which list it should visit to find a free block. This means that (with just a little bit of luck) our five concurrent processes will never collide with each other; they will always be using five completely different table blocks to insert their rows.

FREELIST MANAGEMENT

The complete cycle of activity relating to freelist management is outside the scope of this book, but the following points are reasonably accurate for the simplest cases:

By default, a table is defined with just one **segment freelist**, and Oracle bumps the high water mark (HWM) by five blocks and adds those blocks to the freelist every time the freelist is emptied. Generally, it is only the top block on a freelist that is available for inserts in ordinary heap-organized tables.

If you specify multiple freelists, Oracle allocates one more segment freelist than you expect and uses the first one as the *master freelist*. This master freelist is used as the focal point for ensuring that all the other freelists behave in a reasonable way and stay about the same length (which is somewhere between zero and five blocks).

Historically, you could only set the `freelists` parameter when you created the table, but this changed somewhere in the 8*i* timeline (possibly in 8.1.6) so that you could modify the value used for *future* allocations with a simple, low-cost, `alter table` command.

Repeat the baseline test with `freelists` set to 5, though, and you will discover that the price you pay for reducing contention can be very high. Look what happened to the `clustering_factor` and the desirability of the index when I first made this change (script `free_lists.sql` in the online code suite):

```
INDEX_NAME               BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
-------------------- ---------- ----------- -----------------
T1_I1                         1          86             26000


select  count(small_vc)
from    t1
where   date_ord = trunc(sysdate) + 7
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=115 Card=1 Bytes=13)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=115 Card=1000 Bytes=13000)
```

The `clustering_factor` has changed from around 1,000 (close to the number of table blocks) to 26,000 (the number of table rows), so the optimizer thinks it is a truly appalling index and declines to use it for our single-date query. The saddest thing about this is that the data for that one date will still be in exactly the same 30 to 35 table blocks that they were in when we had `freelists` set to 1, it's just the row order that has changed.

In Chapter 4, I produced a schematic of a table and index showing the notional mechanism that Oracle used to count its way to the `clustering_factor`. If we produce a similar diagram for a simplified version of our latest example—using `freelists` set to 2, and pretending that Oracle adds only two blocks at a time to a freelist—then the schematic would look more like Figure 5-1.

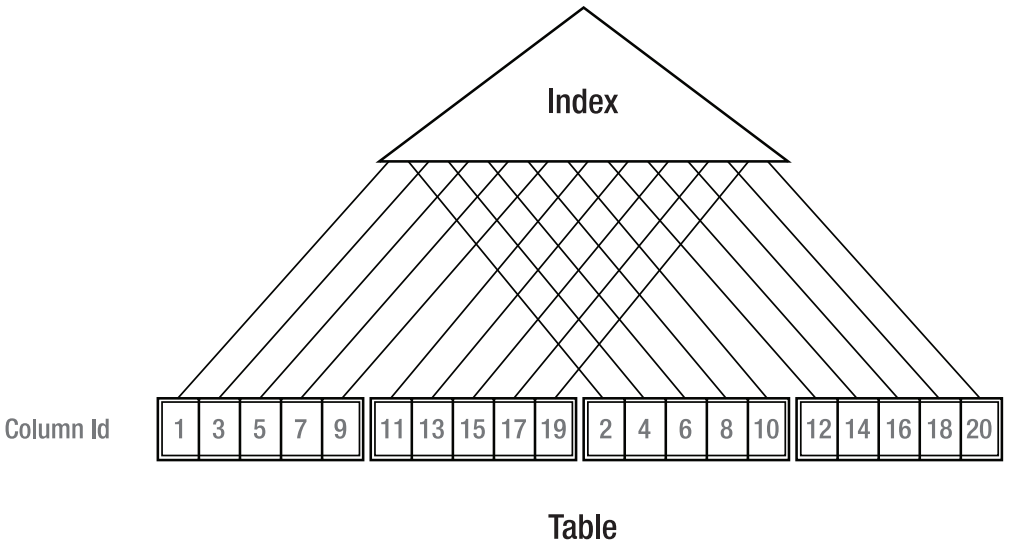

Figure 5-1. *The clustering_factor and multiple freelists*

Process 1 is busy inserting rows into block 1, but process 2 is using a different freelist, so it is busy inserting rows into block 3 (in a real system, it is more likely to be inserting rows into a position five blocks further down the table). But both processes are using the same sequence generator, and if the two processes happen to run perfectly in step, alternating values from the sequence will appear in alternating blocks. Consequently, as Oracle walks the index, it keeps stepping back and forth between the same pair of table blocks, incrementing the `clustering_factor` as it goes. The counter I displayed for the `clustering_factor` in the first schematic isn't needed here—because it simply stays in step with the values of the `ID` column.

If you look at the diagram, it is very obvious that it will take just two block reads to get all the values from 1 to 10, but Oracle's method for calculating the `clustering_factor` makes the optimizer think that it is going to have to visit 10 different blocks. The problem is that Oracle doesn't maintain a *recent history* when calculating the `clustering_factor`, it simply checks whether the current block is the same as the previous block.

FREELIST SELECTION

When a process needs to insert a row into a table with multiple `freelists`, it selects a freelist based on the process ID. (MetaLink note 1029850.6 quotes the algorithm as `mod(process_id, freelist count) + 1`.) This means that collisions between processes may still occur, irrespective of the number of `freelists` you define.

In my test run, I happened to have five processes that each picked a separate freelist. Your results may vary quite significantly. It may seem a little surprising that the choice is based on the process ID, rather than the session ID—but the rationale behind this choice may have been to minimize contention in a **shared server** (**MTS**) environment.

When you fix a table contention problem, you may find that queries that should be using index range scans suddenly start to use tablescans because of this simple arithmetic issue. We shall see an interesting fix for this problem shortly.

Reducing Leaf Block Contention (Reverse Key Indexes)

Before looking at fixes for a misleading value of the `clustering_factor`, let's examine a couple of other features that can produce the same effect. The first is the reverse key index, introduced in Oracle 8.1 as a mechanism for reducing contention (particularly in RAC systems) on the leading edge of sequence-based indexes.

A reverse key index operates by reversing the byte order of each column of the index before inserting the resulting value into the index structure. The effect of this is to turn sequential values into index entries that are randomly scattered. Consider, for example, the value (`date_ord`, `seq_no`) = (`'18-Feb-2004', 39`); we could use the `dump()` function to discover that internally this would be represented as `({78,68,2,12,1,1,1},{ c1,28})`:

```
select
        dump(date_ord,16)       date_dump,
        dump(seq_no,16)         seq_dump
from    t1
where   date_ord = to_date('18-feb-2004')
and     seq_no = 39
;

DATE_DUMP                       SEQ_DUMP
------------------------------  ------------------
Typ=12 Len=7: 78,68,2,12,1,1,1  Typ=2 Len=2: c1,28
```

but when it is reversed it becomes (`{1,1,1,12,2,68,78}, {28,c1}`):

```
select
        dump(reverse(date_ord),16)       date_dump,
        dump(reverse(seq_no),16)         seq_dump
from    t1
where   date_ord = to_date('18-feb-2004')
and     seq_no = 39
;

DATE_DUMP                       SEQ_DUMP
------------------------------  ------------------
Typ=12 Len=7: 1,1,1,12,2,68,78  Typ=2 Len=2: 28,c1
```

Note how the two columns are reversed separately, which means that in our example, all the entries for 18 February 2004 would still be close together in our index, but something odd would happen to the sequencing of the numeric portion within that date. If we dump out a section of index around the value (`'18-Feb-2004', 39`) with the `alter system dump datafile` command, we find the following ten values as consecutive entries for `seq_ord` (script `reverse.sql` in the online code suite produces the same result in a much more convenient fashion, so you can find out how far away the value 38 and 40 appear from 39):

```
REVERSED_SEQ_ORD                SEQ_ORD
------------------------------ ----------
28,7,c2                            639
28,8,c2                            739
28,9,c2                            839
28,a,c2                            939
28,c1                               39
29,2,c2                            140
29,3,c2                            240
29,4,c2                            340
29,5,c2                            440
29,6,c2                            540
```

What has this done to our `clustering_factor` and execution plan? Go back to the table we used in the baseline test (using the default value of one for `freelists`), and rebuild the index as a reverse key index (script `reversed_ind.sql` in the online code suite):

```
alter index t1_i1 rebuild reverse;

begin
        dbms_stats.gather_table_stats(
                user,
                't1',
                cascade => true,
                estimate_percent => null,
                method_opt => 'for all columns size 1'
        );
end;
/

INDEX_NAME               BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
-------------------- ---------- ----------- -----------------
T1_I1                         1          86             25962

select  count(small_vc)
from    t1
where   date_ord = trunc(sysdate) + 7
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=115 Card=1 Bytes=13)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=115 Card=1000 Bytes=13000)
```

The purpose of reversing the index is to scatter the index entries when incoming table entries hold sequential values—but the consequence of this is that adjacent index entries correspond to scattered table entries; in other words, the `clustering_factor` has just become extreme.

Because the clustering_factor is now close to the number of rows in our test case, the execution plan has changed from an index range scan to a tablescan, even though the rows we want are still in the same small cluster of table blocks.

Our data distribution has not changed, but Oracle's perception of it has because the mechanism for calculating the clustering_factor is not aware of the impact of reverse key indexes.

REVERSE KEY INDEXES AND RANGE SCANS

You may have heard that reverse key indexes cannot be used for range scans (despite the example in the text). This is only true in the special case of a single-column unique index (i.e., the simplest, but possibly commonest, case in which they might be used).

For a *multicolumn* index (e.g. {order_date, order_number}), Oracle can use a range scan for a query that tests for equality on the leading columns of the index. Similarly, an equality predicate on a single-column *nonunique* index—perhaps a less likely target for index reversal—will produce a range scan.

It is important to be careful with words—this common misunderstanding about reverse key indexes and range scans arises because a range scan *operation* need not be the result of a range-based *predicate*.

Reducing Table Contention (ASSM)

There is another new feature that can apparently destroy the effectiveness of an index. Again, it is a feature aimed at reducing contention by scattering the data. And, again, an attempt to solve one performance problem can introduce another.

In most of the test cases in this book, I have created my data in tablespaces that use the traditional **freelist space management** option. The main reason for sticking to freelist space management was to make sure that the tests were fairly reproducible. But for the next test, you need a tablespace that uses **automatic segment free space management** (more commonly known as **automatic segment space management**, or **ASSM**). For example:

```
create tablespace test_8k_assm
           blocksize 8K
           datafile 'd:\oracle\oradata\d9204\test_8k_assm.dbf'
           size 50m reuse
           extent management local
           uniform size 1M
           segment space management auto
;
```

Oracle introduced this new strategy for segment space management to avoid problems of contention on table blocks during inserts, especially in RAC environments. There are two key features to ASSM.

The first feature is structural: each segment in an ASSM tablespace uses a few blocks at the start of each extent (typically one or two for each 64 blocks in the extent) to maintain a map of all the other blocks in the extent, with a rough indication—accurate to the nearest quarter-block—of how much free space is available in each block.

MORE ON ASSM BLOCKS

My description of ASSM blocks isn't a complete picture of what happens, as the details can vary with block size, the overall segment size, and the contiguity of extents. You may find in a large object with several adjacent small extents that a single block in one extent maps all the blocks for the next two or three extents, up to a maximum of 256 blocks. You will find that the first extent of a segment is a special case—in an ASSM tablespace with 8KB block sizes, the segment header block is the fourth block of the segment!

Also, as far as the space map is concerned, the meaning of the expression *free* is a little fluid. When the bitmap reports space as free with entries like `21:75-100% free`, the range is a percentage of the block—but if you have set a very low `PCTFREE` for the object, you can find that the difference between 75–100% free and "full" is just a few bytes. (And Oracle seems to be a little slow to move blocks from the full status as you delete rows.)

The second feature of ASSM appears at run time: when a process needs to insert a row, it selects a space map block that is dictated by its process ID, and then picks from the space map a data block that is (again) dictated by the process ID. The net effect of ASSM is that concurrent processes will each tend to pick a different block to insert their rows, minimizing contention between processes without intervention by the DBA.

The most important phrase in the last sentence is a "different block." To avoid contention, different processes scatter their data across different blocks—this may give you a clue that something nasty could happen to the `clustering_factor`. Rerun the baseline test, but create a tablespace like the preceding one, and add the following line to your table creation statement (script `assm_test.sql` in the online code suite):

```
tablespace test_8k_assm
```

The results you get from the test this time might look like the following—but may look extremely different.

```
INDEX_NAME                BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
------------------------- ------ ----------- -----------------
T1_I1                          1          86             20558

select count(small_vc)
from   t1
where  date_ord = trunc(sysdate) + 7;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=116 Card=1 Bytes=13)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=116 Card=1000 Bytes=13000)
```

Again, without any change in the data insertion code, data definition, or activity of the end users, we have introduced a specific Oracle feature at the infrastructure level that has changed an execution plan from an indexed access path to a tablescan.

Your results on this test may be very different from mine. It is a feature of ASSM that the scattering of data on inserts is affected by the process IDs of the processes doing the insertion. In a couple of repeats of the test, after disconnecting and reconnecting to Oracle to get different process IDs, one run of the test produced a `clustering_factor` of 22,265 and the next a `clustering_factor` of 18,504.

Another piece of information that came through from this particular test was the randomness of inserts and contention when using ASSM. By running the following SQL, I could get a measure of how much collision occurred on the blocks in the table:

```
select
        ct, count(*)
from
        (
        select block, count(*) ct
        from
                (
                select
                        distinct dbms_rowid.rowid_block_number(rowid) block,
                        substr(small_vc,1,1)
                from t1
                )
        group by block
        )
group by ct
;
```

You will remember that I included a tag with each call to the procedure, and that the tag value was copied into the `small_vc` column. In the preceding query, I pick up the block number, and tag value (I used the letters `A` to `E` for the five processes running the procedure) to find out how many blocks ended up with rows inserted by all five processes, how many ended up with rows from any four processes, and so on. The results are shown in Table 5-1.

Table 5-1. *Collision Counts with ASSM and Freelists*

Number of Concurrent Processes Hitting Block	Blocks Affected ASSM Test 1	Blocks Affected ASSM Test 2	Blocks Affected FREELISTS 5 (a)	Blocks Affected FREELISTS 5 (b)
1	361	393	745	447
2	217	258		297
3	97	37		
4	38	33		
5	12	6		
Total blocks	725	727	745	744

As you can see, the ASSM tests (the second and third columns) still show a significant number of blocks with apparent collisions between concurrent processes. For example, in the third column of the table, we see 258 blocks that have been used by two of the data-loading processes.

For comparative purposes, I have also reported the results from two tests where I set `freelists` to 5. In `freelists` test (a), every single table block was subject to inserts from just one process. This isn't a guaranteed result, though; I just got lucky in that test as each of my five processes happened to pick a separate freelist. In `freelists` test (b), two processes attached themselves to the same freelist, so they were in a state of permanent collision for the duration of the run.

So you can see that a perfectly configured set of `freelists` can give you absolutely minimal contention on the table blocks during concurrent inserts—but it can go persistently wrong. On the other hand, a degree of randomness is introduced by ASSM that means you will hardly ever get perfect avoidance of contention, but the contention you get is more likely to be lower volume and spread over time—the fact that all five processes used block X for inserts does not necessarily mean that they were all trying to use it at exactly the same moment.

You might note that there also seems to be a small space saving—the tests with ASSM used about 20 data blocks fewer than the tests with `freelists`. This is a side effect of the ASSM insertion algorithm that manages to be a little more cunning and persistent than the insertion algorithm for traditional freelist processing. In my case, though, the saving was offset by the fact that 12 blocks were allocated for level 1 bitmaps (2 per extent), 1 block was allocated for the level 2 bitmap, and an extra 16 blocks at the top of the table had been preformatted and part used. There were actually 754 formatted blocks below the high water mark.

■ **Note** I have heard a couple of cases where Oracle gets a little too persistent with ASSM, and finds itself checking literally hundreds of data blocks to find one with enough space for a row—and doesn't mark the space map to indicate that the block is full when it should do so.

If you do need to start worrying about contention for highly concurrent inserts, you need to investigate the relative benefits and costs of the features that Oracle gives you for avoiding contention. In particular, you need to be careful when there are only a few processes doing all the work.

Reducing Contention in RAC (Freelist Groups)

Another option for reducing contention on tables subject to highly concurrent inserts, particularly for OPS (as it was) and RAC (as it is now) was the option to create a table with multiple `freelist groups`. The manuals still make some comment about this feature being relevant only to *multiple-instance* Oracle, but in fact it can be used in *single-instance* Oracle, where it has an effect similar to setting multiple `freelists`. (Of the two options, `freelists` is usually the sensible, and sufficient, one for single-instance Oracle.)

Multiple freelists, as we saw earlier, reduce contention because each freelist has its own *top block*, so insertion takes place to multiple table blocks instead of just one block. There is, however, still a point of contention because a freelist starts with a pointer to its top block, and all the pointers are held in the segment header block. When running RAC, even when you have eliminated contention on table blocks by specifying multiple freelists, you may still have an overheated segment header block bouncing around between instances.

You can specify multiple freelist groups as part of the specification for a table (see script flg.sql in the online code):

```
storage (freelist groups 5)
```

If you do this (in a non-ASSM tablespace), you get one block per freelist group at the start of the segment, after the segment header block. Each block (group) gets associated with an instance ID, and each block (group) handles an independent set of freelists. So the contention on the segment header is eliminated.

BUFFER BUSY WAITS

One of the classes recorded in the view v$waitstat is named *free list*. Given the existence of freelists and freelist groups, the name is a little ambiguous. In fact, this class refers to freelist group blocks. Waits for segment header freelists may be one of the causes when you see waits in the class *segment header*.

Using freelist groups can be very effective, once you work out how many to declare. If you make sure that you set the value sufficiently high that every instance you have (and every instance you are likely to want) has its own freelist group block, then the problems of contention on inserts, even on indexed, sequence-based columns, tend to disappear.

There are some scenarios where having multiple freelist groups on the table automatically eases contention on the table blocks in RAC systems. Moreover, contention on the index leaf blocks of sequence-based columns can be eliminated without side effects on the clustering_factor, provided you ensure two things. First that the **cache size** on the sequence is reasonably large, for example:

```
create sequence big_seq cache 10000;
```

Since each instance maintains its own "cache" (actually a just a low/high, or current/target, pair of numbers), the values inserted by one instance will differ greatly from the values inserted by another instance—and a big numerical difference is likely to turn into a pair of distant leaf blocks.

Second, you also need to leave freelists set to one on the table so that the section of index being populated by each instance won't be affected by the flip-flop effect described in the section on freelists.

But there is a significant side effect that you need to be aware of. Because each instance is effectively populating its own, discrete section of index, every instance, except the one populating the current highest valued section of the index, will cause *leaf block splits* to be *50/50* splits with no possibility of back-fill. In other words, on a RAC system you can take steps to avoid contention on the table, avoid contention on indexes on sequence-based columns, and avoid damaging the clustering_factor on those indexes; but the price you pay is that the size of the

index (specifically the leaf block count) will probably be nearly twice as large as it would be on a database running under a single instance.

A minor drawback with multiple `freelist groups` (as with multiple `freelists`) is that there will be multiple sets of new empty blocks waiting to be used. The high water mark on objects with multiple `freelist groups` will be a little higher than otherwise (with a worst case of 5 * `freelist groups` * `freelists`), but for large objects this probably won't be significant.

REBALANCING FREELIST GROUPS

A well-known issue of multiple `freelist groups` is that if you use a single process to delete a large volume of data, all the blocks freed by the delete will be associated with just one freelist group, and cannot be acquired automatically for use by the `freelists` of other `freelist groups`. This means that processes that are trying to insert new data will not be able to use the existing free space unless they have attached themselves to the "right" freelist group. So you could find an object formatting new blocks, and even adding new extents, when there was apparently plenty of free space.

To address this issue, there is a procedure with the ambiguous name of `dbms_repair.rebuild_freelists()`, which redistributes the free blocks evenly across all the object's `freelist groups`. Unfortunately, there is a bug in the code that makes the distribution uneven unless the process ID of the process running the procedure is a suitable value—so you may have to run the procedure a few times from different sessions to make it work optimally.

The major drawback to `freelist groups` is that you cannot change the number of `freelist groups` without rebuilding the object. So if you've carefully matched the number of `freelist groups` to the number of instances in your system, you have a reorganization problem when you decide to add a couple of nodes to the system. Remember to plan for growth.

Column Order

In Chapter 4, we saw how a range-based predicate (e.g., `col1 between 1 and 3`) would reduce the benefit of later columns in the index. Any predicates based on columns appearing after the earliest range-based predicate would be ignored when calculating the *effective index selectivity*—although they would still be used in the *effective table selectivity*—and this could leave Oracle with an unreasonably high figure for the cost of that index. This led to the suggestion that you might restructure some indexes to put columns that usually appeared with a range-based predicate toward the end of the index definition.

This is just one important consideration when deciding the column order of an index. Another is the possibility for improving the compressibility of an index by putting the least selective (most repetitive) columns first. Another is the option for arranging the columns so that some very popular queries can perform an `order by` without doing a sort (an execution mechanism that typically appears as `sort (order by) nosort` in the execution plan).

Whatever your reason for deciding to change the order of columns in an index, remember that it might change the `clustering_factor`. The knock-on effect of this might be that the calculated cost of the index for a range scan becomes so high that Oracle ignores the index.

We can use an exaggerated model to demonstrate this effect (see script `col_order.sql` in the online code suite):

```
create table t1
pctfree 90 pctused 10
as
select
        trunc((rownum-1)/ 100)     clustered,
        mod(rownum - 1, 100)       scattered,
        lpad(rownum,10)            small_vc
from
        all_objects
where
        rownum <= 10000
;

create index t1_i1_good on t1(clustered, scattered);
create index t1_i2_bad  on t1(scattered, clustered);

--      Collect statistics using dbms_stats here
```

I used the standard trick of setting a large `pctfree` to spread the table over a larger number of blocks without generating a huge amount of data. The 10,000 rather small rows created by this script required 278 blocks of storage. The `trunc()` function used in the `clustered` column gives me the values from 0 to 99 that each repeat 100 times before changing; the `mod()` function used in the `scattered` column keeps cycling through the numbers 0 to 99. I have created two indexes on the same pair of columns, reversing the column ordering from the *good* index to produce the *bad* index. The naming convention for each index is derived from an examination of its `clustering_factor`:

```
INDEX_NAME                BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
------------------------- ---------- ----------- -----------------
T1_I1_GOOD                     1          24               278
T1_I2_BAD                      1          24             10000
```

When we execute a query that (according to the general theory of range-based predicates) we think might use the index `t1_i2_bad`, this is what we see:

```
select
        count(small_vc)
from
        t1
where
        scattered = 50                  -- equality on 1st column of t1_i2_bad
and     clustered between 1 and 5       -- range on 2nd column of t1_i2_bad
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=1 Bytes=16)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=4 Card=6 Bytes=96)
   3    2       INDEX (RANGE SCAN) OF 'T1_I1_GOOD' (NON-UNIQUE) (Cost=3 Card=604)
```

Despite the fact that we have an index that seems to be a perfect match for the requirements of this query, its first column (with the equality predicate) is `scattered` and its second column (with the range-based predicate) is `clustered`, the optimizer has chosen to use the *wrong* index, the one that will be driven by the range-based predicate.

When we add a hint to force the optimizer to use the index that we thought was carefully crafted to match the query, Oracle will use it, but the cost is more than double that of the index that the optimizer chose by default.

```
select
        /*+ index(t1 t1_i2_bad) */
        count(small_vc)
from
        t1
where
        scattered = 50
and     clustered between 1 and 5
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=9 Card=1 Bytes=16)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=9 Card=6 Bytes=96)
   3    2       INDEX (RANGE SCAN) OF 'T1_I2_BAD' (NON-UNIQUE) (Cost=2 Card=6)
```

This really highlights the main defect in the optimizer's **derivation** of the `clustering_factor` it has used to work out the cost of an indexed access path. The optimizer estimates the number of visits to table blocks, but has no idea about how many of those visits should be *discounted* because they are returning to a recently visited block.

Irrespective of which index we use in this example, we will visit exactly the same number of table blocks—but the order in which we visit them will be different, and this has been enough to make a big difference to the optimizer's calculations.

For completeness, let's just run our statistics through the formula.

Selectivity of `'scattered = 50'`:	1 / 100 = 0.01
Selectivity of `'clustered between 1 and 5'`:	(5 – 1) / (99 – 0) + 2/100 = 0.060404
Combined selectivity:	0.01 * 0.060404 = 0.00060404

```
cost (t1_i1_good) =
     1 +
     ceil(0.060404 * 24) +     -- range on first column, invalidates second column
     ceil(0.00060404 * 278)    -- 2nd column can be used before visiting the table
                 = 1 + 2 + 1 = 4

cost (t1_i2_bad) =
     1 +
     ceil(0.00060404 * 24) +   -- can use both columns for start/stop keys
     ceil(0.00060404 * 10000)  -- but the clustering_factor is overwhelming
                 = 1 + 1 + 7 = 9
```

The numbers make the impact of the clustering_factor very obvious. Although the first set of figures shows that the range-based predicate on the first column has reduced the effectiveness of the t1_i1_good index, the reduction is minor compared with the impact caused by the enormous increase in the clustering_factor in the second set of figures.

In this example, the extra resources used because we have picked the wrong index will be minimal—we still visit exactly the same number of rows in the table—so no extra I/O there—and we happen to examine two leaf blocks rather than one when we use the wrong index.

The guaranteed penalty of using the wrong index would be a little extra CPU spent scanning an unnecessary number of entries in the index. There are 500 entries in either index where clustered between 1 and 5, and we will examine about 400 of them (from (1,50) to (5,50)) if we use index t1_i1_good for our query. There are 100 entries in the index where scattered = 50, and we will examine about five of them (from (50,1) to (50,5)) if we use index t1_i2_bad.

In real systems, the choice is more subtle than picking one of two indexes with the same columns in a slightly different order; the scope of the error becomes much larger with changes in complex execution plans—not just a little waste of CPU.

Extra Columns

It's not just a change in column order that could introduce a problem. It's a fairly common (and often effective) practice to add a column or two to an existing index. By now I'm sure you won't be surprised to discover that this, too, can make a dramatic difference to the clustering_factor, hence to the desirability of the index.

Imagine a system that includes a table for tracking product movements. It has a fairly obvious index on the movement_date, but after some time, it might become apparent to the DBA that a number of commonly used queries would benefit from the addition of the product_id to this index (see script extra_col.sql in the online code suite).

```
create table t1
as
select
        sysdate + trunc((rownum-1) / 500)      movement_date, -- 500 rows per day
        trunc(dbms_random.value(1,60.999))     product_id,
        trunc(dbms_random.value(1,10.000))     qty,
        lpad(rownum,10)                        small_vc,
        rpad('x',100)                          padding
from
        all_objects
where
        rownum <= 10000        -- 20 days * 500 rows per day.
;

rem     create index t1_i1 on t1(movement_date);              -- original index
rem     create index t1_i1 on t1(movement_date, product_id);  -- modified index

INDEX_COLUMNS            BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
------------------------ ---------- ----------- -----------------
movement_date                 1          27               182
movement_date, product_id     1          31              6645
```

Although the index size (as indicated by the leaf block count) has grown somewhat, the significant change is yet again the clustering_factor.

When the index is just (movement_date), we expect to see lots of rows for the same date entering the database at the same time, and the 500 rows we have created for each date will be packed in a clump of 9 or 10 adjacent blocks in the table at about 50 rows per block. An index based on just the movement_date will have a very good clustering_factor.

When we change the index to (movement_date, product_id), the data is still clustered by date, but any two entries for the same product_id on the same date are likely to be in two different table blocks in that little group of nine or ten. As we walk the index for a given date, we will be jumping back and forth around a small cluster of table blocks—not staying inside one table block for 50 steps of the index. Our clustering_factor will be hugely increased.

We can see the effect of this with a couple of queries:

```
select
        sum(qty)
from
        t1
where
        movement_date = trunc(sysdate) + 7
and     product_id = 44
;

select
        product_id, max(small_vc)
from
        t1
where
        movement_date = trunc(sysdate) + 7
group by
        product_id
;
```

The first query is an example of the type of query that encouraged us to add the extra column to the index. The second query is an example of a query that will suffer as a consequence of the change. In both cases, Oracle will be visiting the same little clump of about ten blocks in the table—but the extra column changes the order in which the rows are visited (which is what the clustering_factor is about), so the cost changes, and in the second case the execution plan changes for the worse.

We start with the execution plans for first query (before and after change), and note that the cost of the query does drop in a way that could reasonably represent the effect of the higher precision of the index:

```
Execution Plan (9.2.0.6 autotrace – first query - original index)
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=12 Card=1 Bytes=14)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=12 Card=8 Bytes=112)
   3    2       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=2 Card=500)
```

```
Execution Plan (9.2.0.6 autotrace - first query - modified index)
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=7 Card=1 Bytes=14)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=7 Card=8 Bytes=112)
   3    2       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=1 Card=8)
```

And now here are execution plans for second query (before and after change), which highlight the disaster that can occur when the `clustering_factor` no longer represents the original purpose of the index:

```
Execution Plan (9.2.0.6 autotrace - second query - original index)
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=19 Card=60 Bytes=1320)
   1    0   SORT (GROUP BY) (Cost=19 Card=60 Bytes=1320)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=12 Card=500 Bytes=11000)
   3    2       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=2 Card=500)

Execution Plan (9.2.0.6 autotrace - second query - modified index)
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=36 Card=60 Bytes=1320)
   1    0   SORT (GROUP BY) (Cost=36 Card=60 Bytes=1320)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=29 Card=500 Bytes=11000)
```

As you can see, the deceptively high `clustering_factor` has camouflaged the locality of the data, and the optimizer has switched from a precise indexed access path to a much more extravagant table scan.

Correcting the Statistics

So far I've spent all my time describing the way in which the `clustering_factor`, as calculated by Oracle, may not be truly representative of the way the data really is clustered in the table. With some understanding of the data and the way Oracle does its arithmetic, you can correct the problem, and I'd like to stress the word *correct*.

The sys_op_countchg() Technique

It is possible to tell Oracle anything you like about the statistics of your system, overriding any of the figures that it has collected; but the smart thing to do is to identify the numbers that are wrong and supply the right ones. It is not sensible simply to fiddle around creating numbers until some piece of SQL happens to work the way you want.

It is easy to hack the statistics, but your aim should be to give Oracle a better idea of the truth—because with an accurate image of your data, and the way you use it, the optimizer can do a better job.

Since the only thing I've been talking about in this chapter is the `clustering_factor`, I'm going to tell you how to modify it, and what to modify it to.

Look at the package dbms_stats. It contains two critical classes of procedures: get_xxx_stats and set_xxx_stats. For the purposes of this chapter, we are interested only in get_index_stats and set_index_stats. In principle, we can always adjust the clustering_factor of an index by a piece of PL/SQL that reads the statistics from the **data dictionary**, modifies some of them, and writes the modified values back to the data dictionary, for example (script hack_stats.sql in the online code suite):

```
declare

        m_numrows               number;
        m_numlblks              number;
        m_numdist               number;
        m_avglblk               number;
        m_avgdblk               number;
        m_clstfct               number;
        m_indlevel              number;

begin

        dbms_stats.get_index_stats(
                ownname         => NULL,
                indname         => '{index_name}',
                numrows         => m_numrows,
                numlblks        => m_numlblks,
                numdist         => m_numdist,
                avglblk         => m_avglblk,
                avgdblk         => m_avgdblk,
                clstfct         => m_clstfct,
                indlevel        => m_indlevel
        );

        m_clsfct := {something completely different};

        dbms_stats.set_index_stats(
                ownname         => NULL,
                indname         => '{index_name}',
                numrows         => m_numrows,
                numlblks        => m_numlblks,
                numdist         => m_numdist,
                avglblk         => m_avglblk,
                avgdblk         => m_avgdblk,
                clstfct         => m_clstfct,
                indlevel        => m_indlevel
        );
end;
/
```

HACKING THE DATA DICTIONARY

There is an enormous difference between hacking the data dictionary with a published, documented PL/SQL API, and hacking the data dictionary with statements like `update col$ set ...`.

In the former case, you may not understand what you are telling Oracle about your system, but at least you will be leaving the data dictionary in a self-consistent state. In the latter case, (a) you don't know how many other changes you should have made at the same time, and (b) you don't know if all your changes will actually arrive at the data dictionary, as it seems to get refreshed from the dictionary cache (`v$rowcache`) in a fairly random way, so (c) you can very easily leave your database in an inconsistent state that will lead to subsequence security breaches, crashes, and silent yet extensive data corruption.

The technique is simple; the subtle bit is deciding what value you should use for `clustering_factor`.

The answer to this question depends on the circumstances. However, let's start with a completely different question: how, exactly, does Oracle work out the `clustering_factor`? Call `dbms_stats.gather_index_stats()` with `sql_trace` switched on, and if you are running Oracle 9*i*, you will find out. For a simple B-tree index, the trace file will contain a piece of SQL looking something like the following (try this after running script `base_line.sql`):

```
/*
        Do this from an SQL*Plus session then examine the trace file.

alter session set sql_trace true;

begin
        dbms_stats.gather_index_stats(
                user,
                't1_i1',
                estimate_percent => null
        );
end;
/

exit

*/

 select /*+
                cursor_sharing_exact
                dynamic_sampling(0)
                no_monitoring
                no_expand
                index(t,"T1_I1")
                noparallel_index(t,"T1_I1")
        */
```

```
        count(*)                                                as nrw,
        count(distinct sys_op_lbid(49721,'L',t.rowid))          as nlb,
        count(
                distinct hextoraw(
                        sys_op_descend("DATE_ORD")||sys_op_descend("SEQ_ORD")
                )
        )                                                       as ndk,
        sys_op_countchg(substrb(t.rowid,1,15),1)                as clf
from
        "TEST_USER"."T1"  t
where
        "DATE_ORD" is not null
or      "SEQ_ORD" is not null
;
```

In the preceding query, the column `nrw` turns into the number of rows in the index (`user_indexes.num_rows`), `nlb` turns into the number of leaf blocks (`user_indexes.leaf_blocks`), `ndk` becomes the number of distinct keys in the index (`user_indexes.distinct_keys`), and `clf` becomes the `clustering_factor` (`user_indexes.clustering_factor`).

The appearance of the `sys_op_descend()` function came as a bit of a surprise; it is the function normally used to generate the values stored for indexes with descending columns, but I think it is used here to insert a separator byte between the columns of a multicolumn index, so that the counts will be able to distinguish between items like `('aaa','b')` and `('aa','ab')`—which would otherwise appear to be identical.

The `sys_op_lbid()` function seems to return a **leaf block ID**—and the exact meaning of the ID returned is dependent on the single letter parameter. In this example, 49721 is the `object_id` of the index named in the `index` hint, and the effect of the `L` parameter seems to be to return the absolute address of the first entry of the leaf block in which the supplied table rowid exists. (There are options for **index organized tables** [**IOT**s], secondary indexes on IOTs, bitmap indexes, partitioned indexes, and so on.)

But the most interesting function for our purposes is `sys_op_countchg()`. Judging from its name, this function is probably *counting changes*, and the first input parameter is the block ID portion (`object_id`, relative file number, and block number) of the table's rowid, so the function is clearly matching our notional description of how the `clustering_factor` is calculated. But what is that `1` we see as the second parameter?

When I first understood how the `clustering_factor` was defined, I soon realized that its biggest flaw was that Oracle wasn't remembering recent history as it walked the index; it only remembered the previous table block so that it could check whether the latest row was in the same table block as last time or in a new table block. So when I saw this function, my first guess (or hope) was that the second parameter was a method of telling Oracle to remember a list of previous block visits as it walked the index.

Remember the table that I created in script `freelists.sql`, with a `freelists` set to 5. Watch what happens if we run Oracle's own stats collection query (rewritten and tidied as follows) against this table—using different values for that second parameter (script `clufac_calc.sql` in the online code suite):

```
select /*+
                cursor_sharing_exact
                dynamic_sampling(0)
                no_monitoring
                no_expand
                index (t,"T1_I1")
                noparallel_index(t,"T1_I1")
        */
        sys_op_countchg(substrb(t.rowid,1,15),&m_history) as clf
from
        "TEST_USER"."T1"  t
where
        "DATE_ORD" is not null
or      "SEQ_ORD" is not null
;

Enter value for m_history: 5

       CLF
----------
       746

1 row selected.
```

I had to use a substitution parameter while running this query from a simple SQL*Plus session, as the function crashed if I tried to code the query into a PL/SQL loop with a PL/SQL variable as the input. I ran the query seven times in a row, entering a different value for `m_history` each time, and have summarized the results of the test in Table 5-2. The first set of results comes from a run of `freelists.sql` where I had used five concurrent processes and been lucky enough to get perfect separation. The second set comes from a run where I doubled the number of concurrent processes from 5 to 10, with a less-fortunate separation of processes—giving me 52,000 rows and 1,502 blocks in the table.

Table 5-2. *Effects of Changing the Mystery Parameter on sys_op_countchg*

m_history	Calculated CLF (5 Processes)	Calculated CLF (10 Processes)
1	26,000	43,615
2	26,000	34,533
3	26,000	25,652
4	25,948	16,835
5	746	3,212
6	746	1,742
7	746	1,496

Just as the value I enter for `m_history` matches the `freelists` setting for the table, the `clustering_factor` suddenly changes from *much too big* to *really quite reasonable*! It's hard to believe that this is entirely a coincidence.

So using Oracle's own function for calculating the `clustering_factor`, but substituting the `freelists` value for the table, may be a valid method for correcting some errors in the `clustering_factor` for indexes on strongly sequenced data. (The same strategy applies if you use multiple `freelist groups`—but multiply `freelists` by `freelist groups` to set the second parameter.)

Can a similar strategy be used to find a modified `clustering_factor` in other circumstances? I think the answer is a cautious "yes" for tables that are in ASSM tablespaces.

Remember that Oracle currently allocates and formats 16 new blocks at a time when using automatic segment space management (even when the extent sizes are very large, apparently). This means that new data will be roughly scattered across groups of 16 blocks, rather than being tightly packed.

Calling Oracle's `sys_op_countchg()` function with a parameter of 16 could be enough to produce a reasonable `clustering_factor` where Oracle currently produces a meaningless one. The value 16 should, however, be used as an **upper bound**. If your real degree of concurrency is typically less than 16, then your actual degree of concurrency would probably be more appropriate.

Whatever you do when experimenting with this function—don't simply apply it across the board to all indexes, or even all indexes on a particular table. There will probably be just a handful of critical indexes where it is a good way of telling Oracle a little more of the truth about your system—in other cases you will simply be confusing the issue.

Informal Strategies

We still have to deal with problems like reverse key indexes, indexes with added columns, and indexes where the column order has been rearranged. Playing around with the `sys_op_countchg()` function is not going to help in these cases.

However, if you consider the examples in this chapter, you will see that the they have a common thread to them. In each case the *driving* use of the index comes from a subset of the columns.

In the reverse key example, (`date_ord, seq_no`), the critical use of the index depended on only the `date_ord` column and the presence of the `seq_no` added no precision to our queries.

In the example about adding extra columns, (`date_movement, product_id`), the critical use of the index was the `date_movement`; the `product_id` was a little tweak to enhance performance (for certain queries).

In the example of rearranging columns, (`scattered, clustered`), the argument is weaker, but we can detect that an underlying pattern in the table is strongly dictated by the clustered column, regardless of the fact that the index columns are not ordered in a way that picks this up.

In all three cases, you could argue that a *more appropriate* `clustering_factor` could be found by creating an index using only the *driving* columns, calculating the `clustering_factor` for that index, and transferring the result to the original index. (You might want to do this on a backup copy of the database, of course.)

I think the argument for doing this is very good in the first two cases mentioned previously, but a little weak for the third case. In the third case, the validity of the argument depends much more on the actual use of the index, and the nature of the queries. However, when the *driving*

column argument fails, you may be able to fall back to the sys_op_countchg() technique. In the example, the data is grouped by the clustered column with a group of 9 or 10 blocks—calling the sys_op_countchg() function with the value 9 may be the best way of finding an appropriate clustering_factor for your use of that index.

Finally, there is the option of just knowing the right answer. If you know that a typical key value will find all its data in (say) 5 table blocks, but Oracle thinks it will have to visit 100 table blocks, then you can simply divide the clustering_factor by 20 to tell Oracle the truth. To find out how many table blocks Oracle thinks it has to visit, simply look at the column user_indexes.avg_data_blocks_per_key, which is simply a restated form of the clustering_factor, calculated as round (clustering_factor / distinct_keys).

Loose Ends

There are many other cases to consider if you want to produce a complete picture of how the clustering_factor can affect the optimizer, and I don't have space to go into them, but here's a thought for the future. Oracle 10*g* has introduced a mechanism to compact a table online. This only works for a table with **row movement** enabled that is stored in a tablespace using ASSM. You might use a sequence of commands like the following to compact a table:

```
alter table x enable row movement;
alter table x shrink space compact;       -- moves rows around
alter table x shrink space;               -- drops the high water mark
```

Before you rush into using this feature, just remember that it allows you to reclaim space by filling holes at the start of the table with data moved from the end of the table. In other words, any natural clustering of data based on arrival time could be lost as data is moved one row at a time from one end of the table to the other. Be careful about the effect this could have on the clustering_factor and desirability of the indexes on such a table.

Summary

The clustering_factor is very important for costing index range scans; but there are some features of Oracle, and some performance-related strategies, that result in an unsuitable value for the clustering_factor.

In many cases, we can predict the problems that are likely to happen, and use alternative methods for generating a more appropriate clustering_factor. We can always use the dbms_stats package to patch a correct clustering_factor into place.

If the clustering_factor is exaggerated because of multiple freelists, or the use of ASSM, then you can use Oracle's internal code for generating the clustering_factor with a modified value for the second parameter of the sys_op_countchg() function to get a more realistic value.

If the clustering_factor is exaggerated because of reverse key indexes, added columns, or even column reordering, then you may be able to generate a value based on knowing that the real functionality of the index relies on a subset of the columns. If necessary, build the reduced index on the backup data set, generate the correct clustering_factor, and transfer it to the production index.

Adjusting the clustering_factor really isn't hacking or cheating; it is simply ensuring that the optimizer has better information than it can derive (at present) for itself.

Test Cases

The files in the download for this chapter are shown in Table 5-3.

Table 5-3. *Chapter 5 Test Cases*

Script	Comments
base_line.sql	Script to create the baseline test with freelists set to 1
free_lists.sql	Repeats the test with freelists set to 5
reversed_ind.sql	Repeats the test and then reverses the index
reverse.sql	SQL to dump a list of numbers sorted by their reversed internal form
assm_test.sql	Repeats the test case in a tablespace set to ASSM
flg.sql	Repeats the test with freelists set to two and freelist groups set to three
col_order.sql	Demonstration of how changing the column order affects the clustering_factor
extra_col.sql	Demonstration of the effects of adding a column to an existing index
hack_stats.sql	Script to modify statistics directly on the data dictionary
clufac_calc.sql	The SQL used by the dbms_stats package to calculate the clustering_factor
setenv.sql	Sets a standardized environment for SQL*Plus

CHAPTER 6

Selectivity Issues

So far, I have stuck with all the easy options. I've been using numeric data, avoiding nulls, generating nice data distributions, and taking half a dozen other routes to making the optimizer behave well.

However, there are numerous ways in which the arithmetic works as I've described but still produces results that look completely wrong. In this chapter, I discuss the commonest reasons why the standard **selectivity** calculations produce unsuitable answers, and how you can (sometimes) bypass the problem. I also discuss a couple of features that may cause the optimizer to apply the arithmetic to the wrong predicates.

In Chapter 3, I avoided going into details about histograms. I'm still going to avoid them here. The effects of (the two kinds of) histograms add yet another layer of complexity to what's going on in the selectivity calculations, and there are plenty of oddities to clear up before we have to face histograms.

I will start by showing you how to apply the standard formula to date and character types, and then discuss what happens if we store data in columns of the wrong data type. We can then move on to the ways in which "special values" can cause problems, and close with cases where Oracle does arithmetic with predicates that you didn't know you had.

Different Data Types

We have been using the basic **selectivity** formula on numeric data types only, so far. The time has come to turn to a couple of other common data types, just in case there are any differences we need to be aware of.

We have been using the basic formula for the selectivity of a range-based predicate, and know that this varies slightly depending on whether you have zero, one, or two closed (meaning the equality in <=, >=) ends to your range. Using `N` as the number of closed ends, the version of the formula that uses `user_tab_columns.num_distinct` can be written as

```
(required range) / (column high value - column low value) + N / num_distinct
```

For example, in a column with 1,000 different (integer) values ranging from 1 to 1000, and the pair of predicates `colX > 10` and `colX <= 20`, you have one closed end (`<=`), and the formula would give you

```
(20 - 10) / (1000 - 1) + 1/1000 = 10/999 + 1/1000
```

Intuition will tell you that this example hasn't produced quite the right answer since we probably wanted 10 possible values out of 1,000, but it's pretty close.

Date Values

Things don't change much if you start to use **date** data types. Consider a date-only column holding values from 1 January 2000 to 31 December 2004; a predicate `date_col between 30th Dec 2002 and 5th Jan 2003` would then give you a selectivity of

```
(5th Jan 2003 - 30th Dec 2002 ) / ( 31st Dec 2004 - 1st Jan 2000 ) +
2/(number of different dates)
```

Fortunately, Oracle can do proper date arithmetic, so this turns into the following:

```
6               /               1826            + 2/1827
```

Again, this is not quite the right answer, as a human interpretation of the requirements would have spotted that you were selecting 7 dates out of a list of 1,827, with a selectivity of exactly 7/1,827.

The error in both these examples is just demonstrating the problem raised in Chapter 3. Oracle is using arithmetic for continuous data, but in many cases people use discrete data (i.e., lists of specific values). When the number of distinct values in our lists grows large, of course, the error in the calculation is usually small—but for lists with only a few distinct values, you need to be careful.

Character Values

Let's try one more example. What does it mean to measure the *total range* of a character column where the lowest value is *'Aardvark'* and the highest is *'Zymurgy'*? And how do you work out the *difference* between two words when the predicate is `colC between 'Apple' and 'Blueberry'`? To find the answers to these questions, a good place to start looking is the `user_tab_histograms` view where you discover that Oracle uses a numeric representation of character strings. (We aren't going to discuss what the optimizer does with this information, we are merely going to take advantage of the fact that we can see the values that have been generated.)

In a test table, `t1`, with two columns, one of type `varchar2(10)` and one of type `char(10)`, I have inserted the strings `'Aardvark'`, `'Apple'`, `'Blueberry'`, and `'Zymurgy'` in both columns, and then generated statistics—including histograms.

Columns of type `char(n)` are space padded to their full declared length (apart from the special case when the contents are null), which is why I have included an example of a `char(n)` in this test. As usual, my demonstration environment starts with an 8KB block size, locally managed tablespaces with a 1MB extent, manual segment space management, and system statistics (`cpu_costing`) disabled (see script `char_types.sql` in the online code suite):

```
create table t1 (
        v10         varchar2(10),
        c10         char(10)
)
;

--      Insert the data and generate histograms (see script in online code suite)
```

```
select
        column_name,
        endpoint_number,
        endpoint_value
from
        user_tab_histograms
where
        table_name = 'T1'
order by
        column_name,
        endpoint_Number
;

Col   End no                                          End Value
----- ------ -------------------------------------------------
C10        1  339,475,752,638,459,000,000,000,000,000,000,000     -- 'Aardvark  '
           2  339,779,832,781,209,000,000,000,000,000,000,000     -- 'Apple     '
           3  344,891,393,972,447,000,000,000,000,000,000,000     -- 'Blueberry '
           4  469,769,561,047,943,000,000,000,000,000,000,000     -- 'Zymurgy   '

V10        1  339,475,752,638,459,000,000,000,000,000,000,000     -- 'Aardvark'
           2  339,779,832,781,057,000,000,000,000,000,000,000     -- 'Apple'
           3  344,891,393,972,447,000,000,000,000,000,000,000     -- 'Blueberry'
           4  469,769,561,047,943,000,000,000,000,000,000,000     -- 'Zymurgy'
```

It's not immediately obvious what these numbers represent, but we can get further clues from the `user_tab_columns` view and (going back to 8*i*) the `user_tab_histograms` view where the column `endpoint_actual_value` is always populated.

■ **Note** In 9*i*, Oracle only populates the `endpoint_actual_value` column of the histogram views if there is at least one pair of values that, roughly speaking, are identical for the first six characters—which is why I have used 8*i* to generate the output for this example.

Oracle appears to behave as follows:

- Extract a maximum of 32 bytes from the column; this representation of the column value is how the `low_value`, `high_value` and `end_point_actual` values are stored.
- Extract the first 15 bytes from the 32, padding with zeros at the right if necessary.
- Convert the 15 bytes hex number to decimal and round to 15 significant figures.

Let's take a worked example—the string `'Aardvark'`—to see if we can end up with the value stored in the histogram:

- 'Aardvark', when dumped in hex from the char(10) column, contains the following list of byte values (note the 20,20 space padding that is imposed by the char(10) definition): '41,61,72,64,76,61,72,6B,20,20'.
- Since this is less than 15 bytes, we append a few zeros to produce the number 0x416172647661726B20200000000000. (If the column had been declared as char(40) the value would already have been padded with spaces (0x20) up to 40 characters, so we would stop at the fifteenth byte, and the value we would use would look like 0x416172647661726B20202020202020.)
- Converting this rather large hex number to decimal we get 339,475,752,638,459,043,065,991,628,037,554,176.
- And if we throw away everything after the first 15 digits then, as required, we get 339,475,752,638,459,000,000,000,000,000,000,000.

Looking at the rounding, you might notice that after about the first six or seven characters of a string, the rest of the letters don't have any impact on the numeric representation used by the optimizer—which is why the numeric values for 'Apple' are the only ones that vary when comparing the char(10) with the varchar2(10) versions. We don't really need to go into any more detail—but you could imagine that the optimizer might have trouble coping with data consisting (for example) of URLs, when lots of them start with http://. (A problem ameliorated, but not solved, by the fact that the first 32 characters of the URL would be stored as the endpoint_actual_value.)

The problem can be much worse, in fact, because of the increasing popularity of **national language support**. If you pick a **multibyte** character set for your database character set, then Oracle will be using the first 15 **bytes** of the string, not the first 15 **characters**. So the precision gets even worse. (Of course, if you switch to a fixed-width multibyte character set, all your character data gets longer, so this introduces a whole new area of performance testing anyway.) See script nchar_types.sql in the online code suite.

Character strings can cause massive problems with range-based queries, but the problems show up most commonly when you aren't really thinking about strings—as you will see in the next section.

Daft Data Types

One of the first examples in this chapter examined the predicate date_col between 30th Dec 2002 and 5th Jan 2003 on a table holding data from 1 January 2000 to 31 December 2004, and we saw how the optimizer could work out the selectivity of such a predicate. But let's see how things can go wrong in popular implementations of storing dates. We create and populate a table with one row per date over a five-year date range (see script date_oddity.sql in the online code suite).

```
create table t1 (
        d1          date,
        n1          number(8),
        v1          varchar2(8)
)
;

insert into t1
select
        d1,
        to_number(to_char(d1,'yyyymmdd')),
        to_char(d1,'yyyymmdd')
from
        (
        select
                to_date('31-Dec-1999') + rownum        d1
        from
                all_objects
        where
                rownum <= 1827
        )
;
```

In this example, I have stored the same information in three different ways. The first column is a proper Oracle date. The second and third columns are popular options for the *database-independent* applications that tend to store the date as a character string in the format YYYYMMDD (year/month/day) or its numeric equivalent. How does this affect the way the optimizer copes with the following simple queries?

```
select  *
from    t1
where   d1 between to_date('30-Dec-2002','dd-mon-yyyy')
               and to_date('05-Jan-2003','dd-mon-yyyy')
;

select  *
from    t1
where   n1 between 20021230 and 20030105
;

select  *
from    t1
where   v1 between '20021230' and '20030105'
;
```

In all three cases, Oracle will have to do a full tablescan as we haven't created any indexes—but how many rows does the optimizer think the query will return? Run each query through `autotrace`, and check the cardinality. Table 6-1 shows the results for a few different versions of Oracle.

Table 6-1. *Different Data Types Give Different Cardinality*

Column Type	Cardinality 8.1.7.4	Cardinality 9.2.0.4/10.1.0.2	Cardinality 9.2.0.6 /10.1.0.4
Date	9	8	8
Numeric	397	396	396
Character	457	457	396

The proper date column has produced the right answer (nearly); the error is the usual one of the optimizer using arithmetic appropriate to continuously varying values when we are really handling a relatively small list of discrete values. Oracle understands what dates are, what they mean, and how to do arithmetic with them.

But what's gone wrong with the numeric and character versions? Nothing, really—we've just hidden from the optimizer the fact that they are really holding dates, so the optimizer has used the standard arithmetic on a data set that is a little peculiar. Remember the standard formula for a range—in this case using the version that is closed at both ends (`between` is a short-hand for `'>= X and <= Y'`). Let's apply it for the numeric case:

```
Selectivity = (required range) / (high value - low value) + 2/num_distinct
                = (20030105 - 20021230) / (20041231 - 20000101) + 2/1827
                =         8875          /         41130         + 2/1827
                = 0.215779 + 0.001095
                = 0.216874           -- more than 20% of the data apparently!
```

To finish things off, multiply the selectivity by the number of rows in the table (1,827) and we get 1,827 * 0.216874 = 396.228.

The optimizer doesn't know that we are dealing with dates, so although we see numbers that we can recognize as a date range that crosses a month boundary or a year boundary, the optimizer doesn't see this as "the next day," it simply sees a huge gap—and we haven't given the optimizer any information about gaps. It's inevitable that the arithmetic will go wrong.

To make it easier to show that the value of 457 reported for the character types in older versions is following the standard formulae, I created a PL/SQL function to apply the conversion algorithm I described earlier on in the chapter (see script `char_fun.sql` in the online code suite). Consequently, I can just run the following SQL statement to work out the cardinality:

```
select
        round(
                1827 * (
                        2/1827 +
                        (cbo_char_value('20030105') - cbo_char_value('20021230')) /
                        (cbo_char_value('20041231') - cbo_char_value('20000101'))
                ),2
        )               cardinality
from
        dual
;

CARDINALITY
-----------
     456.51
```

Of course, you will have noticed that the cardinality changes as you move to 9.2.0.6 and 10.1.0.4 (so watch out on the upgrades), and the nature of the change seems to be rather devious. Somehow, Oracle seems to have treated the `varchar2` column and its predicate values as if they were numeric. It is possible that a programmer in the CBO group has slipped in a "cunning guess" tweak to the optimizer in this very special case. The tweak disappears as soon as you format the date-like string to include nonnumeric characters (e.g., `'2002-01-02'`).

If your application suffers from this use of incorrect data types, you may be able to perform some damage limitation by creating histograms on the critical columns. In this example, the numeric and character columns covered 60 months, so if you drew a picture of the data, you would see 60 spikes and 59 gaps. Try building a histogram of about 120 buckets—or 180, or 240—and see what happens.

Table 6-2 shows you the effect of building a histogram with 120 buckets on the two non-date columns. The improvement is significant—though the answers are still far from perfect. The slight difference between 8*i* and the other versions is due to a change in the way Oracle generates histograms, which we will examine in Chapter 7.

Table 6-2. *Histograms Can Help Overcome Incorrect Data Typing*

Column Type	Original	8*i*	9*i*/10*g*
Numeric	397/398	16	15
Character	457/397	16	15

Another piece of general-purpose intervention that may help the optimizer to arrive at roughly the right selectivity is to introduce **function-based indexes** (*FBI*). Of course, adding an index to a table is an overhead that should be considered carefully; but if you have SQL that needs to get the selectivity correct (and it isn't appropriate to slip in a `cardinality()` or `selectivity()` hint), then you may be able to rewrite your code to cater to the presence of function-based indexes.

For example, create an index on `to_date(v1,'yyyymmdd')`:

```
create index t1_v1 on t1(to_date(v1,'yyyymmdd'));

begin
        dbms_stats.gather_index_stats(
                ownname          => user,
                indname          =>'T1_V1',
                estimate_percent => null
        );
end;
/

begin
        dbms_stats.gather_table_stats(
                ownname          => user,
                tabname          =>'T1',
                cascade          => false,
                estimate_percent => null,
                method_opt       => 'for all hidden columns size 1'
        );
end;
/
```

I have used a call to the `gather_index_stats()` procedure, followed by a call to the `gather_table_stats()` procedure, with the `method_opt` of `for all hidden columns`. From 9*i* onwards, you might want to query view `user_tab_cols` to identify just those *virtual columns* that are in need of statistics, rather than hitting every column in the table.

With this setup in place, you finally convert your code from

```
where    v1 between '20021203' and '20030105'
```

to

```
where    to_date(v1,'yyyymmdd') between to_date('30-Dec-2002','dd-mon-yyyy')
                                    and to_date('05-Jan-2003','dd-mon-yyyy')
```

The optimizer will be able to use the statistics on the *virtual column* defined in your index to come up with the correct cardinality, even if the execution plan doesn't use the index.

Leading Zeros

One of the other ways in which the data-type error rears its ugly head is in the use of **synthetic keys** (also known as **surrogate keys**, or **meaningless IDs**). Various problems start to appear in more complex systems when meaningless numbers are injected into the data, and these can be exaggerated when the synthetic number is actually stored as a fixed length character column with leading zeros.

Take, for example, a site that I visited recently that had no proper primary keys, but was using a `char(18)` for every identifier in the system. At the time of writing, a few of the sequences had reached the million mark, so a typical stored value would be `000000000001000123`. Most of

the queries on a system using this type of strategy are likely to be of the form `where id = {string constant};` but if they start using range-based predicates, strange performance problems are likely to appear.

The script `char_seq.sql` in the online code suite emulates the situation by creating a table of two million rows with an `id` column that is generated by zero-padding a sequence number. When I first wrote this section, the current versions of Oracle were 9.2.0.4 and 10.1.0.2, and this is the result I got after generating simple statistics and running the following query with `autotrace` on:

```
select
        *
from    t1
where
        id between '000000000000060000'
               and '000000000000070000'
;

Execution Plan (9.2.0.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1039 Card=17 Bytes=323)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=1039 Card=17 Bytes=323)
```

Note the extremely low calculated cardinality (17 rows) for a query that is visibly supposed to return about 10,000 rows (the numbers varied slightly for other versions of Oracle, but were still wrong by two or three orders of magnitude).

As with the problem of using character or number types for dates, I was able to reduce the error by creating histograms, or using function-based indexes. With a histogram of the default 75 buckets, the execution plan showed an estimated cardinality of 8,924—which is at least in the right ballpark.

■ **Note** The default action for 10*g* when gathering table statistics is to use the `auto_sample_size` to generate the histogram, but in this particular example this resulted in the calculated cardinality coming out as 1!

After creating an index that included the expression `to_number(id)` as one of its columns, regenerating the statistics (without a histogram), and adjusting the text of the query accordingly, the calculated cardinality was close to 10,000 for 9*i* and 10*g* (but only 5,000 under 8*i*).

But time passes, and by the time I came to reviewing this chapter, I was running 9.2.0.6 and 10.1.0.4, and everything had changed. This is the execution plan I got in 9.2.0.6:

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1039 Card=10002 Bytes=220044)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=1039 Card=10002 Bytes=220044)
```

Magically, the cardinality is correct. And it's the same little tweak that appeared in the section on dates. As far as I can make out, if the `user_tab_columns.low_value`, `user_tab_columns.high_value`, and literal predicate values look like numbers, then the optimizer works out the selectivity as if it were handling a numeric column.

So, for the purposes of making the demonstration in 9.2.0.6 and 10.1.0.4 display the same problems it used to under earlier versions of Oracle, I had to modify the SQL in script `char_seq.sql` so that the values started with an `A`.

Deadly Defaults

Imagine you have accumulated five years' worth of data in your accounting system—say from 1 January 2000 to 31 December 2004—and decide to run a report spanning all the data in 2003. It seems likely that all the queries that have a predicate `where date_closed between 1st Jan 2003 and 31st Dec 2003` should be using tablescans since they are likely to be querying nearly 20% of the data.

You've stored the dates as real Oracle `date` columns—so what could possibly go wrong? (We'll assume that the accounting system isn't suffering from the extreme treatment that can occasionally result in very large tables holding very small amounts of data).

Alas, even when database-independent applications decide to use Oracle `date` types properly, they may still try to avoid `null` values. Rather than leaving any columns `null`, every nullable column is given a default value (usually through front-end code rather than using the relevant database feature). So what might the average database-independent developer choose as a good value to represent a `null` date? How about something far in the future, like 31 December 4000?

But remember how the optimizer calculates the selectivity of range scans. The queries that you know are obviously supposed to find 20% of the data (one year out of five) now appear to the optimizer to be much more selective than you think. Doing a quick calculation for the range scan selectivity, you believe the selectivity is

```
( 31 Dec 2003 - 01 Jan 2003 ) / (31 Dec 2004 - 01 Jan 2000)  + 2/1827 = 0.20044
```

But, given the extra, far-out, value, the optimizer thinks the selectivity is

```
( 31 Dec 2003 - 01 Jan 2003 ) / (31 Dec 4000 - 01 Jan 2000)  + 2/1828 = 0.00159
```

With this misleading value for the selectivity (which translates into a dramatically incorrect cardinality), it's not surprising if the optimizer manages to pick the wrong path through the data. And it takes just one row with this unfortunate default value to make the statistics look like rubbish.

The solution, again, is to create a histogram so that the optimizer can see an odd data distribution. The test script (see `defaults.sql` in the online code suite) creates a table with approximately 100 rows per day for the five years, but every thousandth row is set to the special value 31 December 4000.

```
create table t1
as
/*
with generator as (
        select  --+ materialize
                rownum   id
        from    all_objects
        where   rownum <= 2000
)
*/
select
        /*+ ordered use_nl(v2) */
        decode(
                mod(rownum - 1,1000),
                        0,to_date('31-Dec-4000'),
                          to_date('01-Jan-2000') + trunc((rownum - 1)/100)
        )  date_closed
from
        generator        v1,
        generator        v2
where
        rownum <= 1827 * 100
;
```

Check the cardinalities of the two execution plans generated for the following query. The first plan was the result of collecting simple statistics. The second appeared after generating a histogram of the default 75 buckets on the `date_closed` column:

```
select
        *
from    t1
where   date_closed between to_date('01-Jan-2003','dd-mon-yyyy')
                        and to_date('31-Dec-2003','dd-mon-yyyy')
;

Execution Plan (No histogram 9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=46 Card=291 Bytes=2328)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=46 Card=291 Bytes=2328)

Execution Plan (With histogram 9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=46 Card=36320 Bytes=290560)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=46 Card=36320 Bytes=290560)
```

Note how the second execution plan, with the histogram in place, shows a calculated cardinality that is very close to the 36,500 rows we would expect for a year's worth of data at 100 rows per day. The first execution plan, in contrast, has a cardinality that is far too small. In more complex queries, this type of error is likely to result in a ridiculous execution plan being produced.

Discrete Dangers

There are other problems with outlying values. The use of an extreme value to replace a `null` is not the only reason for a slightly odd data distribution that ends up causing bad execution plans. The same effect can also appear because people sometimes use an unusual value to represent "special events."

Consider, for example, an accounting system that has a `period` column—storing data for periods 1 to 12 plus a period (for **adjustments**) that is given a number of 99. The script `discrete_01.sql` in the online code suite emulates this (and also includes a second option, where the special period is number 13):

```
create table t1
as
with generator as (
        select  --+ materialize
                rownum         id
        from    all_objects
        where   rownum <= 1000
)
select
        /*+ ordered use_nl(v2) */
        mod(rownum-1,13)        period_01,
        mod(rownum-1,13)        period_02
from
        generator        v1,
        generator        v2
where
        rownum <= 13000
;

update t1 set
        period_01 = 99,
        period_02 = 13
where
        period_01 = 0;
;

commit;
```

As you can see, I have set up a table with two `period` columns. Both of these columns hold 13 distinct values, with 1,000 rows for each value. But one column uses 13 (the next possible integer) as its *adjustments* period, the other uses the far-out value of 99.

When we query the table for all the data in the second quarter (periods 4 to 6, say), we know that we really want to see 3,000 rows. We can work out from the standard formula that Oracle will predict the wrong cardinality because the formula is

```
num_rows * ( (our_high - our_low) / (table_high - table_low) + 2 / num_distinct) )
```

For the column period_01, using special value 99, this gives 13,000 * (2 / 98 + 2 / 13) = 2,265.306.

For the column period_02, using special value 13, this gives 13,000 * (2 / 12 + 2 / 13) = 4,166.667. But when we test the query against period_01, we get the following:

```
select  count(*)
from    t1
where   period_01 between 4 and 6
;

Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=5 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=5 Card=1663 Bytes=4989)
```

The cardinality predicted by Oracle for the column with period 99 is 1,663, not the 2,266 we expect from the standard formula. (The cardinality predicted by Oracle for the column with period 13 does, however, match the 4,167 given by the formula.) It gets worse, because look what happens with the four different quarters of the year in which we might be interested:

```
Period_01 between 1 and 3          Cardinality = 1,265      --
Period_01 between 4 and 6          Cardinality = 1,663      --
Period_01 between 7 and 9          Cardinality = 1,929      --
Period_01 between 10 and 12        Cardinality = 2,265      -- finally, one that's right!
```

It is possible that this change in cardinality could make our execution plans change, for no reason beyond the time of year we are running the query! (Assuming all other related data distributions stay the same, of course.)

You might try a cunning trick to work around this problem. What happens if you change the predicate to something that is equally valid—at least, from the perspective of your superior knowledge of the data? (See script discrete_02.sql in the inline code source.)

```
select  count(*)
from    t1
where   period_01 > 3 and period_01 < 7
;

Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=5 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=5 Card=735 Bytes=2205)
```

The predicate in this query will also return the 3,000 rows we want, but only because we have ensured that the values 4, 5, and 6 are the only values that can be found in the range. (I hope there's a constraint making sure that that's true.) But Oracle predicts a cardinality of 735 (and 4,333 for the column that uses 13 as the adjustments period). What does the formula say when we have open values (no equality at the ends):

```
num_rows * ( (our_high - our_low) / (table_high - table_low) )
```

This means we should see a cardinality of (14,000 * 4 / 98) = 531, not 735.

What we are seeing is a special case (with a built-in bug of its own), which I can summarize in Table 6-3. By running a loop that checks every viable 3-period query against our funny column, I can generate this table, showing the predicted cardinality and the difference between the current row and the previous row in the table for corresponding pairs of predicates. (This is from a 9.2.0.6 test—the figures from 8*i* are slightly different due to the usual rounding issues.)

Table 6-3. *Range-based Predicates with a Strange Decay Pattern*

Low	High	Between Low and High Cardinality (Change)	Low	High	> Low and < High Cardinality (Change)
–2	0	1,000			
–1	1	1,000	–2	2	1,000
0	2	1,133 (+133)	–1	3	1,000
1	3	1,265 (+132)	0	4	1,000
2	4	1,398 (+133)	1	5	1
3	5	1,531 (+133)	2	6	867 (–133 from 1000)
4	6	1,663 (+132)	3	7	735 (–132)
5	7	1,796 (+133)	4	8	602 (–133)
6	8	1,929 (+133)	5	9	531 (–71)
7	9	2,061 (+132)	6	10	531
8	10	2,194 (+133)			
9	11	2,265 (+71)			
89	91	2,265			
90	92	2,194 (–71)			
91	93	2,061 (–133)	90	94	531
92	94	1,929 (–132)	91	95	531 (+71)
93	95	1,796 (–133)	92	96	602 (+133)
94	96	1,663 (–133)	93	97	735 (+133)
95	97	1,531 (–133)	94	98	867 (+132)
96	98	1,398 (–133)	95	99	1

Table 6-3. *Range-based Predicates with a Strange Decay Pattern*

Low	High	Between Low and High Cardinality (Change)	Low	High	> Low and < High Cardinality (Change)
97	99	1,265 (–133)	96		1,000 (+133 from 867)
98	100	1,133 (–132)	97		1,000
99	101	1,000 (–133)			
100	102	1,000			

I have lined up the rows so that in any one row, the two sets of predicates have the same intent. For example, the line with the entries (4, 6, 1663, 3, 7, 735) represents the results from the following predicates:

```
where   period_01 between 4 and 6              -- card = 1663
where   period_01 > 3 and period_01 < 7        -- card = 735
```

I have only included rows where there is some variation in the cardinalities as the range of the predicates move along the original data set. Once you run a test and build a chart like this, some details stand out very clearly, which I can summarize as follows:

- For a large fraction of the range between the column low and the column high, the standard formula applies—we get 2,265 and 531 as the computed cardinality. (Of course, we know that neither answer matches the number of rows we will actually get, but the value is at least consistent with the standard formula.)
- As the predicate falls outside the range of the column values, then the cardinality falls back to `num_rows / num_distinct`. (13,000 / 13 = 1,000).
- The values in the *greater than* / *less than* columns for 1 through 5 and 95 through 99 (the two points where our requested ranges just touches the column low and high respectively) have to be the result of a piece of *special case* code, or a bug.
- All other values are a straight-line interpolation (adding roughly 133 per step) between the standard value and the boundary value of `num_rows / num_distinct`.
- The step in the cardinality (133) is `num_rows /` (column high – column low) = 13,000 / 98.

It's very satisfying to design a test that produces such a clear result so easily—but having got this far, there really isn't a lot of extra benefit in knowing exactly when the rules change or even if the standard formula is actually more complex than I originally suggested. Perhaps the standard formula includes a component that is very small in almost all cases, but becomes visible at the boundaries and only when dealing with a small number of distinct values. Who can guess what the rationale is for this odd-looking behavior.

I don't know why we get these results, but for my personal benefit, I now know that in cases where a critical column has a small number of distinct values and the range of the values is large compared to the number of distinct values, then a `between` or `greater than` / `less than` predicate on that column will behave badly—especially near the low and high values for the column.

The good news is this: if you create a histogram on the column in the test cases (see scripts `discrete_01a.sql` and `discrete_02a.sql` in the online code suite), then the problems disappear and Oracle gets the right answer every time.

10*g* Update

In Chapter 3, I mentioned the change that Oracle had introduced in 10.1.0.4 (very specifically) to deal with `column = constant` when the constant was known to be outside the low/high range for the column. The same effect has been introduced in range-based queries—and happened to show itself when I was rerunning the discrete tests.

In the results shown in Table 6-3, you saw that the computed cardinality reaches 1,000 and then stays constant as Oracle hits and passes the boundary of the low/high range in 9.2.0.6.

In 10.1.0.4, as we move the predicate further and further outside the low/high range for the column, the cardinality drops off by approximately 10 for each unit change in our predicate. Table 6-4 is a short extract from the top end of the range when we repeat the tests in `discrete_01.sql` against a 10.1.0.4 database.

Table 6-4. *Cardinalities in 10.1.0.4 Change Outside the Column Low/High*

Low	High	Between Low and High	Low	High	>Low and < High
96	98	1,398	95	99	1
97	99	1,265	96	100	1,000
98	100	1,133	97	101	1,000
99	101	1,000	98	102	1,000
100	102	990	99	103	1,000
101	103	980	100	104	990
102	104	969	101	105	980

The rate of change represents a straight line from a cardinality of 1,000 down to a cardinality of one—which will be reached when we are outside the low/high range by a distance equal to the low/high range. In other words, because (high – low) = 98, Oracle's model allows for the possibility of data as far out as (low – 98) and (high + 98), but the further you are from the known range, the less data you are likely to find.

Surprising sysdate

One of the biggest surprises you can get from simple selectivity calculations appears when you start using one of the most popular *pseudo-columns* in Oracle, the `sysdate`.

There are 1,440 minutes in a day, so if you run the following SQL, you will get a table holding four-and-one-half days' worth of minutes (see script `sysdate_01.sql` in the online code suite).

```
create table t1 as
```

```
select
        rownum                                  id,
        trunc(sysdate - 2) + (rownum-1)/1440    minutes,
        lpad(rownum,10)                         small_vc,
        rpad('x',100)                           padding
from
        all_objects
where
        rownum <= 6480
;
```

Clearly, then, queries with any of the predicates (a)–(d) in the following list should return 1,440 or 1,441 rows (one day plus the odd minute) and predicates (e)–(f) should return 2,880 or 2,881 rows (two days plus the odd minute).

```
a) where minutes between sysdate and sysdate + 1
b) where minutes between trunc(sysdate) and trunc(sysdate) + 1

c) where minutes between sysdate - 1 and sysdate
d) where minutes between trunc(sysdate) - 1 and trunc(sysdate)

e) where minutes between sysdate - 1 and sysdate + 1
f) where minutes between trunc(sysdate) - 1 and trunc(sysdate) + 1
```

So why, when you check the cardinality from `explain plan`, do you get the figures shown in Table 6-5 (which will vary with time of day) when you run the queries at 12:00 p.m. on a 9*i* system (with the usual sort of rounding differences creeping in an 8*i* system)?

Table 6-5. *Expressions Using sysdate Do Strange Things*

	Range	Cardinality (9.2.0.6) at Noon
a)	`Sysdate and sysdate + 1`	144
b)	`Trunc(sysdate) and trunc(sysdate) + 1`	180
c)	`Sysdate - 1 and sysdate`	180
d)	`Trunc(sysdate) - 1 and trunc(sysdate)`	144
e)	`Sysdate - 1 and sysdate + 1`	16
f)	`Trunc(sydate) - 1 and trunc(sysdate) + 1`	16

These results are very inaccurate; moreover they are not even self-consistent; for example, you would probably expect the cardinality of (e) to be pretty much the same as cardinality(a) + cardinality(c). The reason, though, is simple. The optimizer treats `sysdate` (and `trunc(sysdate)` and a few other functions of `sysdate`) as known constants at parse time, but `sysdate + N` becomes an unknown, and gets the same treatment as a bind variable—which means a fixed

5% selectivity. (Note, in particular, that `sysdate + 0` will give a different cardinality from `sysdate`.)

Moreover, having lost the ball on `sysdate + N`, the optimizer does its usual trick of turning the `between` into a combination of two independent predicates, so `minutes between sysdate - 1 and sysdate + 1` becomes

```
        minutes >= :bind1
and     minutes <= :bind2
```

This, of course, has the fixed selectivity of 0.25% that was used for predicates (e) and (f) earlier.

As a further demonstration, let's work out the arithmetic on example (a), which becomes

```
where   minutes >= sysdate
and     minutes <= {unknown bind value}
```

We had four-and-a-half days of minutes in the table, of which two days were future minutes (assuming you ran the script at exactly midday), so the selectivity of the first predicate (with its one closed end) is

```
(required range) / (total range) + 1 / number of distinct values =
(2 * 1440) / (4.5 * 1440) + 1/6480 =
0.4445987654321
```

- The selectivity of the second predicate is 0.05 (as a bind variable).
- The combined selectivity is 0.05 * 0.4445987654321 = 0.02222994.
- The cardinality is therefore 0.02222994 * 6480 = 144.05—which matches the displayed result.

So even when you store date information properly, it is possible for the optimizer to produce silly arithmetic about the selectivity and cardinality. And queries for things like "all of last week," "the last 24 hours," etc. must be some of the most popular forms of date query ever used. This is one place where you might find there is a significant benefit in using literal strings instead of bind variables or expressions such as `trunc(sysdate) - 7`.

And then there's a problem when you upgrade to 10*g*, because the problem has been identified and addressed, and all the examples in Table 6-5 supply the appropriate cardinality.

What's so bad about a problem being fixed? Ask yourself how much code you have got that is currently doing the right thing because the optimizer is underestimating the cardinality by a factor of 10 (examples (b) and (c)) or 80 (examples (e) and (f)). What's going to happen to that code when the optimizer suddenly gets the cardinality right? How many join orders are going to reverse themselves; how many indexed access paths are going to become tablescans; how many nested loops will turn into hash joins (and perhaps stop doing partition elimination on the way)? The side effects of an optimizer bug fix may need rigorous testing.

Function Figures

All the examples we've looked at so far have been focused on predicates on stored columns—but what happens if your SQL includes predicates like

```
upper(name) like 'SMITH%'
mod(number_col,10) = 0
pl_sql_func(last_name, first_name) =  'SMITH_JOHN'
```

Essentially, the arithmetic drops back to the fixed percentages that the optimizer uses for bind variables—with a couple of minor variations—as shown in Table 6-6. See scripts `like_test.sql` and `fun_sel.sql` in the online code suite.

Table 6-6. *Fixed Percentages Used for Selectivity on Character Expressions*

Example of Predicate	Treatment
`function(colx) = 'SMITH'`	Fixed 1% selectivity.
`not function(colx) = 'SMITH'`	Fixed 5% of selectivity.
`function(colx) > 'SMITH'`	Fixed 5% of selectivity.
`not function(colx) > 'SMITH'`	Fixed 5% of selectivity.
`function(colx) >= 'SMITH' and function(colx) < 'SMITI''`	Derived 0.25% selectivity (5% * 5%).
`function(colx) between 'SMITHA' and 'SMITHZ'`	Derived 0.25% selectivity (5% * 5%).
`not function(colx) between 'SMITHA' and 'SMITHZ'`	Derived 9.75% selectivity (5% + 5% – (5% * 5%)).
`function(colx) like 'SMITH%'`	Fixed 5% of selectivity—contrary to obvious interpretation, which would give 0.25%.
`not function(colx) like 'SMITH%'`	Fixed 5% of selectivity.
`function(colx) in ('SMITH','JONES')`	Derived 1.99% (1% + 1% – (1% * 1%)). Even in 10*g*, this function-driven in-list calculation uses the erroneous formula from 8*i*.

I have listed a few examples that are all about character comparisons, but the same calculations hold for numeric and date values where appropriate.

Bear in mind that if you create **function-based indexes**, you are actually creating indexes on *virtual columns*, and when you collect statistics on the table and its indexes, you can collect statistics on the *virtual columns* at the same time. In these circumstances, a supplied predicate such as

```
function(colx) = 0
```

is optimized as

```
SYS_NC00005$ = 0
```

And Oracle has statistics for the column named `SYS_NC00005$` (visible in view `user_tab_cols` from 9*i* onwards), so the normal selectivity arithmetic applies, and the preceding table is irrelevant. (It would be nice if Oracle Corp. added the option for unindexed *virtual columns*, so we could have the benefit of correct statistics for commonly used expressions, without the need to create the index.)

Correlated Columns

So far, we have been reviewing data that has been produced by the random number generator, and when we have indexed pairs of columns, they have been independent columns. We have avoided any risk of confusing the issue by using dependent (correlated) columns in our predicates, and this happens to be the way the optimizer assumes the world is always going to be. When you have data where there is some statistical relationship between two columns in the same table, odd results can appear.

To investigate the issue, we're going to start with the same data set that we used in the first test case of Chapter 4:

```
create table t1
as
select
        trunc(dbms_random.value(0,25))        n1,           -- 25 values
        rpad('x',40)                          ind_pad,
        trunc(dbms_random.value(0,20))        n2,           -- 20 values
        lpad(rownum,10,'0')                   small_vc,
        rpad('x',200)                         padding
from
        all_objects
where
        rownum  <= 10000
;
```

This gives us two columns, `n1` and `n2`, which between them probably have 500 different combinations of values. But before creating the index and statistics on this data set, run a simple update statement (the modified code can be found in the script `dependent.sql` in the online code suite):

```
update t1 set n2 = n1;
```

We now have a data set where there are only 25 different distinct keys in the index:

```
( 0, 'x                                   ', 0)
( 1, 'x                                   ', 1)
        ...
(24, 'x                                   ', 24)
```

The change to the data doesn't have much impact on the actual size of the index (which held 1,111 leaf blocks in the original test, but now holds 1,107 leaf blocks). With our knowledge of what we have actually done to the data, we can work out that a query such as the following will return 400 rows after hitting about 45 index leaf blocks (1/25 of the whole index) and an awful lot of the table. But look what `autotrace` says:

```
select
        /*+ index(t1) */
        small_vc
from
        t1
where
        ind_pad = rpad('x',40)
and     n1      = 2
and     n2      = 2
;

Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=14 Card=16 Bytes=928)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=14 Card=16 Bytes=928)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=4 Card=16)
```

According to the execution plan, this query will return 16 rows at a cost of 14 I/Os! Huge underestimates of cost and cardinality can obviously cause performance problems. Not only will this simple query be slower and more labor-intensive than the optimizer suggests, but if the optimizer is looking for a driving table in a multitable join, this table looks like a good choice to drive a nested loop because of the implication that the next stage of the query would only have to be run 16 times, rather than 400.

This type of error also works (or rather, causes problems) the other way round, of course—consider what happens when you change the `where` clause to the following:

```
where
        ind_pad = rpad('x',40)
and     n1      = 1
and     n2      = 3
```

Using our human insight into the data, we know that Oracle should be able to take just one quick (three-block) trip down the index to find out at runtime that there are no rows to return. But the optimizer still says the result set will be 16 rows at a cost of 14 I/Os.

Overestimates of cost and cardinality are just as bad as underestimates, because they may make the optimizer miss the best choice of driving table.

CORRELATED COLUMNS AND DYNAMIC SAMPLING

Correlation between columns in the same table always causes problems if those columns appear together in your `where` clause; the issue is not restricted to indexes. Sometimes you will be able to work around the problem by using the `optimizer_dynamic_sampling` parameter, or the `dynamic_sampling` hint (both of which appeared at some stage in 9*i*) to instruct Oracle to take a run-time sample of 32 or more blocks from critical tables to see what fraction of the rows matches your `where` clause.

One of the very strange things about this problem when it appears in indexes is that the database contains the right information (for some queries)—and the optimizer ignores it. If you look at the `user_indexes` view for this index, you will find the figures shown in Table 6-7.

Table 6-7. *Index Statistics from the Test Case in dependent.sql*

Statistics	Value
`Blevel`	2
`leaf_blocks`	1107
`distinct_keys`	25
`clustering_factor`	6153
`avg_leaf_blocks_per_key`	44
`avg_data_blocks_per_key`	246

When walking the index, Oracle examined the data in detail. It has actually recorded the information that there really are only 25 distinct keys. Moreover, the view also records the fact that when you execute a query for a single complete key, you will have to traverse approximately 44 leaf blocks to get the rowids for that key and then make approximately 246 table block visits to pick up the rows—if there is any data at all for that key.

We haven't looked at the `avg_leaf_blocks_per_key` or `avg_data_blocks_per_key` before, but for single-column indexes, and for multicolumn indexes where the columns aren't correlated, you will find that queries using an equality on all the indexed columns have a basic cost that is close to

```
blevel + avg_leaf_blocks_per_key + avg_data_blocks_per_key
```

This is easiest to appreciate on a single-column index, because

- The selectivity is 1 / (number of distinct keys).
- `avg_leaf_blocks_per_key` is calculated as `round(leaf_blks / distinct_keys)`.
- `avg_data_blocks_per_key` is calculated as `round(clustering_factor / distinct_keys)`.

The result you get by using this shortcut estimate instead of the proper calculation won't necessarily be completely accurate because (as the definitions show) the stored values seem to be calculated using the `round()` function, whereas the cost calculations use the `ceiling()` function; but it is often close enough to give you a clue about the likely utility of an index in your system.

When you get to Chapter 11, you will find that there is a case where the optimizer does use some of these stored figures, rather than the Wolfgang Breitling formula.

Dynamic Sampling

Let's try our query against our correlated columns but include the `dynamic_sampling()` hint in the SQL and see what happens to the execution plan (script `dependent.sql` again):

```
select
        /*+ index(t1) dynamic_sampling(t1 1) */
        small_vc
from
        t1
where
        ind_pad   = rpad('x',40)
and     n1        = 2
and     n2        = 2
;

Execution Plan (9i/10g) without dynamic sampling
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=14 Card=16 Bytes=928)
1  0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=14 Card=16 Bytes=928)
2  1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=4 Card=16)

Execution Plan (9.2.0.6) with dynamic sampling
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=14 Card=442 Bytes=25636)
1  0  TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=14 Card=442 Bytes=25636)
2  1    INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=4 Card=16)

Execution Plan (10.1.0.4) with dynamic sampling
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=288 Card=393 Bytes=22794)
1  0  TABLE ACCESS (BY INDEX ROWID) OF 'T1' (TABLE) (Cost=288 Card=393 Bytes=22794)
2  1    INDEX (RANGE SCAN) OF 'T1_I1' (INDEX) (Cost=46 Card=393)
```

Clearly, there are some differences in the cardinality estimates between 9*i* and 10*g*, but at least they both get much closer to the 400 that we expect to see. The more significant point in this example, though, is that 10*g* has managed to produce a suitable estimate for the cost as well.

This is what's going on. With the `dynamic_sampling(t1 1)` hint, we have told the optimizer to take a random sample of the data set, checking what fraction of the rows sampled meet the conditions of the query. If you generate a `10053 trace` file while this is going on, this is what you would find in the 9.2.0.6 trace file (after a little reformatting):

```
** Generated dynamic sampling query:
SELECT
        /*+ ALL_ROWS IGNORE_WHERE_CLAUSE */
        NVL(SUM(C1),0), NVL(SUM(C2),0)
FROM
        (
        SELECT /*+ IGNORE_WHERE_CLAUSE NOPARALLEL("T1") */
                1 AS C1,
                CASE
                     WHEN
                          "T1"."N1"=2
                      AND "T1"."IND_PAD"='x                                    '
                      AND "T1"."N2"=2
                      THEN 1
                      ELSE 0
                END AS C2
        FROM "T1" SAMPLE BLOCK (8.355795) "T1"
        ) SAMPLESUB

** Executed dynamic sampling query:
    level : 1
    sample pct. : 8.355795
    actual sample size : 837
    filtered sample card. : 37
    orig. card. : 10000
    block cnt. : 371
    max. sample block cnt. : 32
    sample block cnt. : 31
    min. sel. est. : 0.0016
** Using dynamic sel. est. : 0.04420550
```

The hint has been used to sample a specific table at level 1, which means 32 blocks. Oracle knows from the table statistics that the table size is 371 blocks, which means the sample size ought to be (100 * 32 / 371)%: unfortunately, this comes to 8.625337%, and we can see that Oracle has used 8.355795%—which represents 31 blocks, not the 32 blocks indicated in the manuals (you'll notice that this 31 appears in the subsequent summary as `sample block cnt`).

You can see that the SQL in the in-line view selects two values for the outer select to count. The first is the value 1—so `sum(c1)` simply counts the rows in the sample. The second is a `case` statement that evaluates to 1 when a row matches our original predicate, and 0 when it doesn't—so that `sum(c2)` counts the rows that match our predicate.

In the summary page, we see that Oracle examined 837 rows and found 37 matches, a fraction of 0.04420550. Oracle has used this figure for the selectivity of our predicate (possibly after comparing it to the original theoretical selectivity shown here as the `min sel. est.`).

Unfortunately, although Oracle has used this selectivity to calculate the cardinality (10,000 * 0.04420550 = 442), it hasn't used it to calculate the costs.

The behavior of 10*g* is very similar, although the summary of results is a little more extensive. The output is included in the script `dependent.sql`, but the critical figure is the `dynamic`

sel. est. of 0.03928171. This leads to a cardinality of 393 (multiply by 10,000 and round), and can then be used to calculate the cost in the standard fashion:

```
cost =
        blevel +
        ceiling(leaf_blocks * effective index selectivity) +
        ceiling(clustering_factor * effective table selectivity) =

        2 +
        ceiling(1,107 * 0.03928171) +
        ceiling(6,153 * 0.03928171) =

        2 + 44 + 242 = 288
```

The only other thing of note is that 10.1.0.4 behaves differently from 10.1.0.2, which managed to produce the following with a level 2 sample:

```
** Dynamic sampling initial checks returning TRUE (level = 2).
** Dynamic sampling index access candidate : T1_I1

SELECT
        /* OPT_DYN_SAMP */
        /*+ ALL_ROWS NO_PARALLEL(SAMPLESUB) NO_PARALLEL_INDEX(SAMPLESUB) */
        NVL(SUM(C1),0), NVL(SUM(C2),0), NVL(SUM(C3),0)
FROM
        (
        SELECT /*+ NO_PARALLEL("T1") INDEX("T1" T1_I1) NO_PARALLEL_INDEX("T1") */
                1 AS C1, 1 AS C2, 1 AS C3
                FROM    "T1" "T1"
                WHERE   "T1"."N1"=2
                AND     "T1"."IND_PAD"='x                                       '
                AND     "T1"."N2"=2
                AND     ROWNUM <= 2500
        ) SAMPLESUB
```

This is an interesting approach, as it simply counts to check whether the index statistics are sound, and makes no effort to check the table. Possibly it has been superseded, possibly it has been enhanced but is no longer relevant to my test case. I suspect that there is still plenty of scope for further development of dynamic sampling (and surprises for people investigating it).

Optimizer Profiles

If you've used 10*g* at all, you've probably done some experiments with the SQL Tuning Advisor, and found that sometimes the advice is to *accept a profile*. The SQL Tuning Advisor can be instructed to take lots of time analyzing a statement, and work out how it can be made to run faster (a process I sometimes refer to as *offline optimization*). One possible step in the analytical process is a detailed statistical study of the actual data content, querying the base table, and testing partial joins with data sampling.

You shouldn't do this of course, but if you are advised to accept a profile and take a note of the tuning task ID that generated it, you can find out what goes into that profile by querying some of the wri$ tables. In particular, the following query may turn up some interesting results:

```
select  attr1
from    wri$_adv_rationale
where   task_id = &&m_task
;

ATTR1
--------------------------------------------------------------------------------
OPT_ESTIMATE(@"SEL$1", JOIN, ("T2"@"SEL$1", "T1"@"SEL$1"), SCALE_ROWS=15)
OPT_ESTIMATE(@"SEL$1", TABLE, "T2"@"SEL$1", SCALE_ROWS=200)
OPTIMIZER_FEATURES_ENABLE(default)

3 rows selected.
```

Profiles are just a set of stored hints that supply extra information to Oracle at optimization time. And, although you obviously should not do this, you can put hints like these into end-user SQL. In the following example, I have removed the double-quotes, and the query block references. (The example is not in the online suite.)

```
select
        /*+
                OPT_ESTIMATE(TABLE, T2, SCALE_ROWS=200)
                OPT_ESTIMATE(JOIN, (T2, T1), SCALE_ROWS=15)
        */
        count(t1.v1) ct_v1, count(t2.v1) ct_v2
from
        t1, t2
where
        t2.n2 = 15
and     t2.n1 = 15
and     t1.n2 = t2.n2 + 0
and     t1.n1 = t2.n1
;
```

The effect of these hints is to tell Oracle that the single table access path into T2 will return 200 times the number of rows indicated by the stored statistics, and that the join from T2 to T1 will return 15 times the number of rows expected. (Note, however, that these are undocumented, internal hints, and I think the way they are used is still subject to change, so I wouldn't use them in production code if I were you, though if you wanted to you could experiment with them from time to time.)

If you have problems with dependent columns, profiles are likely to help you by helping Oracle to acquire some intelligent information about the awkward data distributions and their likely impact.

■ **Caution** My experiments with the `opt_estimate` hint showed the effects changing across different versions of 10.1. Do not try using it as a hint on a production system until it is documented for public use.

Transitive Closure

One of the problems you can have with the optimizer is that the code is sometimes too clever. When an execution plan seems unreasonable and you are busy trying to work out where the selectivity went wrong, remember that there may be some predicates floating around that you didn't write and can't see (unless you make proper use of the latest version of `explain plan`).

The optimizer may have used a mechanism known as **transitive closure** to generate a few predicates that will only show up if you use a proper tool to display the full execution plan.

Transitive closure works by logical inference. Assume you have two predicates (see script `trans_close_01.sql` in the online code suite):

```
        n1 = 100
and     n2 = n1
```

then the optimizer will be able to create the predicate

```
        n2 = 100
```

and include that in its calculations. But there is a catch—as it introduces the predicate with the constant, the optimizer may be allowed to eliminate the predicate without the constant, so that the final `where` clause looks like the following:

```
        n1 = 100
and     n2 = 100
```

This can be useful in some cases, but it can have some strange side effects. Consider this example (script `trans_close_02.sql` in the online code suite):

```
create table t1
as
select
        mod(rownum,10)              n1,
        mod(rownum,10)              n2,
        to_char(rownum)             small_vc,
        rpad('x',100)               padding
from
        all_objects
where
        rownum <= 1000
;
```

```
create table t2 as select * from t1;

--        Collect statistics using dbms_stats here

select
        count(*)
from
        t1, t2
where
        t1.n1 = 5
and     t2.n1 = t1.n1
;
```

The definitions of `t1` and `t2` are identical. They both have 1,000 rows, and the column `n1` is defined to hold 100 copies each of the values 1 to 10. When we run the query, we should join 100 rows in `t1` to 100 rows in `t2` for an output count of 10,000 rows. This is the execution plan we get from 9.2.0.6 using `dbms_xplan.display()`:

```
-------------------------------------------------------------------------
| Id  | Operation             |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |             |     1 |     6 |   404 |
|   1 |  SORT AGGREGATE       |             |     1 |     6 |       |
|   2 |   MERGE JOIN CARTESIAN|             | 10000 | 60000 |   404 |
|*  3 |    TABLE ACCESS FULL  | T1          |   100 |   300 |     4 |
|   4 |    BUFFER SORT        |             |   100 |   300 |   400 |
|*  5 |     TABLE ACCESS FULL | T2          |   100 |   300 |     4 |
-------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("T1"."N1"=5)
   5 - filter("T2"."N1"=5)
```

The cardinality is correct, but look at the cost. Notice the `merge join Cartesian` that appears in line 2, and the way that the `buffer sort` in line 4 has acquired a cost of 400 (which looks remarkably like the cost of the `t2` tablescan 100 times—once for each row in `t1`). Checking the predicate information, you find no predicate joining the two tables—the join predicate disappears as the second constant predicate is created.

But if you can interfere with the closure algorithm in a suitable way, the plan changes to the following (note how the cost seems reasonable, but the cardinality [with heading "Rows"] is now much too low):

```
-------------------------------------------------------------------------
| Id  | Operation            |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |     1 |     6 |     9 |
|   1 |  SORT AGGREGATE      |             |     1 |     6 |       |
|*  2 |   HASH JOIN          |             |  1000 |  6000 |     9 |
|*  3 |    TABLE ACCESS FULL | T1          |   100 |   300 |     4 |
|*  4 |    TABLE ACCESS FULL | T2          |   100 |   300 |     4 |
-------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("T2"."N1"="T1"."N1")
   3 - filter("T1"."N1"=5)
   4 - filter("T2"."N1"=5)
```

There are two ways to achieve this effect:

- Add the predicate `t2.n1 = 5` explicitly.
- Add a duplicate of the predicate `t2.n1 = t1.n1`.

A third (undesirable) option is the good old *rule-based* trick—changing the join predicate to `t2.n1 = t1.n1 + 0`. Unfortunately, when you adopt this worst-practice approach, the optimizer gets the right join cardinality (10,000) and a sensible cost (9).

So be a little careful if you find joins that are producing strange plans and cardinalities; you may have to doctor them (and document the hacks in anticipation of the next upgrade) to con the optimizer into treating them sensibly.

Of course, it is easiest to consider transitive closure with equality signs all around, but the optimizer can be cleverer than that. In the more general case, the optimizer is able to infer the following: if `n1 operator constant` and `n2 = n1`, then `n2 operator constant`. For example: if `n1 < 10` and `n2 = n1`, then `n2 < 10`.

Similarly, if `col1 > col2` and `col2 > {constant K}`, then Oracle can infer the predicate `col1 > {constant K}`.

You can even have cases like this one: if `n1 > 10` and `n1 < 0`, then `0 > 10`, which is always false, and therefore can short-circuit an entire branch of an execution plan. The predicates involved can even be ones that have been generated from constraint definitions. (See script `trans_close_03.sql` in the online code suite for some examples.)

The investigation of transitive closure involving the more generalized predicates (such as `n1 < 10`) for the nonjoin highlights a nasty little anomaly. The mechanism is not self-consistent. Consider Table 6-8.

Table 6-8. *Transitive Closure Is Not 100% Consistent*

Predicates You Supply	Predicates Used by Oracle After Transitive Closure
`n1 = 5 and n1 = n2`	`n1 = 5 and n2 = 5`
`n1 < 5 and n1 = n2`	`n1 < 5 and n2 < 5 and n1 = n2`
`n1 between 4 and 6 and n1 = n2`	`n1 >= 4 and n1 <= 6 and n2 >= 4 and n2 <= 6 and n1 = n2`

Of course, it is necessary to preserve the join condition when the nonjoin condition is not a test for equality, or you could get the wrong results. It is also correct (in the general case) to eliminate the join condition on equality—at least as far as calculating the cardinality is concerned, even though this may wreck the options for good execution plans. In 10*g* release 2, you will find a new hidden parameter that allows you to specify whether or not the join condition survives—the default is to allow it to survive, and once again the details of the cardinality calculation have changed.

Help (or trouble) is at hand, though in the shape of an undocumented effect of an inappropriate parameter. You may recall the parameter `query_rewrite_enabled` from the days when you needed to set it to `true` to allow function-based indexes to work. In recent versions of Oracle, this parameter is no longer relevant to function-based indexes. However, if you do set `query_rewrite_enabled` to true, the rules for transitive closure change—until you get to 10*g*.

Rerun my query from `trans_close_02.sql` in 8*i* or 9*i* with this parameter set to `true`, and the join predicate does not disappear. The plan, costs, and cardinality you get match the plan from the first two hacks I suggested (which means, of course, that the estimated cardinality is wrong).

Constraint-Generated Predicates

Here's a little puzzle—I have a table with the column `ename` that is declared as `varchar2(30)`. I create (and collect stats on) a simple (no functions involved) B-tree index on that column:

```
create index t1_i1 on t1(ename);
```

A user executes the following query:

```
select * from t1 where upper(ename) = 'SMITH';
```

Is it possible for the optimizer to use the simple index efficiently (with a precisely targeted range scan) to acquire the correct data?

The answer is yes—but only from 9*i* onwards, and only within certain limits (see script `constraint_01.sql` in the online code suite).

```
create table t1 (
        id      number,
        v1      varchar2(40) not null,
        constraint t1_ck_v1 check (v1=upper(v1))
);
```

```
begin
        dbms_random.seed(0);
        for n in 1..10000 loop
                insert into t1 (id, v1) values (n, dbms_random.string('U', 30));
        end loop;
end;
/

create index t1_i1 on t1(v1);                          -- not an FBI

--        Collect statistics using dbms_stats here

select
        *
from
        t1
where
        upper(v1) = 'SMITH'
;
```

The secret is in the constraint(s). The optimizer has loaded the constraints into memory, applied them to the query, and then used transitive closure to come up with new predicates. Hence

```
v1 = upper(v1)                  -- the constraint
upper(v1) = 'SMITH'             -- the actual predicate
```

imply

```
where   v1 = 'SMITH'            -- closure, and the index can be used.
```

As with the ordinary examples of transitive closure from the previous section, you can see these generated predicates in the full execution plan or from `dbms_xplan`.

There are a few little quirks and bugs with this feature. In earlier versions of both 9*i* and 10*g*, the mechanism could produce wrong results in some special cases, but I think the errors have been fixed in 9.2.0.6 and 10.1.0.4. There is still an outstanding limitation with 10*g* (that is not present in 9*i*)—the test case in script `constraint_01.sql`, for example, does not use the index for the following type of predicate:

```
where  upper(v1) = :bind_var
```

Of course, it is just possible that the 9*i* behavior is a bug, and the 10*g* behavior is the bug fix. (I can't think why this would be in this case, but sometimes a feature disappears because it isn't logically safe to implement—side-effects due to the risk of `null` values often fall into this category).

The issue with wrong results revolved around check constraints that included built-in functions that could return a `null` value when given a nonnull column value. You will notice that my column declaration included the `not null` constraint (strangely, it has to be declared at the column level, not as a table check constraint). If you fail to do this, there are some classes of constraint for which the predicate closure mechanism simply will not work, although you

can force it into play by including an explicit `is not null` clause in your query. You will also find that this mechanism will not be invoked if the constraints are declared to be deferrable.

The mechanism is so clever that it can have the slightly surprising effect of putting the generated predicates somewhere you don't expect them to be. Consider this example (see script `constraint_02.sql` in the online code suite):

```
create table t1 as
select
        trunc((rownum-1)/15)    n1,
        trunc((rownum-1)/15)    n2,
        rpad(rownum,215)        v1
from
        all_objects
where
        rownum <= 3000
;

create table t2 as
select
        mod(rownum,200)         n1,
        mod(rownum,200)         n2,
        rpad(rownum,215)        v1
from
        all_objects
where
        rownum <= 3000
;

create index t_i1 on t1(n1);
create index t_i2 on t2(n1);

alter table t2 add constraint t2_ck_n1 check (n1 between 0 and 199);

--        Collect statistics using dbms_stats here

select
        count(t1.v1)        ct_v1,
        count(t2.v1)        ct_v2
from
             t1, t2
where
          t2.n2 = 15
and       t1.n2 = t2.n2
and       t1.n1 = t2.n1
;
```

We might predict that the optimizer could combine the first two predicates in this query to generate the predicate `t1.n2 = 15` (losing the predicate `t1.n2 = t2.n2` as it did so). But look what else we get when we run this query through `dbms_xplan`.

```
-------------------------------------------------------------------------
| Id  | Operation            |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |     1 |   444 |    33 |
|   1 |  SORT AGGREGATE      |             |     1 |   444 |       |
|*  2 |   HASH JOIN          |             |    15 |  6660 |    33 |
|*  3 |    TABLE ACCESS FULL | T1          |    15 |  3330 |    16 |
|*  4 |    TABLE ACCESS FULL | T2          |    15 |  3330 |    16 |
-------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("T1"."N1"="T2"."N1")
   3 - filter("T1"."N2"=15 AND "T1"."N1">=0 AND "T1"."N1"<=199)
   4 - filter("T2"."N2"=15)
```

We have indeed lost one join predicate and gained another constant predicate, but look carefully at the filter predicate for line 3: it contains the range-based check that was our constraint check from table `t2`—but the check is made against table `t1`. Because there is an equality between `t1.n1` and `t2.n1`, the optimizer can see that the only rows that could be joined from `t1` must conform to the constraint check on `t2`, so it has migrated the text of the constraint into a predicate against `t1`, as this may allow it to do a more accurate calculation of join cardinality. (If table `t1` already had the same constraint in place, this predicate generation would not occur.)

CONSTRAINTS, PREDICATES, AND DYNAMIC SAMPLING

I first came across this example of a constraint on one table becoming a predicate on another while I was preparing a presentation on dynamic sampling. Once the parameter `optimizer_dynamic_sampling` is set to 4 or more, then the optimizer will request a sample of any table with two or more *single-table* predicates against it before generating the full execution plan.

When I was first testing the effects of dynamic sampling, I wrote a query with two tables, each having just one *single-table* predicate—and Oracle surprised me by sampling one of the tables. Transitive closure had added two extra predicates to one of the tables. Oracle is often like that—when you're busy looking at one feature, another feature sneaks in to confuse the issue.

You can expect the mechanism to become increasingly sophisticated as time passes. The two examples shown previously demonstrated cases where there was a simple constraint on a single column—for a slightly more sophisticated example, look at `constraint_03.sql` in the online code suite, which is an example of Oracle using a table-level constraint of the form `check (n2 >= n1)` to create a constraint that allows a query to change its execution plan from a tablescan to an indexed access path.

Summary

There are good reasons why the optimizer can do some surprising things when calculating the cardinality. If you know why things can go wrong, you have some chance of fixing these problems sensibly.

Critical points are as follows: the optimizer does not do very well with range scans on character strings. If you get a highly skewed distribution on the first six or seven characters of a column, then the optimizer is likely to get a poor estimate of selectivity (and cardinality) quite frequently. There is no quick and easy workaround to this, but a histogram with a lot of buckets can help.

Storing data in the wrong data type—particularly storing dates in numeric or character columns, or turning sequence numbers into zero-padded character strings—is likely to cause problems. The optimizer may get more accurate estimates of selectivity if you can create histograms with a sufficiently large number of buckets. You may also be able to work around some problems by creating function-based indexes that restate the data in the correct type—if you can also change the SQL to match.

Applications that use special values instead of nulls in columns are liable to produce poor execution plans if the columns are used in range-based predicates. Again, a histogram on the critical column may make an enormous difference.

Applying functions to columns that do not participate in function-based indexes can lead to some unexpected selectivities. You may even find that restating the same clause in a different way can lead to a significant change in selectivity because of some of the inconsistencies of the fixed constants used for the selectivity in these cases.

Transitive closure can create and lose predicates for you, and generally the impact will be good. Occasionally, the side effects can be catastrophic, sometimes they may simply be surprising. Although people sometimes insist on avoiding **referential integrity** constraints on data warehouses because of the overheads, remember that constraints may help the optimizer find a better execution plan. And don't be surprised when Oracle picks up a constraint, turns it into a predicate, and then migrates it through transitive closure to a different table.

Test Cases

The files in the download for this chapter are shown in Table 6-9.

Table 6-9. *Chapter 6 Test Cases*

Script	Comments
`char_types.sql`	Simple script to show numeric representation of character strings
`nchar_types.sql`	Repeat of the preceding, using a multibyte character set
`date_oddity.sql`	Demonstration of cardinality problems due to incorrect data types
`char_fun.sql`	Creates a simple function to calculate the numeric value used by the optimizer to represent a string
`char_seq.sql`	Problems with zero-padded character strings used for sequence numbers

Table 6-9. *Chapter 6 Test Cases*

Script	Comments
defaults.sql	Demonstration of the impact of a silly default value instead of a null
discrete_01.sql	Demonstration of the effect of discrete values on a between clause
sysdate_01.sql	Demonstration of error in sysdate-related cardinality
like_test.sql	Character column like 'XXX%'
fun_sel.sql	Selectivity for function(colx)
dependent.sql	Demonstration of effects of dependent columns used in a predicate
trans_close_01.sql	Simple example of transitive closure
trans_close_02.sql	Transitive closure having an odd side effect—autotrace output
trans_close_03.sql	Transitive closer with ">"—autotrace output
constraint_01.sql	Uses constraints to generate useful predicates
constraint_02.sql	A surprise from constraint-based predicates
constraint_03.sql	Demonstrates that even table-level constraints between columns can help
discrete_02.sql	Demonstration of the effect of discrete values on *greater than*/*less than*
discrete_01a.sql	Repeats discrete_01.sql, using a histogram to fix the problem
discrete_02a.sql	Repeats discrete_02.sql, using a histogram to fix the problem.
char_value.sql	Creates a function to return the numeric value equivalent to an input varchar2()
trans_close_02a.sql	As trans_close_02.sql using dbms_xplan
trans_close_03a.sql	As trans_close_03.sql using dbms_xplan
setenv.sql	Sets the standardized test environment for SQL*Plus

CHAPTER 7

Histograms

A lot of misinformation exists on the Internet about **histograms**: what they do, what they look like, how they work, and when they don't work. There are also plenty of gaps in the available information.

In this chapter, I hope to fill in some of those gaps and correct some of the misunderstandings. Unfortunately, I shall never know how successful I have been in this aim, because I recently came across bug 2757360 on MetaLink, dated Jan 2003, which included the wonderful line: "*There are any number of bugs on the CBO and histogram behavior ...* "

You could ask many questions about histograms, but the important ones are probably these: what are histograms; how does Oracle use them; when does Oracle use them; when does Oracle ignore them; what problems do they solve; what problems do they introduce; and how do you build a useful histogram if Oracle won't do it for you? The answers to all these questions are in this chapter.

Getting Started

Everyone *knows* that histograms are supposed to be used for data where a column has a few special values that appear far more frequently than the rest; and everyone *knows* that you should only build histograms on indexed columns. These beliefs are so misleading that I'm going to ignore them for the moment and start my explanation of histograms with an example that has no extreme values and no indexes. The test script in the online code suite is called `hist_intro.sql`, and as usual, my demonstration environment starts with an 8KB block size, locally managed tablespaces with a 1MB extent, manual segment space management, and system statistics (CPU costing) disabled.

```
execute dbms_random.seed(0)

create table t1
as
with kilo_row as (
        select  /*+ materialize */
                rownum
        from    all_objects
        where   rownum <= 1000
)
```

```
select
        trunc(7000 * dbms_random.normal)        normal
from
        kilo_row        k1,
        kilo_row        k2
where
        rownum <= 1000000
;
```

You will notice that this statement uses the **subquery factoring** mechanism introduced in 9*i*, which means the code won't run under 8*i*. The result is a table holding 1,000,000 random values in a **normal distribution**. I've used the `dbms_random` package to generate the data, and the first line of code, the `seed()` call, is very important for reproducible examples.

This test case should produce a total of 42,117 different values, ranging from –32,003 to 34,660. Since there are 1,000,000 rows, there will be (on average) 24 rows per recorded value. Of course, if we happen to check a few specific values, we will find that this average could be quite misleading. For example, three rows have the value –18,000; only one row has the value +18,000; and for the value 0 we find 109 rows.

Of course, if you are familiar with the famous **bell curve** of the normal distribution, you won't be surprised at the variations in these results. To highlight this variation, we could collect and graph the results from this query:

```
select normal, count(*) ct from t1 group by normal;
```

If we did this, the graph would look something like the approximation shown in Figure 7-1.

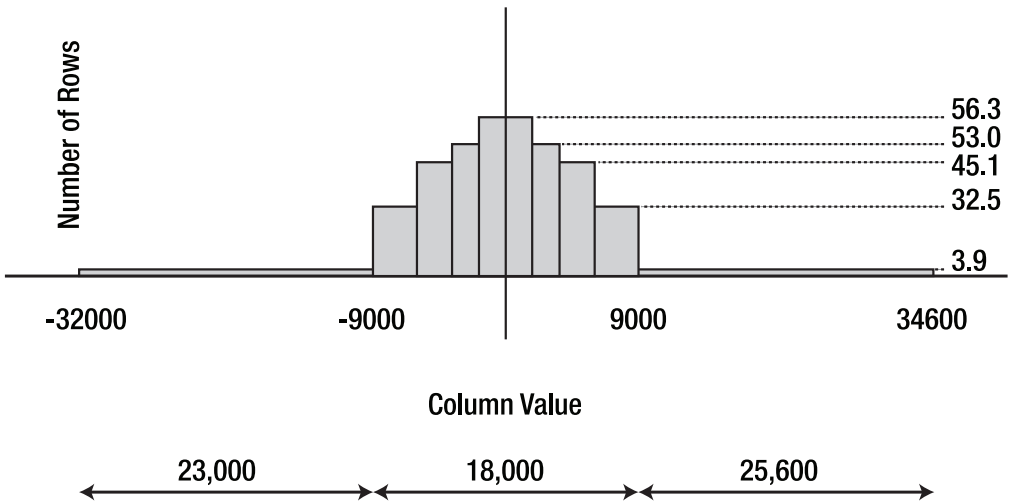

Figure 7-1. *Graphing the data distribution*

Rather than plotting 42,000 distinct points, I've sorted my data set into order, split it into ten blocks of 100,000 rows (anyone who is familiar with quartiles and percentiles will recognize the technique), and then drawn a bar for each of the ten blocks (or **deciles**, as they're called). This can be done with a simple piece of SQL, which I show here in two stages:

```
select
        normal,
        ntile(10) over (order by normal) tenth
from t1
;
```

In the first step, I use the `over()` clause of the **analytic function** `ntile()` to sort the data into order and break the ordered list into ten evenly sized sections—which will be 100,000 rows each. The `ntile()` function extends each row by adding a new column that holds the number of the section that the row belongs to. I have used the alias `tenth` for this new column. If I wanted a more accurate picture of my data, I could simply have increased the value in the `ntile()` function to something larger than 10.

Having worked out how to use the `ntile()` function to sort and section my data, the second step of the analysis wraps the initial query in an **in-line view** and generates the low and high values for each section, from which I can derive the width and height of the rectangles in my graph.

```
select
        tenth                                           tenth,
        min(normal)                                     low_val,
        max(normal)                                     high_val,
        max(normal) - min(normal)                       width,
        round(100000 / (max(normal) - min(normal)),2)   height
from (
        select
                normal,
                ntile(10) over (order by normal) tenth
        from t1
)
group by tenth
order by tenth
;
```

The width of each bar is given by the boundaries of the deciles; the height of each bar is (100,000 / width), which has the effect of averaging the row distribution across the available range in the decile and ensuring that all the bars have the same **area**. The results are as follows:

```
     TENTH    LOW_VAL   HIGH_VAL      WIDTH     HEIGHT
---------- ---------- ---------- ---------- ----------
         1     -32003      -8966      23037       4.34
         2      -8966      -5883       3083      32.44
         3      -5883      -3659       2224      44.96
         4      -3659      -1761       1898      52.69
         5      -1761         17       1778      56.24
         6         17       1792       1775      56.34
         7       1792       3678       1886      53.02
         8       3678       5897       2219      45.07
         9       5897       8974       3077      32.50
        10       8974      34660      25686       3.89
```

Pick any value between –32,003 and –8,966 (the 1st tenth) and the height of that bar tells you that there won't be many matching rows in the table (the height is about 4). Similarly, there won't be many rows for any value between 8,974 and 34,660 (the 10th tenth). Most of our data is clustered in the middle section of the graph. In fact, 80% of the data (eight bars out of ten) is packed into just 27% (from –8,966 to +8,974) of the total range of values.

Now, here's an important question. If this were your production data set and you were querying this data frequently, would you want Oracle to behave as if the queries were always aimed at the extreme ranges or at the packed data set in the middle? Or would different users have different requirements that covered the entire range of possibilities? Bear in mind that that's a business-related question, not a technology question.

If your business is only interested in the central range, then you will probably want Oracle to behave as if there were about 45 rows for any given value. If your business is always interested in the outside edges, then you want Oracle to behave as if there were about 4 rows for any given value. And what is Oracle supposed to do if your queries go all over the place?

Remember that it is important to calculate the correct cardinality at each stage of an execution plan, because the cardinality at any one point in the plan can affect join orders, join methods, and choice of indexes. But this data set shows that the *correct* estimate of the cardinality for a simple equality condition may depend more on the business purpose than on the raw data.

When the graph isn't a flat line, it is the business requirement that dictates which bit of the graph is the important bit, hence what the "correct" cardinality should be. Any time you draw a picture of your data and find that it has peaks, or gaps, or anything other than a flat, continuous profile, you may have problems getting the optimizer to work out a suitable cardinality for most of your queries. And, as we know, inappropriate estimates of cardinality lead to unsuitable execution plans.

Before I end this section, I will be giving you an alternative method for getting the figures you would need to draw my graph. But for the moment, it's time to get back on track with histograms. Let's start by creating a histogram of ten bars (or **buckets** as Oracle tends to call them) on the column `normal`:

```
begin
        dbms_stats.gather_table_stats(
                user,
                't1',
                cascade => true,
                estimate_percent => null,
                method_opt => 'for columns normal size 10'
        );
end;
/
```

Then query the view `user_tab_histograms` for the 11 values stored there. (To draw N bars, we need N + 1 endpoints.)

```
select
        rownum                                  tenth,
        prev                                    low_val,
        curr                                    high_val,
        curr - prev                             width,
        round(100000 / (curr - prev) , 2)       height
from
        (
        select
                endpoint_value                          curr,
                lag(endpoint_value,1) over (
                        order by endpoint_number
                )                                       prev
        from
                user_tab_histograms
        where
                table_name = 'T1'
        and     column_name = 'NORMAL'
        )
where
        prev is not null
order by
        curr
;
```

Again, I've used an analytic function, this time the `lag()` function, that allows us to transfer values from previous rows into the current row if we can supply a suitable ordering in the `over()` clause. In this case, I've lagged the data by one row so that I can find the difference (`curr - prev`) between adjacent rows in the `user_tab_histograms` view.

With the values of (`curr - prev`) available, I can complete my SQL statement—and when I run it, it produces exactly the same values I got by running my original query against the raw data set. I told you that I was going to give you another way of getting the figures to draw my graph—this is it, a nice, quick query against view `user_tab_histograms`.

Isn't that an amazing coincidence? No, not really, because from 9*i* onward, you can enable **SQL trace** while using the `dbms_stats` package to generate a histogram, and find that behind the scenes, the package is running SQL like the following:

```
select
        min(minbkt),
        maxbkt,
        substrb(dump(min(val),16,0,32),1,120)   minval,
        substrb(dump(max(val),16,0,32),1,120)   maxval,
        sum(rep)                                sumrep,
        sum(repsq)                              sumrepsq,
        max(rep)                                maxrep,
        count(*)                                bktndv,
        sum(case when rep=1 then 1 else 0 end)  unqrep
from
        (
        select
                val,
                min(bkt)                minbkt,
                max(bkt)                maxbkt,
                count(val)              rep,
                count(val) * count(val)    repsq
        from
                (
                select  /*+
                            cursor_sharing_exact dynamic_sampling(0) no_monitoring
                        */
                        "NORMAL"                            val,
                        ntile(10) over(order by "NORMAL")     bkt
                from
                        "TEST_USER"."T1" t
                where
                        "NORMAL" is not null
                )
        group by val
        )
group by
        maxbkt
order by
        maxbkt
;
```

Look very carefully at the innermost of the inline views. Note the line where the `ntile(10)` appears. Apart from a few cosmetic changes, and a few extra figures used for dealing with extreme conditions and the value of `user_tab_columns.density`, the SQL that generates the figures stored in `user_tab_histograms` is exactly the same as my original graph-drawing SQL. A histogram is just a picture of your data set.

Look back at the graph one more time, and say to yourself, "Oracle Corp. calls this type of graph a **height balanced histogram**." If you've ever had trouble understanding Oracle's concept of height balanced histograms, you now know why ... they aren't "height-balanced," they're just the straightforward style of histogram that most people probably first learned about at the age of 12, and have never had to think about since.

HEIGHT BALANCED HISTOGRAMS

When I searched Google for *height balanced* and *histogram*, every reference I found pointed me back to Oracle. It was only after a tip-off from Wolfgang Breitling that I did a search for *equi-depth* and *histogram* and found that the histograms that Oracle describes as height balanced are commonly called equi-depth (or sometimes equi-height) in the standard literature.

None of the terms gives me an intuitive understanding of how the graphs represent the data—so I tend to avoid using the expression *height balanced*, and just call the things histograms.

Generic Histograms

Oracle uses histograms to improve its selectivity and cardinality calculations for nonuniform data distributions. But you can actually use two different strategies: one for data sets with only a few (fewer than 255) distinct values, and one for data sets with lots of distinct values. Oracle calls the former a **frequency histogram** (although technically it probably ought to be a **cumulative frequency histogram**) and the latter a **height balanced histogram** (although, as you saw earlier, the heights involved are not balanced in any way that a non-mathematician would appreciate).

Although the two types of histogram have their own special features, there are many common areas in their use that I plan to cover in this section. At the simplest level, irrespective of type, a histogram is simply a collection of pairs of numbers (stored in views like `user_tab_histograms`, `user_part_histograms`, `user_subpart_histograms`) that can be used to draw pictures of the data. However, while collecting the histogram data, Oracle also collects some extra information that it uses to calculate a modified `density` for the column. After generating a histogram for a column, you will usually find that the `density` value reported in `user_tab_columns` (et al.) is no longer equal to `1 / num_distinct`.

When working out selectivities and cardinalities, Oracle may be able to make use of the full detail of the histogram, but sometimes has to fall back on using just the `density`. It is this fallback position that can cause surprises, so the rest of this section is a quick tour of features that interfere with the best use of histograms.

Histograms and Bind Variables

We have seen in earlier chapters that the optimizer uses the `density` as the selectivity for predicates of the form `column = constant` or `column = :bind_variable`. So with histograms in place, the selectivity of a simple equality predicate changes, even for a query involving a bind variable. If you've ever heard that histograms become irrelevant when you use (*un-peeked*) bind variables, it's not quite true—the detail cannot be used, but the effect of the `density` may be very relevant. Unfortunately, as I noted earlier, if there is a particular area of the graph that you are really interested in, you may need to override Oracle's estimated `density` and create a *business-relevant* `density`.

Bind Variable Peeking

Of course, things got messier when 9*i* introduced **bind variable peeking**. Whenever a statement is optimized, Oracle will (in almost all cases) check the actual values of any bind variables and optimize the statement for those specific values.

PARSING AND OPTIMIZING

When an SQL statement is first executed, it has to be checked for syntax, interpreted, and optimized. Thereafter if the same piece of text is fired at the database again, it may be recognized as *previously used*, in which case the existing execution plan may be tracked down and reused.

However, even when a statement is still in memory, some of the information about the execution plan may become invalid, or may get pushed out of memory by the standard memory management **LRU** routines. When this happens, the statement will have to be reoptimized. (You can detect this from the `loads` and `invalidations` columns in `v$sql`, summarized in the `reloads` and `invalidations` columns of `v$librarycache`. Reloads occur when information is lost from memory; invalidations occur when some of the dependent information changes.)

The reuse (or sharing) of SQL is generally a good thing—but if the execution plan generated on the first use of a statement caters to an *unlucky* set of values, then every subsequent execution of that statement will follow the same *unlucky* execution plan until you can force the statement out of memory—perhaps by the extreme method of flushing the *shared pool*.

Bind variable peeking has the unfortunate side effect of making it very easy for one user to introduce an execution plan that is bad news for every other user for an arbitrary amount of time, until that execution plan happens to get flushed from the shared pool for some reason.

This works quite well for **OLTP** systems, of course, as OLTP systems tend to use the same small number of high-precision queries extremely frequently, and it is quite likely that every execution of a given statement should employ the same plan. It is possible, though, that the first time a statement is optimized, an unusual set of values was passed in through the bind variables. If this happens, the optimizer might find a path that is good for that set of values, but very bad for the more popular use of the statement.

cursor_sharing

Another feature relating to bind variables is the use of the `cursor_sharing` parameter. The cost of parsing and optimizing is so high, and reduces scalability so much, that Oracle Corp. created the `cursor_sharing` parameter to allow developers a backdoor into sharable cursors.

When the parameter `cursor_sharing` is set to the value `force`, the optimizer will replace (almost) any literal constants in your SQL (and some of your PL/SQL) with system-generated bind variables, and check to see whether there is a previously generated sharable cursor that could be used for this modified statement. Because of the switch to bind variables, `cursor_sharing` has a big impact on the effectiveness of histograms. (Of course, after converting the literal constants to bind variables, 9*i* then peeks at the actual values if it has to optimize the statement.)

CURSOR_SHARING AND PL/SQL

In most cases, Oracle will replace literal values with bind variables with names like :SYS_B_0; however, if you write code that calls anonymous PL/SQL blocks using the old `begin proc_name (...)end;` syntax, then Oracle will not perform bind variable substitution. Since PL/SQL blocks don't need to be optimized (even though they still need to have cursor areas prepared for them), the overhead caused by this deficiency in the mechanism may be something you can put up with.

If you can convert your code to the newer `call proc_name(...)` syntax, then Oracle will do bind variable substitution. Note, however, that some older versions of Oracle have a bug that causes sessions to crash if you mix literals and real bind variables in a call, and then enable `cursor_sharing`.

Two workarounds appeared in 9*i* to deal with the traps introduced by `cursor_sharing=force`. The easy workaround is the hint `/*+ cursor_sharing_exact */`, which can be added to a statement to tell Oracle that the statement should not have its literal constants replaced by bind variables.

The more subtle and dangerous workaround is to use the option `cursor_sharing=similar`. With this value for `cursor_sharing`, Oracle will first replace literal constants with bind variables, and then decide to peek at the bind variables so that it can optimize for the incoming values on *every single parse call* for the statement if it seems to be a good idea.

The comments about this feature in the 9.2 manuals say that Oracle will reoptimize if *the values of the variables would make a difference to the execution plan*. It seems that two things will trigger this reoptimization: first, if any of the predicates involves a range scan, and second, even on a simple equality, if there is histogram on a column that appears in a predicate, the query will be reoptimized. (See script `similar.sql` in the online code suite for an example showing this.)

When this happens, the resources needed for optimization increase, as does the contention, because Oracle rewrites the query with bind variables, decides it should not be sharable, and inserts it into the library cache as a new child cursor in `v$sql` (where lots of copies of the same, substituted text will presumably be accumulating under the same **latch**).

The moral of this story is that if you really think you have to set `cursor_sharing=similar`, make sure you don't create more histograms than you absolutely need to, or you may introduce more performance problems than you solve. (In fact, you should always avoid creating histograms that you don't really need—it's just that this setting for `cursor_sharing` really exacerbates the problem.)

OUTLINES, EXPLAIN PLAN, AND CURSOR_SHARING

There is a little trap waiting for you if you create **stored outlines** or make use of `explain plan` while `cursor_sharing` is set to `force` or `similar`. When you `explain plan for` {sql statement}, or `create outline for` {sql statement}, Oracle does not replace constants with bind variables. Instead it creates an execution plan based on the actual values supplied.

The effect of this is that the execution plan you see is not necessarily the plan that would be used if you actually run the query. Moreover, in the case of stored outlines, the SQL with the actual values is stored in table `outln.ol$`. If you choose to run the original statement, it will be rewritten at run time to include bind variables. This means the text of the run-time SQL won't match the text of the SQL stored in `outln.ol$`, so the stored outline won't even be invoked for the SQL that (apparently) generated it. This may be why some notes on MetaLink once claimed that stored outlines could not work with `cursor_sharing` enabled.

A similar issue applies to the initial releases of 10*g* when you create a `profile`—but I understand that this issue has been addressed in 10*g* release 2.

When Oracle Ignores Histograms

It is quite expensive to generate histograms, so you should think very carefully about doing it. Oracle doesn't use them very much, and often won't get any benefit from them. In fact, there are several cases where Oracle ignores histograms when you might think they would be used.

Histograms and Joins

Oracle only uses histograms fully when there are highly visible input values—in other words, in cases where you have predicates of the form `column operator constant` (although the constant might come from peeking at a bind variable). Think about this carefully, and you realize that Oracle may not be able to make much use of histogram information to help it optimize joins (but see Chapter 10). Consider this simple query:

```
select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t1.n2 = 99
and     t1.n1 = t2.n1
;
```

Even if you have histograms on columns `t1.n1` and `t2.n1`, how can Oracle have any idea about the range and frequency of values of the join columns for rows where `t1.n2` is 99?

On the other hand, remember that creating a histogram on a column will affect the recorded density of that column; so if calculations of join selectivity or cardinality use the density, then histograms on join columns will have some effect. Unfortunately, as you will see in Chapter 10, the calculation of join selectivity seems to use the `num_distinct` in Oracle 8*i* and Oracle 9*i*, and only switches to the density in Oracle 10*g*—which means you could have some nasty surprises when you upgrade even when *nothing has changed*!

JOIN HISTOGRAMS

I have seen a couple of papers suggesting that it is a good idea to create histograms on the columns at the ends of **PK/FK** relations. Unfortunately, the papers did not include any justification for this suggestion, nor did they offer an explanation of the issue that this was supposed to address. In fact, PK/FK joins are the one place where the optimizer seems to be really good at calculating the correct join cardinality unaided—making histograms unnecessary.

It is possible that the authors were considering the special case of joins where the primary key of the parent table appeared in a predicate with a literal constant (which, as you saw in Chapter 6, would be forwarded to the foreign key columns on the child table by transitive closure).

In some cases, though, histograms do make a significant difference (as you will see in Chapter 10); but only under certain conditions that PK/FK joins aren't likely to meet.

Histograms and Distributed Queries

Oracle keeps on getting better at handling **distributed queries**—but even in Oracle 10*g*, the optimizer doesn't try to pull histograms from the remote site(s) to improve its execution plan, and it's not just a question of ignoring histograms on join columns. Even when you have remote columns compared to literal constants, Oracle does not attempt to use the histogram. Again, it falls back to the `num_distinct` (or density, depending on version). For example, consider these two queries (which can be found in script `dist_hist.sql` in the online code suite):

```
select
        home.skew2,
        away.skew2,
        home.padding,
        away.padding
from
        t1                  home,
        t1@d920@loopback    away
where
        home.skew = 5
and     away.skew = 5
and     home.skew2 = away.skew2
;

select
        /*+ driving_site(away) */
        home.skew2,
        away.skew2,
        home.padding,
        away.padding
```

```
from
        t1                      home,
        t1@d920@loopback        away
where
        home.skew = 5
and     away.skew = 5
and     home.skew2 = away.skew2
;
```

In this example, I have created a **loopback database link** to help me emulate a distributed query, and then joined a table to itself. The column `skew` that appears with the predicate `skew = 5` has a very skewed data distribution, and has had a histogram built on it.

When I generate an execution plan for the query:

```
select count(*) from t1 where skew = 5;
```

Oracle estimates a cardinality of 22 if I have the histogram, and 41 if I don't. In fact, whatever I do with local queries, or remote (i.e., single-site) queries, Oracle always manages to come up with a cardinality of 22 if I have a histogram in place.

As soon as I write a distributed query, though, Oracle loses track of the histogram for whichever copy of the table is not at the **driving site**. In my test case, regardless of which site was driving the query, the execution plan was as follows:

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=33 Card=28 Bytes=9296)
1    0   HASH JOIN (Cost=33 Card=28 Bytes=9296)
2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=16 Card=22 Bytes=4488)
3    1     REMOTE* (Cost=16 Card=41 Bytes=5248) D920.JLCOMP.CO.UK@LOOPBACK
```

We are querying `t1` in two different disguises, but with the same predicate. Note how the cardinality for the local query is different from the cardinality for the remote version, which (depending on driving site) gets reported in the `other` column of the `plan_table` as one of the following:

```
SELECT "SKEW","SKEW2","PADDING" FROM "T1" "AWAY" WHERE "SKEW"=5
SELECT "SKEW","SKEW2","PADDING" FROM "T1" "A2" WHERE "SKEW"=5
```

Of course, if you check the `v$sql_plan` view at the remote site after running the query, you find that the remote site actually has used the histogram to work out the cardinality of the incoming query. But it's too late by then; the overall plan has already been fixed, and the site that created it may have selected the wrong join order or join method because it didn't have the right information about the remote table's cardinality.

Frequency Histograms

I mentioned earlier on that there are two different types of histogram, referred to by Oracle as the frequency histogram and the height balanced histogram. We will examine the frequency histogram first, as it is much simpler to deal with. The script `c_skew_freq.sql` in the online

code suite builds a test case, with a column called skew defined so that the value 1 appears once, 2 appears twice, and so on up to the value 80 that appears 80 times for a total of 3,240 rows.

When we create a frequency histogram on this table and query the view user_tab_histograms, this is what we see:

```
begin
        dbms_stats.gather_table_stats(
                user,
                't1',
                cascade => true,
                estimate_percent => null,
                method_opt => 'for all columns size 120'
        );
end;
/

select
        endpoint_number, endpoint_value
from
        user_tab_histograms
where
        column_name = 'SKEW'
and     table_name = 'T1'
order by
        endpoint_number
;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              1              1
              3              2
              6              3
             10              4
             15              5
                  . . .
           3160             79
           3240             80
```

This isn't the typical output from user_tab_histograms, which we will see later in the section on height balanced histograms. The endpoint_value is repeating a value from the table, and the endpoint_number reports the number of rows in the table where the skew column is less than or equal to that value. Reading down the list, we can see the following:

- There is one row with value 1 or less.
- There are three rows with value 2 or less.
- There are six rows with value 3 or less.
- There are ten rows with value 4 or less.

And so on. An alternative way to read the figures, which is valid because the presence of a frequency histogram tells us that only a small number of discrete values exists in that table, is to look at differences across the endpoint_number column, hence:

- There is one row with the value 1.
- There are three – one = two rows with the value 2.
- There are six – three = three rows with the value 3.
- There are ten – six = four rows with value 4.

And so on—in fact, given the way in which we compare the current count with the previous count to determine the number of rows for a given value, we can write a simple SQL statement using an analytic function to turn the frequency histogram in user_tab_histograms back into the list of values in our table:

```
select
        endpoint_value                  row_value,
        curr_num - nvl(prev_num,0)      row_count
from        (
        select
                endpoint_value,
                endpoint_number                         curr_num,
                lag(endpoint_number,1) over (
                        order by endpoint_number
                )                                       prev_num
        from
                user_tab_histograms
        where
                column_name = 'SKEW'
        and     table_name = 'T1'
)
order by
        endpoint_value
;
```

You may have noticed one oddity with the call to dbms_stats that I used to create the frequency histogram. Although only 80 distinct values appear in the column, and 80 rows in the resulting histogram, I asked Oracle to generate a histogram with 120 buckets.

There is a problem with dbms_stats and frequency histograms. From 9*i* onwards, Oracle started to create histograms by using the SQL I listed at the end of the "Getting Started" section. Unfortunately, this SQL will almost invariably fail to produce a frequency histogram if you ask for exactly the right number of buckets.

To get Oracle to spot the frequency histogram I needed for my 80 values, I found that I had to request 107 buckets. This is a backward step when compared to the old analyze command, which would build a frequency histogram if you specified the correct number of buckets. In particular, this means that if you have more than about 180 distinct values in a column, you

may find that you cannot build a frequency histogram on it unless you use the `analyze` command—or use a mechanism that I will demonstrate at the end of this section. (I understand that this problem has been fixed in 10*g* release 2.)

Once we have the histogram in place, we need to check what happens in various circumstances—basically by checking the calculated cardinality against our visual estimate for different types of query (script `c_skew_freq_01.sql` in the online code suite, results are from 9.2.0.6, but 8*i* differed by one on a few items), as shown in Table 7-1.

Table 7-1. *CBO Arithmetic Matching Human Understanding*

Predicate	Description	CBO	Human
`skew = 40`	Column = constant	40	40
`skew = 40.5`	Column = nonexistent, but in-range	1	0
`skew between 21 and 24`	Between range with mapped values	90	90
`skew between 20.5 and 24.5`	Between range with mapped values	90	90
`skew between 1 and 2`	Between range at extremes	3	3
`skew between 79 and 80`	Between range at extremes	159	159
`skew > 4 and skew < 8`	Greater than/less than range	18	18
`skew = -10`	Below high value	1	0
`skew = 100`	Above high value	1	0
`skew between -5 and -3`	Range below low value	1	0
`skew between 92 and 94`	Range above high value	1	0
`skew between 79 and 82`	Range crossing boundary	159	159
`skew = :b1`	Column = :bind	41	???
`skew between :b1 and :b2`	Column between :bind1 and :bind2	8	???

As you can see, any query using literal values gets the right answer—assuming you are prepared to accept 1 as an appropriate answer from the optimizer when our understanding lets us know that there really is no data.

ZERO CARDINALITY

As a general rule, the optimizer doesn't allow 0 to propagate through a cardinality calculation. Whenever the computed cardinality is 0, the optimizer plays safe, and introduces a cardinality of 1.

There is at least one special exception to this rule. In the case where a predicate is a logical contradiction such as `1 = 0`, the obvious zero cardinality is accepted. Often, these predicates are internally generated predicates that allow the optimizer to prune entire branches of execution plans at run time.

In the two examples with bind variables, we see that the CBO has used the standard 0.25% for the range, and when we check `column = :bind`, we find that 41 comes from `num_rows / num_distinct`.

Finally, looking carefully at the `density`, we discover (and confirm with a few more test data sets) that `density = 1 / (2 * num_rows)`—which is probably why we end up seeing a cardinality of 1 whenever we go outside the low/high range, or ask for a value that is not in the histogram.

There is one little detail that you might also notice if you look closely at histograms. In 10*g*, the number of buckets for a frequency histogram (`user_tab_columns.num_buckets`) matches the number of distinct values in the table. For earlier versions of Oracle, it is one short. This is just a change in the view definition, which failed to allow for frequency histograms in earlier versions.

Clearly, frequency histograms are a good thing—but there are a couple of special points to consider when creating them or using them:

- You might as well use 254 (the maximum allowed) as the number of buckets when gathering statistics—Oracle will only record the required number of rows anyway, so there is no point risking missing the critical bucket count.
- If critical values in your data change, your histogram needs to be rebuilt—otherwise Oracle will have an out-of-date picture of the data. Note that 40.5 reported a cardinality of 1: if we changed all the 40s to be 40.5, then Oracle would still report `skew = 40` with 40 rows.
- Frequency histograms have no impact on expressions with bind variables (if they are not peeked). The selectivity on `column = :bind` is still `1 / num_distinct`, the selectivity on ranges are still 5% and 0.25% for unbounded and bounded ranges.
- The CBO is better at spotting that range predicates fall outside the low/high values when there are histograms in place.

Faking Frequency Histograms

I have pointed out that `dbms_stats` needs more buckets than the `analyze` command to spot the option for a frequency histogram. So what do you do when there are 254 distinct values in your table—other than use the `analyze` command to create the histogram?

Take a close look at the procedures in `dbms_stats` called `prepare_column_values`, `get_column_stats`, and `set_column_stats`. The script `c_skew_freq_02.sql` in the online code suite gives an example of using these procedures to give a column a frequency histogram that could not be generated by a normal call to `dbms_stats`. The key points are the calls:

```
select
        skew,
        count(*)
bulk collect into
        m_val_array,
        m_statrec.bkvals
from
        t1
```

```
group by
        skew
order by
        skew
;

m_statrec.epc := m_val_array.count;
```

Here we collect the values and counts from the column into a pair of `varray` types, and record the array size ready for a call to `prepare_column_values`:

```
dbms_stats.prepare_column_values(
        srec        => m_statrec,
        numvals     => m_val_array
);
```

The code also reads the column statistics into local variables, modifies the variables, and then writes them back to the data dictionary using `get_column_stats` and `set_column_stats` respectively to read and write the values.

Unfortunately, the call to `set_column_stats` does not recognize that we are creating a frequency histogram, and so leaves the density unchanged—thus we have to use a little extra code to calculate the appropriate value before calling `set_column_stats`.

One final thought about frequency histograms and the ease with which you can create them. Although I would advise great caution when manipulating the statistics held in the data dictionary, the procedures are available, and they can be used. You know more about your data and the way your data is used than the optimizer can possibly work out. It is perfectly reasonable to invent some numbers that are a more accurate representation of the truth than the image the optimizer sees.

For example—assume you have a table with 1,000,000 rows, but the end users are only ever interested in 100,000 of those rows, which include a particular column that holds only 25 different values. It might be a perfectly sensible strategy to create a frequency histogram that uses a query against just those 100,000 rows and sets the column statistics to claim that the other 900,000 rows hold nulls.

As another example—if you have a column holding 400 different values, Oracle will only be able to create a height balanced histogram. This may well be too crude for your purposes. You might want to try creating a frequency histogram that describes the 254 most popular values instead.

Warning to Fakers

Strangely, the optimizer seems to have various code paths that can only be taken if you have faked some statistics. There are also various code paths that seem to be taken at the end of a complex decision tree—things like "We use `1/num_rows` at this point, but if `X < Y` and the table at the other end of the join has more than 100 times the rows in this table, then we use the `density`." So you do need to be very careful with the `set_xxx_stats()` procedures. You are trying to help Oracle by describing your data well, so your description has to be self-consistent, and odd things can happen if you don't cover all the details properly.

Script fake_hist.sql in the online code suite shows some effects of getting it "wrong"—which you might want to do by choice, of course. The most important thing to remember is that even though the histogram is supposed to describe your data by listing how many rows you have for each value, the optimizer will adjust these row counts according to other critical values that it holds about the table.

The maximum cumulative value in a frequency histogram—which appears in the column endpoint_number—should match the number of rows in the table minus the number of nulls in the column. If it doesn't, then Oracle adjusts the cardinality calculation accordingly.

The density is a critical value for the number of rows returned. If you have a specific value in the histogram that is supposed to return X rows, but X is less than density * (user_tables.num_rows - user_tab_columns.num_nulls), the optimizer will use the larger number.

In my test case, I have 300,000 rows in the table, and one column with ten distinct values. After generating the cumulative frequency histogram, Oracle had a density of 0.000001667 for the column, and knew that the value 10 appeared exactly 1,000 times according to the histogram.

So, on the baseline test, the following query showed an execution plan with a cardinality of 1,000:

```
select
        *
from
        t1
where
        currency_id = 10
;
```

I then ran several independent tests, using the hack_stats.sql script from the online code suite to modify the statistics on the data dictionary, rebuilding the baseline after each test:

- When I changed user_tables.num_rows to 150,000, the optimizer reported card = 500.
- When I changed user_tab_columns.num_nulls to 200,000, the optimizer reported card = 333.
- When I changed user_tab_columns.density to 0.01, the optimizer reported card = 3,000.

The last test was the most surprising—when you create a frequency histogram, the density is always set to 1/(2 * num_rows). But there seems to be a bit of optimizer code that deals with the possibility that someone has been messing about with the density, and has set it so high that one of the known values in the frequency histogram is apparently going to return fewer rows than an unknown value that is not in the histogram.

Clearly, this could be a very convenient way to identify up to 254 specific values in a column, and then set a figure for the number of rows that the optimizer should allow for every other value that might be used when querying this column with a constant. Of course, the usual problems of bind variables, bind variable peeking, and joins have to be taken into account.

"Height Balanced" Histograms

If we take our sample data set from the previous section and create a histogram with 75 buckets (script `c_skew_ht_01.sql` in the online code suite), we can notice small, but significant, details in the results that get stored in the data dictionary:

```
begin
        dbms_stats.gather_table_stats(
                user,
                't1',
                cascade => true,
                estimate_percent => null,
                method_opt => 'for all columns size 75'
        );
end;
/


select
        num_distinct, density, num_buckets
from
        user_tab_columns
where
        table_name = 'T1'
and     column_name = 'SKEW'
;

NUM_DISTINCT    DENSITY NUM_BUCKETS
------------ ---------- -----------
          80 .013885925          58

select
        endpoint_number, endpoint_value
from
        user_tab_histograms
where
        column_name = 'SKEW'
and     table_name = 'T1'
order by
        endpoint_number
;
```

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
                           ...
             59             71
             60             72
             62             73
             64             74
             65             75
             67             76
             69             77
             71             78
             73             79
             75             80

59 rows selected.
```

Looking first at the information from `user_tab_columns`, we see that Oracle has correctly counted that we have 80 distinct values in the column `n1`. The density is not 1/80, it's more like 1/72—but could have been almost anything. We also note that Oracle is claiming that we have a histogram with 58 buckets, despite the fact that we asked for 75 buckets, and can actually see in view `user_tab_histograms` that we got 75 buckets (the highest endpoint number is 75) recorded in 59 rows.

In fact 8*i*, 9*i*, and 10*g* vary in their results:

- 8*i* gives a density of 0.006756757, creates 74 buckets, and reports 58 buckets.
- 9*i* gives a density of 0.013885925, creates 75 buckets, and reports 58 buckets.
- 10*g* gives a density of 0.013885925, creates 75 buckets, and reports 75 buckets.

The bucket count is largely a cosmetic thing, but the variation in density and actual bucket count occurs because 8*i* uses the `analyze` command internally to generate the histogram, whereas 9*i* and 10*g* run the SQL statement shown at the end of the "Getting Started" section of this chapter. The reports you see in the 10053 trace about histograms (particularly *"height balanced"* histograms) vary across versions as well—and the output from 10.1.0.4 is slightly different from the output from 10.1.0.2. The following three examples all came from running exactly the same script—as you can see, the clarity of information keeps improving:

```
Version 10.1.0.4:
COLUMN:       SKEW(NUMBER)  Col#: 1      Table: T1   Alias: T1
    Size: 3  NDV: 80  Nulls: 0  Density: 1.3886e-002
    Histogram: HtBal  #Bkts: 75  UncompBkts: 75  EndPtVals: 59

Version 9.2.0.6:
Column:       SKEW  Col#: 1      Table: T1   Alias: T1
    NDV: 80        NULLS: 0         DENS: 1.3886e-002
    HEIGHT BALANCED HISTOGRAM: #BKT: 75 #VAL: 59

Version 8.1.7.4 reports no histogram details:
Column:       SKEW  Col#: 1      Table: T1   Alias: T1
    NDV: 80        NULLS: 0         DENS: 6.7568e-003
```

More important than the cosmetic details, though, is the fundamental problem of bad luck that this example highlights. Look back at the list of (endpoint_number, endpoint_value). You will notice that some rows are clearly missing—there is no row with the endpoint_number set to 74, for example. This is why we get a report (10*g* version) of 75 uncompressed buckets, but only 59 endpoint values. Some of the buckets are implied and not stored in the database. We ought to see a row with the values (74, 80) but Oracle can infer that row. Expanding the previous output to show all the buckets, stored and inferred, we should see the following:

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
                           ...
             59             71
             60             72
             61             73
             62             73
             63             74
             64             74
             65             75        ***
             66             76
             67             76
             68             77
             69             77
             70             78
             71             78
             72             79
             73             79
             74             80
             75             80
```

If you drew the graph for this data, you would find that you had to stack two bars on top of each other for each of the values 73, 74, 76, 77, 78, 79, and 80—but somewhere in the middle, there would be a little dip in your graph around 75 (marked with the ***).

Oracle recognizes "popular" values in your data by the fact that the uncompressed list of endpoints shows values repeating. And according to the statistics captured by Oracle, 75 is *not* a popular value, and 74 *is* a popular value—even though we know that we rigged the data so that there are more 75s than 74s.

With height balanced histograms, you can just get unlucky—you can check this with a few more tests on this simple data set. Create a histogram with 81 buckets, and you will find that every value from 72 (inclusive) upward is seen as a popular value. Increase the number of buckets to 82, and suddenly 71 becomes popular as well—which is a good thing—but at the same time 76 ceases to be popular—which is a bad thing.

There is no way you can be confident that any specific number of buckets will capture all your popular values—although a cavalier approach would be to go straight for 254 buckets, and then check whether all the popular values you cared about had been captured.

You will notice, of course, that the maximum number of buckets is only 254—so at the finest granularity any bucket represents about 1/250 (0.4%) of the number of rows that have a value. If you have more than 250 special values, then you are sure to miss some of them in the histogram. In fact, its worse than that—a single row could span nearly two buckets (0.8% of the

row count) and not be noticed by Oracle as a popular value. Worse still, every "very popular" value takes up more than its fair share of buckets—a small number of very popular rows could ensure that Oracle misses dozens of rows that you would like to classify as popular.

There are times when you may have to fall back on creating an artificial frequency histogram, because that's the best you can do to describe your data set to Oracle.

The Arithmetic

When the optimizer takes advantage of a height balanced histogram to work out cardinality, it adopts one of three major strategies.

Strategy 1 for calculating selectivity/cardinality: For `column = constant`, the constant may be a popular value. A popular value is one that fills at least one bucket, and therefore results in a gap in `user_tab_histograms` view. If Oracle spots such a value, it does its arithmetic by buckets. For example, in the preceding, the value 77 appears at endpoint 69, and the previous value (76) appears at endpoint 67: consequently Oracle considers 77 to be a popular value that spans 2 buckets out of 75. The selectivity of 77 is therefore 2/75, and the cardinality is 3,240 * 2/75 = 86.4—and with `autotrace` we see

```
Execution Plan (9.2.0.6 - select count(*) from t1 where skew = 77)
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=1 Card=86 Bytes=258)
```

Strategy 2 for calculating selectivity/cardinality: We have already noted that 75 is not a special value, so what happens when we execute a query to select all rows for that value? Oracle uses the `density` (not `1/num_distinct`, note). We have the `density` recorded as 0.013885925, so we expect a cardinality of 3,240 * 0.013885925 = 44.99, and again the plan shows

```
Execution Plan (9.2.0.6 - select count(*) from t1 where skew = 75)
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1 Bytes=3)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=1 Card=45 Bytes=135)
```

Of course, you might ask at this point how Oracle works out a density—it is a subject I don't really want to spend much time on, as it needs a fairly lengthy example to help me explain it—and the added value is negligible. So I'll give you the short answer now, and save the long answer for Volume 2.

In purely descriptive terms, the density is as follows:

```
sum of the square of the frequency of the nonpopular values /
(number of nonnull rows * number of nonpopular nonnull rows)
```

By virtue of the way the columns have been constructed, this can be calculated from the big histogram query at the end of the "Getting Started" section as follows:

```
( sum(sumrepsq) - sum(maxrep(i) * maxrep(i) ) /
( sum(sumrep) * ( sum(sumrep) - sum(maxrep(i) ) )
```

The subscripts (i) are there to indicate that only certain rows are to be selected from the query results to contribute to their maxrep value. These rows are the one meeting the following condition:

```
maxbkt > min(minbkt) + 1 or min(val) = max(val)
```

In the special case where there are no popular values (or at least, none that have been detected by the query), this formula degenerates to the simpler form

```
sum(sumrepsq) / (sum(sumrep) * (sum(sumrep))
```

This could be verbalized as: the sum of the squared frequencies over the square of the summed frequencies.

If you analyze the formula carefully, you will realize that there is a potential trap that could give you problems if you happen to be unlucky with your data set. The idea of this density calculation is to factor out the impact of popular values.

For example, if you have 2,000 rows of which 1,000 hold the same value, then a suitably adjusted density would be 1/2,000 rather than 1/1,001 because Oracle should be able to notice queries against the popular value and act accordingly. In principle, then, we want the formula for the adjusted density to reduce the density that would appear in the absence of a histogram. In practice, this works well for extreme cases like the one I have just described.

However, if you have popular values in your data and Oracle does not notice them when building the histogram, the formula will produce an increased density. Yet again, you have to know your data, and know what you are expecting in order to be sure that the optimizer will be able to do the right job.

Strategy 3 for calculating selectivity/cardinality: After that brief digression, we can investigate the issue of range scans. There are many possible variants on the theme, of course—so we will stick with just two indicative cases. Since the data set I have been using is a fairly odd one, I have created another data set for these tests, to make sure that the results showed very clearly what Oracle is doing. The script is hist_sel.sql in the online code suite.

The script generates 10,000 rows using a scaled normal distribution centered on the value 3,000 ranging from –5,000 to 11,000. Superimposed on this distribution, I have 500 rows each of the values 500, 1,000, 1,500, and so on up to 10,000—a list of 20 popular values.

After analyzing this table with 250 buckets (80 rows per bucket), I get a histogram that shows the spikes from my 20 popular values. The number of distinct values is 5,626 and the column density is 0.000119361 (a convincing drop from 1 / 5,626 = 0.000177746).

My first test case is a range scan across buckets with no popular values: n1 between 100 and 200. If we check the histogram, we find that the required values fall across three buckets with endpoints 17, 117, and 251.

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              8           -120
              9             17
             10            117
             11            251
             12            357
```

So we apply the standard formula—bearing in mind that we have to break our required range across two separate buckets so that we examine ranges 100–117 and 117–200:

```
Selectivity = (required range) / (high value - low value) + 2 * density =
              (200-117)/(251-117) + (117-100)/(117-17) + 2 * 0.000177746 =
              0.619403 + 0.17 + .000355486 =
              0.789047508

Cardinality = selectivity * number of rows IN A BUCKET =
              0.789047508 * 80 = 63.1238
```

Sure enough, when we run the query with autotrace, we see the following:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=60 Card=63 Bytes=945)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=60 Card=63 Bytes=945)
```

The second test case is, of course, a range scan across buckets with a popular value: `n1 between 400 and 600`. (One of my spikes was at `n1 = 500`.) We need to check up on a little section from the histogram that will be relevant:

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
             12            357
             13            450
             19            500        ***
             20            520
             21            598
             22            670
```

Note that 500 is a (very) popular value—with six buckets allocated from the histogram. So we expect to account for at least six buckets plus the rest of the range. Checking carefully, we see that the range 400 to 600 will extend from buckets 12 to 22—broken down as follows:

- A selection from 357 to 450
- All buckets from 450 to 598
- A selection from 598 to 670

So we have eight whole buckets (`endpoint_numbers` 13 to 21) plus

```
(450 - 400) / (450 - 357) + (600 - 598) / (670 - 598) + 2 * 0.000177746 =
        50 / 93 + 2 / 72 + 0.000355486 =
        0.537634 + 0.0277778 + 0.000355486 =
        0.565768
```

Remembering to add the 8 for the complete buckets, we get a cardinality of

```
Cardinality = selectivity * number of rows IN A BUCKET =
        8.565867 * 80 = 685.3
```

And again, running the query through autotrace, we get

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=60 Card=685 Bytes=10275)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=60 Card=685 Bytes=10275)
```

Note, because 8*i* uses a call to the old analyze mechanism to generate histograms, the histogram and the density will not match the figures for 9*i* and 10*g*. However, the principles of the arithmetic are the same once the histogram and density have been generated.

Data Problems Revisited

In Chapter 6, I showed a couple of examples where Oracle had a problem getting a good estimate of selectivity because it had been deceived by non-Oracle design decisions, and then I claimed that the creation of a histogram helped in such cases. Let's take a closer look at what's going on in these cases.

Daft Datatypes

We created a table that held five years of date information, but stored it in three different forms—a genuine Oracle date, an eight-digit number looking like `yyyymmdd`, and a character string looking like the number `'yyyymmdd'`. The creation code is repeated here and in the script `date_oddity.sql` in the online code suite:

```
create table t1 (
        d1              date,
        n1              number(8),
        v1              varchar2(8)
)
;

insert into t1
select
        d1,
        to_number(to_char(d1,'yyyymmdd')),
        to_char(d1,'yyyymmdd')
from
        (
        select
                to_date('31-Dec-1999') + rownum        d1
        from
                all_objects
        where
                rownum <= 1827
        )
;
```

Queries that used an equality predicate can cope with the nondate data types, and if we sort the data by the pseudo-date columns, the order works out correctly. But range-based

predicates produce incorrect cardinalities because the numeric and character versions of the information have huge gaps in them. Oracle is aware that 1 April 2003 is one day after 31 March 2003, but cannot possibly apply the same logic when comparing 20030401 to 20030331, or `'20030401'` to `'20030331'`. As we saw in Chapter 6, a query with a predicate of

```
where n1 between 20021230 and 20030105
```

produced a cardinality of 396 in Oracle 9*i*, although we know that the query is supposed to report 7 days, and the equivalent query against the genuine date column produced a cardinality of 8. When we built a histogram of 120 buckets, the query against the numeric column got a lot closer, with an estimated cardinality of 15 rows. We are now going to see why.

In the "Getting Started" section of this chapter, I showed an SQL statement you could run against the `user_tab_histograms` table to show the widths and heights of the bars that the histogram data represented. The version of `date_oddity.sql` that goes with this chapter does the same for the 120-bucket histogram against the numeric data, and a critical section of the results look like this:

```
    BUCKET    LOW_VAL   HIGH_VAL      WIDTH     HEIGHT
---------- ---------- ---------- ---------- ----------
        68   20021028   20021112         84      .1813
        69   20021112   20021127         15      1.015
        70   20021127   20021212         85      .1791
        71   20021212   20021227         15      1.015
        72   20021227   20030111       8884      .0017
        73   20030111   20030126         15      1.015
        74   20030126   20030210         84      .1813
        75   20030210   20030225         15      1.015
        76   20030225   20030312         87       .175
        77   20030312   20030327         15      1.015
```

As you can see, the histogram manages to give Oracle a better picture of the data. There is clearly a very big gap—with some very thinly spread data—between the numbers 20021227 and 20030111. This is the thing that people would expect because they know about the end-of-year gap, but that is a piece of human-oriented information that is not available to the computer—without the assistance of the histogram. Taking the large-scale view, which loses the detail about the months, the histogram gives Oracle a graph of the data (part 3 of Figure 7-2) that allows it to come close to the right answer.

Similarly, the histogram shows that from the end of each month to the beginning (actually, with our bucket size, the middle) of each month, the data is a little thin, but there is approximately one row per day for the rest of the month. Of course, looking at the first line in the preceding list, we know that 20021112 is meant to be just 14 days after 20021028, but Oracle thinks there are 84 days (numbers) that need to share the 14 rows from our original table.

The arithmetic for our query falls completely into bucket 72, so the optimizer uses the standard formula for selectivity, but using the endpoint values from the bucket. We have a `between` clause, which means we have to allow for a correction factor at both ends of the range.

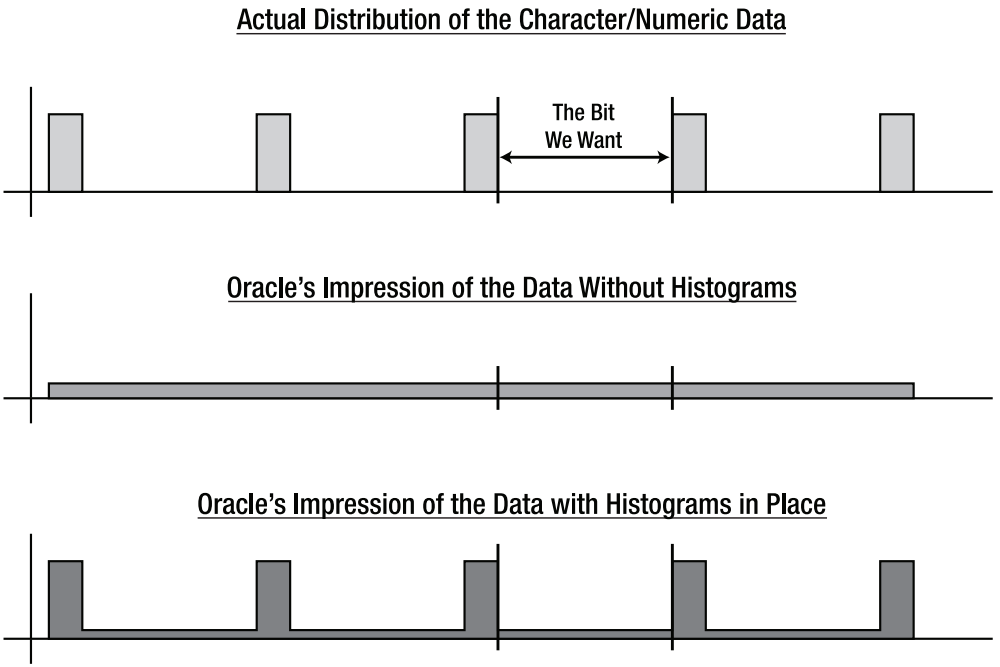

Figure 7-2. *Histograms and wrong data types*

■ **Note** When you start creating histograms, it is no longer guaranteed that `density = 1 / num_distinct`, (although it is still true in this test case, where we have no popular values and don't have a frequency histogram).

In the past, we have used `1 / num_distinct` for the correction factor and not worried about whether we should perhaps use the `density`. In most cases, it is actually the `density` that needs to be used, so our formula becomes the following:

```
selectivity =
        ('required range' / 'total range of bucket' ) + 2 * density =
        (20030105 - 20021230) / (20030111 - 20021227) + 2 * 0.000547345 =
           8875 / 8884 + 0.00109469 =
                1.00008
```

But this is the selectivity that has to be applied to the number of rows in the buckets involved—not to the entire table—and there are 1,827 / 120 = 15 rows per bucket—`round(15 * 1.00008) = 15`, as required.

You will notice that every single value in the table appears exactly the same number of times. There are no values with very large numbers of occurrences. According to the traditional rules of thumb (small number of popular values, etc.) this data set does not need a histogram. But as you can see, a histogram makes an enormous difference to the optimizer.

Any column with a data distribution that does not have a completely flat, unbroken graph, may need a histogram if you use that column in a `where` clause. This graph is flat, but it has some big gaps in it that the optimizer needs to know about.

Dangerous Defaults

The other problem we examined in Chapter 6 revolved around the idea of using a special (extreme) value to avoid using a null. We had a genuine date column, but used the data 31 December 4000 to represent missing data. Again, range-based queries suffered because although the optimizer was doing the date-based arithmetic correctly, the standard formula was using a high value that was totally unrealistic.

We had data from 1 January 2000 to 31 December 2004, and a predicate:

```
where date_closed between to_date('01-Jan-2003','dd-mon-yyyy')
                      and to_date('31-Dec-2003','dd-mon-yyyy')
```

Clearly, we expect Oracle to work out that we want one-fifth of the data. However, Oracle sees a high value of 31 December 4000, and works out a selectivity of about 3/2000 because it uses the following formula:

```
(31st Dec 2003 - 1st Jan 2003) / (31st Dec 4000 - 1st Jan 2000) + 2/1828 = 0.00159
```

But when we build a histogram of just 11 buckets—chosen simply as two per year plus one spare, just to show you that sometimes a very small number of buckets will do—the CBO suddenly got a very good answer for its selectivity, hence cardinality. Using the same technique to display the data in the histogram in *graph-ready* form, the entire histogram looks like this:

```
    BUCKET LOW_VAL     HIGH_VAL          WIDTH     HEIGHT
---------- ----------- ----------- ---------- ----------
         1 01-Jan-2000 15-Jun-2000        166   100.0548
         2 15-Jun-2000 28-Nov-2000        166   100.0548
         3 28-Nov-2000 13-May-2001        166   100.0548
         4 13-May-2001 27-Oct-2001        167    99.4556
         5 27-Oct-2001 11-Apr-2002        166   100.0548
         6 11-Apr-2002 24-Sep-2002        166   100.0548
         7 24-Sep-2002 09-Mar-2003        166   100.0548
         8 09-Mar-2003 23-Aug-2003        167    99.4556
         9 23-Aug-2003 05-Feb-2004        166   100.0548
        10 05-Feb-2004 20-Jul-2004        166   100.0548
        11 20-Jul-2004 31-Dec-4000     729188      .0228
```

If you check the SQL I have used for this table (script `defaults.sql` in the online code suite), you will find that the actual values stored in the histogram are the *Julian* equivalents of the dates, and I have used a `to_date()` function to display them. An approximate graphical representation of these numbers appears in the third part of Figure 7-3.

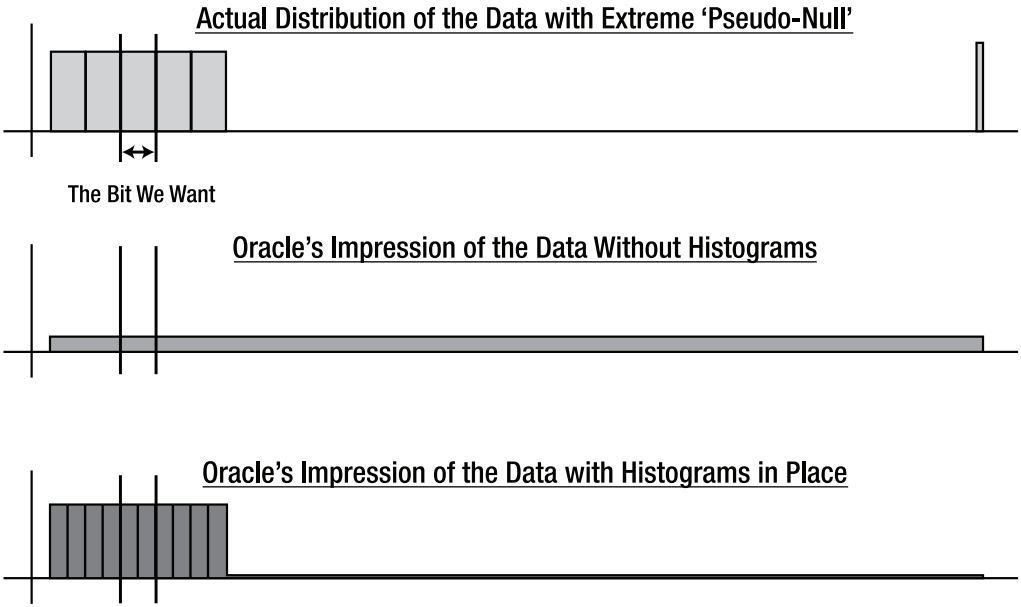

Figure 7-3. *Histograms and pseudo-nulls*

The data set was created with 100 rows per day in the legal range, with 1% of the data set to 31 December 4000. Notice how the last bucket manages to catch the oddity, showing an enormous date range with a very small number (0.0228) of rows per day, while the rest of the histogram shows roughly 100 rows per day for the first 4.5 years.

Any queries we run for the first four years are going to give pretty accurate cardinality estimates. It is only in the last six months (20 July onward) that the figures are going to go bad—and we can narrow the room for error by creating a lot more buckets. Alternatively, we might forget about analyzing such predictable data, and simply write a little program that created an artificial histogram that gave Oracle even better details.

Again, we can see that a histogram isn't just for data with some popular values—we have a graph that is not a simple, flat, continuous line, and the CBO needs to know about it.

In passing, when left to do automatic statistics gathering, 10*g* did not collect a histogram on this data set, even though it needs it.

Summary

If there is a column in your database that has an interesting data distribution and is used in a `where` clause (even in a join condition), then you may need to create a histogram on that column.

Such columns that hold only a small number of values (less than 255) are good candidates for frequency histograms, but be careful about the rate of change of data and the need to keep the histograms up to date.

Oracle may have trouble creating a frequency histogram if you use the `dbms_stats` package (which you should) even if you have only 180 to 200 distinct values in the column. In this case, you can always use manual methods to generate the histogram and populate the data dictionary.

If you have a larger number of distinct values in a column, you will have to make a guess about the number of buckets. In most cases, simply electing to use the maximum will be a safe option. But you may get unlucky. Be aware of your popular, or strange, values, and make sure that they are visible in the histogram. Remember also that the histogram exists to help the optimizer—you can always subvert Oracle's internal method for generating a histogram to present the critical features of your data to the optimizer.

Remember that there are many more reasons for histograms than just the simple-minded examples of "a small number of very popular values."

Test Cases

The files in the download for this chapter are shown in Table 7-2.

Table 7-2. *Chapter 7 Test Cases*

Script	Comments
`hist_intro.sql`	Generates a large set of data with a normal distribution
`similar.sql`	Demonstration of effects of `cursor_sharing = similar`
`dist_hist.sql`	Demonstrates that distributed queries ignore histograms
`c_skew_freq.sql`	Generates a frequency histogram example
`c_skew_freq_01.sql`	Some examples of the cardinality you get with frequency histogram
`c_skew_freq_02.sql`	Example of generating a frequency histogram
`fake_hist.sql`	Examples of the effects of faked statistics on histograms
`hack_stats.sql`	Script to modify statistics directly on the data dictionary
`c_skew_ht_01.sql`	Some examples of the cardinality you get with "height balanced" histograms
`hist_sel.sql`	Further example of cardinality with "height balanced" histograms
`date_oddity.sql`	Example of how histograms can help with incorrect data types
`defaults.sql`	Example of how histograms can help with extreme default values
`setenv.sql`	Sets the standardized test environment for SQL*Plus

CHAPTER 8

Bitmap Indexes

When I started writing this chapter, I thought it would be much shorter than the equivalent chapter on B-tree indexes (Chapter 4) for two reasons. First, bitmap indexes are actually "hidden" inside B-tree structures, so most of the groundwork for the calculation is in the other chapter (so you probably ought to read that chapter before you read this one). Second, bitmap indexes are much less subtle (I thought) than B-tree indexes, so there are fewer complications to consider.

However, as I started writing, I realized that my casual assumption was wrong. Strategically, you use bitmap indexes very differently from B-tree indexes, so although there isn't a lot of new information to worry about when you consider just one bitmap index, you've missed the point of bitmaps if you stop at one. And if you start looking at the arithmetic that the optimizer uses when combining bitmaps, or generating them on the fly, some very strange things begin to appear.

Before you carry on with this chapter, though, there is one point that I have to make very clear. You can get some strange effects by building controlled experiments, and in some cases, the effects I've produced are so strange that I have decided that they are bugs. Why, for example, should the cost of using a bitmap index change in an *arbitrary* direction when you change the value for `db_file_multiblock_read_count`?

Because of the apparent anomalies in the bitmap calculations, I have been a little less fussy about the precision of the arithmetic in this chapter. If I have a theory that predicts a value of 27.23 and the actual answer is 27.51, is the error in my theory or is it a side effect of a bug? Consequently, when I get an answer that seems to be reasonably close to a prediction, I haven't spent a lot of time trying to track down the cause of the difference in case I am wasting time trying to find a solution that doesn't exist.

Getting Started

It's always a good idea to start with a concrete example, and in this chapter I'd like to start with just one table that demonstrates most of the key points affecting the cost of using a bitmap index.

As usual, the target tablespace should be built with 8KB blocks, preferably locally managed with uniform 1MB extents, and not ASSM if you want to get the same results as I did. Also, if you haven't been using the sample `init.ora` definitions from the online code suite, then you will need to check that your `db_file_multiblock_read_count` is set to `8`. As with the B-tree costs, we'll start with CPU costing disabled. The following is an extract from the script `bitmap_cost_01.sql` in the online code suite:

```
create table t1
pctfree 70 pctused 30
as
select
        mod((rownum-1),20)          n1,          -- 20 values, scattered
        trunc((rownum-1)/500)       n2,          -- 20 values, clustered
        mod((rownum-1),25)          n3,          -- 25 values, scattered
        trunc((rownum-1)/400)       n4,          -- 25 values, clustered
        mod((rownum-1),25)          n5,          -- 25 values, scattered for btree
        trunc((rownum-1)/400)       n6,          -- 25 values, clustered for btree
        lpad(rownum,10,'0')         small_vc,    -- target of select
        rpad('x',220)               padding      -- waste some space
from
        all_objects
where
        rownum  <= 10000
;

create bitmap index t1_i1 on t1(n1)
pctfree 90
;

create bitmap index t1_i2 on t1(n2)
pctfree 90
;

create bitmap index t1_i3 on t1(n3)
pctfree 90
;

create bitmap index t1_i4 on t1(n4)
pctfree 90
;

create /* B-tree */ index t1_i5 on t1(n5)
pctfree 90
;

create /* B-tree */ index t1_i6 on t1(n6)
pctfree 90
;
```

```
begin
        dbms_stats.gather_table_stats(
                ownname            => user,
                tabname            => 'T1',
                cascade            => true,
                estimate_percent   => null,
                method_opt         => 'for all columns size 1'
        );
end;
.
;
```

There are six significant columns in this table, each individually indexed. Columns `n1` and `n2` each hold 20 different values, but one column is generated using the `mod()` function on the `rownum` so the values are scattered evenly across the table, and the other column is generated using the `trunc()` function, so all the rows for a given value are clustered together in groups of 500. I have created bitmap indexes on these two columns to show how variations in data clustering affect the statistics.

Columns `n3` and `n4` are similar but with 25 values each. These columns are here so that we can see something of how the optimizer works out costs when combining bitmap indexes.

Columns `n5` and `n6` are identical to columns `n3` and `n4` but have B-tree indexes on them, which allows us to see how the sizes of bitmap indexes and B-tree indexes differ.

You will note that in the test case I have created both the table and its indexes with fairly large settings of `pctfree` to push the sizes up a bit. Coincidentally, this test case happened to highlight a little difference that I hadn't previously noticed in the way 8*i* and 9*i* allocate bitmap entries to leaf blocks—it's surprising how often you can learn something new about Oracle by accident once you start building test cases.

The statistics on the indexes will be as shown in Table 8-1, if you have used 9*i* for the test case.

Table 8-1. *Statistical Differences Between Bitmap and B-tree Indexes*

Statistic	t1_i1 (Bitmap)	t1_i2 (Bitmap)	t1_i3 (Bitmap)	t1_i4 (Bitmap)	t1_i5 (B-tree)	t1_i6 (B-tree)
blevel	1	1	1	1	1	1
leaf_blocks	60	10	63	9	217	217
distinct_keys	20	20	25	25	25	25
num_rows	120	20	125	25	10,000	10,000
clustering_factor	120	20	125	25	10,000	1,112
avg_leaf_blocks_per_key	3	1	2	1	8	8
avg_data_blocks_per_key	6	1	5	1	400	44

Points to notice:

- The number of leaf blocks in the bitmap indexes is dramatically affected by the clustering of the data (`n1` is scattered, the index has 60 leaf blocks; `n2` is clustered, the index has 10 leaf blocks; similarly `n3` / `n4` show 63 / 9 blocks). Generally, bitmap indexes on scattered data tend to be larger than bitmap indexes on clustered, but otherwise similar, data. The size of a B-tree index is not affected the same way (`n5` is scattered data, and `n6` is the same data clustered—both indexes have 217 blocks).
- This specific example shows how counterintuitive the details of bitmap index sizing can be. The indexes `t1_i1` and `t1_i2` have 20 distinct keys, the indexes `t1_i3` and `t1_i4` have 25 distinct keys. Comparing `t1_i1` to `t1_i3` (the two indexes on scattered data), the increase in the number of distinct values has resulted in an increase in the number of leaf blocks. Comparing `t1_i2` to `t1_i4` (the two indexes on clustered data), the increase in the number of distinct values happens to have produced the opposite effect.
- In cases with tables that are not very large, you may find that the values for `distinct_keys` and `num_rows` for a bitmap index are identical—this is a *coincidence*, not a rule. (If you build the test case under 8*i*, you will find that `distinct_keys` and `num_rows` happen to come out to the same value in all cases.)
- In this specific example, the `num_rows` is larger than the `distinct_keys` in the scattered examples (`t1_i1` and `t1_i3`) because (a) the string of bits for each key value had to be broken up into several pieces to fit in the leaf blocks, and (b) the result came from 9*i*.

■ **Note** This is the little detail I discovered while writing this chapter—9*i* handles bitmap leaf block allocation differently from 8*i* for large values of `pctfree`. And for bitmap indexes, you might want to start with a large value of `pctfree`, such as 50, or even 67, to reduce the damage caused by the occasional piece of DML. Wolfgang Breitling—one of my technical reviewers—then discovered when running one of the test scripts that this change also has fairly fluid boundaries that are dependent on the block size used for the index. At the time of writing, neither of us has yet investigated the full extent of this change.

- The `clustering_factor` of a bitmap index is just a copy of the `num_rows` value for the index. The `clustering_factor` has no *direct* connection with the scattering of data in the table. Data scattering does affect the size of bitmap index entries and can make it look as if the `clustering_factor` and the data scatter are arithmetically connected; but this is a side effect, not a direct consequence.
- The `avg_leaf_blocks_per_key` is still roughly appropriate in bitmap indexes. (It is still calculated as `round(leaf_blocks / distinct_keys)`.)
- The `avg_data_blocks_per_key` is completely irrelevant in bitmap indexes. (It is still calculated as `round(clustering_factor / distinct_keys)`, but as you saw earlier, the `clustering_factor` in bitmap indexes does not describe the table.)

DML AND BITMAP INDEXES DON'T MIX

Generally, it is quite disastrous to do any updates to columns with enabled bitmap indexes, or even to do inserts and deletes on a table with enabled bitmap indexes. Concurrent DML operations can easily cause deadlocks, and even small serialized DML operations can cause bitmap indexes to grow explosively.

The situation for small, occasional changes has improved in 10*g*, but for earlier versions, you may find that if you can't disable the bitmap index when you need to apply changes to the data, then creating it with an initial large percentage of empty space (specifically `pctfree 67`) may stabilize the index at a reasonable level of space wastage.

Given that the meaning of some of the statistics (and in particular the `clustering_factor`) is different for bitmap indexes, what effect does that have on the estimated cost of using the index? Try a simple `column = constant` query on each of the columns `n6`, `n5`, `n4`, and `n3` in turn. The columns all have the same number of distinct values, so the results we get from running `autotrace` against the four queries may be very informative. Here are the four execution plans from a 9.2.0.6 system:

```
Execution Plan - (n6: B-tree index on clustered column)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=54 Card=400 Bytes=5600)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=54 Card=400 Bytes=5600)
   2    1     INDEX (RANGE SCAN) OF 'T1_I6' (NON-UNIQUE) (Cost=9 Card=400)

Execution Plan - (n5: B-tree index on scattered column)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=170 Card=400 Bytes=5600)
   1    0   TABLE ACCESS (FULL) OF 'T1' (Cost=170 Card=400 Bytes=5600)

Execution Plan - (n4: bitmap index on clustered column)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=114 Card=400 Bytes=5600)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=114 Card=400 Bytes=5600)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP INDEX (SINGLE VALUE) OF 'T1_I4'

Execution Plan - (n3: bitmap index on scattered column)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=117 Card=400 Bytes=5600)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=117 Card=400 Bytes=5600)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP INDEX (SINGLE VALUE) OF 'T1_I3'
```

You probably weren't surprised to see that the queries against the columns with B-tree indexes had different execution plans for the clustered data (where the index is considered to be reasonably effective) and the scattered data (where the index is considered to be a waste of space).

But the costs from the two bitmap indexes are nearly identical, regardless of whether the target data is extremely scattered, or very densely packed. (It is a nuisance that the optimizer doesn't report any cost against bitmap indexes—this is a critical bit of useful information that is also missing from the `explain plan` and can only be found in `10053 trace` files.)

Of course, looking at the statistics available for the two bitmap indexes, it's not really surprising that the costs given for the two paths are about the same—there isn't really anything in the statistics that describes the way that the data is distributed through the table. (You may think the factor of 7 difference in the `leaf_block` counts should mean something, and there is an element of truth in this, but it's not really a good indicator. In this example, that factor is large because of my artificial 90% free space declaration.)

So, where does the calculated cost come from for bitmap indexes? I can tell you part of the answer, and approximate another part—but this is where my warning about accuracy and bugs starts to have effect.

The Index Component

Rerun the queries with `event 10053` enabled to show the CBO calculation, and you will find the following details in the trace files (this example is from a 9*i* system):

```
For the query n3 = 2 bitmap on scattered data
-------------------------------------------
Access path: index (equal)
      Index: T1_I3
  TABLE: T1
      RSC_CPU: 0   RSC_IO: 3
  IX_SEL:  4.0000e-002  TB_SEL:  4.0000e-002
******** Bitmap access path accepted ********
Cost: 117 Cost_io: 117 Cost_cpu: 0.000000 Selectivity: 0.040000
Not believed to be index-only.
  BEST_CST: 116.54  PATH: 20  Degree:  1


For the query n4 = 2 bitmap on clustered data
-------------------------------------------
Access path: index (equal)
      Index: T1_I4
  TABLE: T1
      RSC_CPU: 0   RSC_IO: 1
  IX_SEL:  4.0000e-002  TB_SEL:  4.0000e-002
******** Bitmap access path accepted ********
Cost: 114 Cost_io: 114 Cost_cpu: 0.000000 Selectivity: 0.040000
Not believed to be index-only.
  BEST_CST: 114.34  PATH: 20  Degree:  1
```

Two important points to notice. First, the `best_cst` in each case is not a whole number—it is reported to two decimal places, and then rounded to the nearest whole number (116.54 went up to 117, 114.34 went down to 114).

The second point comes from the numbers we can see (which are, of course, specific to this example). The reported cost of the index in both cases (`RSC_IO: 3` and `RSC_IO: 1`) is derived in the same way it was for B-tree indexes: `ceiling` (`leaf_blocks` * *effective index selectivity*) plus 0 for the `blevel` (remember from Chapter 4, when the `blevel` is 1, it is excluded from the calculation).

One final detail, which isn't immediately obvious from these figures, is that the final cost of the query multiplies the stated index component of the cost by a factor of 1.1. Possibly this scaleup is to give B-tree indexes a small benefit over bitmap indexes when a table has both types of index in place; possibly it is there to reduce the risk of the optimizer doing a B-tree to bitmap conversion unnecessarily.

If you work backwards from the figures previously reported, you get the following:

- **Using index** `t1_i3`: The cost of the index is 3, which is scaled up to 3.3. But the `best_cst` equals 116.54, so the cost of hitting the actual table blocks has been estimated at 116.54 – 3.3 = 113.24.

- **Using index** `t1_i4`: The cost of the index is 1, which is scaled up to 1.1. But the `best_cst` equals 114.34, so the cost of hitting the actual table blocks has been estimated at 114.34 – 1.1 = 113.24.

Forget about data clustering and data scattering. For bitmap indexes, the calculated cost of hitting the actual *table* for a given amount of data is the same, regardless of how the data is actually scattered or clustered. For bitmap indexes, the cost based optimizer knows *nothing* about the real data scatter.

This startling difference in the costing strategy has occasionally resulted in some confusion about the benefits of using bitmap indexes. Remember that we are looking at the optimizer's estimate of what it thinks will happen, and in particular the optimizer's estimate of the number of I/O requests that will have to take place. But sometimes people will see a B-tree index being ignored by the optimizer and discover that when they change it to a bitmap index, the optimizer uses it. (That's exactly what would happen if you changed my index `t1_i5`—the B-tree on the scattered data—into a bitmap index.)

As a consequence of flawed experiments, it has become *common knowledge* that a bitmap index will *be more effective when fetching larger percentages of the data from the table*. But this is not correct. If you did the proper tests between a bitmap index and a B-tree index, the amount of work done would be the same for both, whatever the cost says. The run-time engine will request a small number of blocks from the index and then request a load of blocks from the table. The number of table blocks will be the same, regardless of whether the index is a bitmap or B-tree.

What has really happened with bitmap indexes then? The optimizer has lost some critical information about the data scatter in the table and has to invent some magic numbers as a guess for the data scatter. What effect does this have when you decide to change a B-tree index into a bitmap index?

If you start with a B-tree index that produces a low cost, then the equivalent bitmap index will probably produce a higher cost—think about the effect of changing B-tree index `t1_16` (cost 54) into bitmap index `t1_14` (cost 114).

If you start with a B-tree index that produces a high cost, then the equivalent bitmap index will probably produce a lower cost—think about the effect of changing B-tree index `t1_i5` (cost so high that it wasn't used) into bitmap `t1_i3` (cost 117, low enough to be used).

■ **Caution** Apart from their incomplete interpretation of their experiments, anyone who advocates turning B-tree indexes into bitmap indexes to "tune" queries is probably overlooking the problems of maintenance costs, locking, and deadlocking inherent in using bitmap indexes on columns where the data is subject to even a low rate of modification.

The Table Component

Going back to the calculations for the two queries with the following predicates:

```
where n3 = {constant}
where n4 = {constant}
```

we have worked out that the optimizer has assigned a cost of 113.24 for visiting the table. Where does this number come from? At this point, we might also take advantage of the other two bitmap indexed columns in the table to check the `10053 traces` for:

```
where n1 = {constant}
where n2 = {constant}
```

If we do so, we would see the table-related cost for these two predicates coming out as 137.99.

Given that each value of `n1` or `n2` retrieves 500 rows, and each value of `n3` or `n4` retrieves 400 rows, it will be no great surprise that the 137.99 you get from the queries against `n1` and `n2` is fairly close to 113.24 * 500 / 400. (Close, but not really close enough, to the value of 141.55 that this calculation gives.) The optimizer seems to be operating fairly consistently in its assumption about data scattering.

According to K. Gopalakrishnan et al. (*Oracle Wait Interface: A Practical Guide to Oracle Performance Diagnostics and Tuning*, Osborne McGraw-Hill, 2004), the optimizer assumes that 80% of the target data is tightly packed, and 20% of the target data is widely scattered. You could apply this approach in two slightly different ways, but let's try it on columns `n1` / `n3` and `n2` / `n4` assuming that 80% of our rows are clustered, and that the remaining 20% of the rows are scattered across the *rest* of the table blocks. The results are shown in Table 8-2.

Table 8-2. *Bitmap Arithmetic—Checking the 80/20 Split*

Value	Indexes on n1 and n2 (20 Values)	Indexes on n3 and n4 (25 Values)
Rows in table	10,000	10,000
Blocks in table	1,112	1,112
Rows per block	8.993	8.993
Rows selected on equality	500	400
Clustered rows	0.8 * 500 = 400	0.8 * 400 = 320
Blocks needed for clustered rows	400 / 8.993 = 44.48	320 / 8.993 = 35.58
Blocks remaining	1,112 – 44.48 = 1,067	1,112 – 35.55 = 1,076

Table 8-2. *Bitmap Arithmetic—Checking the 80/20 Split*

Value	Indexes on n1 and n2 (20 Values)	Indexes on n3 and n4 (25 Values)
Scattered rows (20%)	0.2 * 500 = 100	0.2 * 400 = 80
Blocks needed for scattered rows	100	80
Total count of table blocks needed	100 + 44.48 = 144.48	80 + 35.58 = 115.58
Reported table cost (from 10053)	137.99	113.24

So the assumption that the optimizer used 80% as an approximation for clustering is close, but not quite close enough to give the values we actually see.

But there's a further complication. If you change the value of `db_file_multiblock_read_count`, the cost of these queries change as well, although not in a totally consistent fashion. Broadly speaking, as the parameter increases, the cost of using the bitmap index also increases—but there are odd dips and wobbles in the calculation.

Table 8-3 shows some results (generated by the script `bitmap_mbrc.sql` in the online code suite) for the cost of the following query:

```
select  /*+ index(t1) */
        small_vc
from
        t1
where
        n1 = 2
;
```

As you can see, the cost generally increases as the `db_file_multiblock_read_count` goes up, but the pattern is not stable.

Table 8-3. *db_file_multiblock_read_count Changing the Index Cost*

db_file_multiblock_read_count	Reported Cost
4	131
8	141
16	155
32	170
64	191
80	201
81	199
82	200
83	202
84	200

However, for the purposes of getting a reasonable approximation, I'm happy to stick with an assumption that the optimizer is inventing something like an 80/20 split between clustered rows and scattered rows as its built-in rule for bitmap indexes.

I suspect there is a formula similar to the formula for the *adjusted dbf_mbrc* (see Chapter 2) for modifying the cost of the bitmap calculation; but usually you only need a reasonable approximation to determine why an execution plan is being ignored or accepted, and the basic 80/20 approximation is good enough for me at present.

Bitmap Combinations

If we run with the 80/20 figures and live with the fact that any predictions we make are going to have small errors, what sort of precision are we likely to see with more complex queries? Let's start with a simple `bitmap and` in script `bitmap_cost_02.sql` in the online code suite:

```
select
        small_vc
from
        t1
where
        n1 = 2          -- one in 20
and     n3 = 2          -- one in 25
;
```

We start with the standard rules we learned for B-trees. The index on `n1` has 20 distinct values and 60 leaf blocks, `n3` has 25 distinct values and 63 leaf blocks; both indexes have a `blevel` of 1, which can therefore be ignored. The cost of scanning the indexes to find the relevant bitmaps will be `ceiling(60/20) + ceiling(63/25)`: a total of 6, which we then scale up to 6.6 because of the bitmap multiplier of 1.1.

If the data were truly random, each of our queries would return one row in 500 (20 * 25), which means 20 rows out of the available 10,000—so the easy bit of the arithmetic tells us that we should see a plan with a cardinality of 20.

Using the 80/20 approximation, 4 of those rows will be widely scattered (which in this case means four separate blocks), and 16 of them will be clustered; but we have 9 rows per block on average, which means we need an extra two blocks for those 16 rows—for a total of six blocks.

So we predict that the total cost will be `round(6.6 + 6)` = 13. Running the query with `autotrace` enabled we see the following:

```
Execution Plan (9.2.0.6 version)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=13 Card=20 Bytes=340)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=13 Card=20 Bytes=340)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP AND
   4    3         BITMAP INDEX (SINGLE VALUE) OF 'T1_I3'
   5    3         BITMAP INDEX (SINGLE VALUE) OF 'T1_I1'
```

For confirmation, we can check the `10053 trace` to find the following:

```
Access path: index (equal)
      Index: T1_I1
  TABLE: T1
      RSC_CPU: 0   RSC_IO: 3
  IX_SEL:  5.0000e-002  TB_SEL:  5.0000e-002
  Access path: index (equal)
      Index: T1_I3
  TABLE: T1
      RSC_CPU: 0   RSC_IO: 3
  IX_SEL:  4.0000e-002  TB_SEL:  4.0000e-002
******** Bitmap access path accepted ********
Cost: 13 Cost_io: 13 Cost_cpu: 0.000000 Selectivity: 0.002000
Not believed to be index-only.
  BEST_CST: 12.87  PATH: 20  Degree:  1
```

With a *best cost* of 12.87, we can see that our estimate of 12.66 wasn't perfect, but our figures for the index were correct, and our estimate did come pretty close. (The optimizer seems to have allowed a cost of 6.27 rather than 6 for the visit to the table—maybe this is where the funny treatment of the `db_file_multiblock_read_count` appears.)

A little detail to note when checking the execution plans that use `bitmap and`: the indexes seem to be arranged in order of selectivity, so that the most selective (best eliminator) goes first. This is probably echoed down into the execution engine, as it may allow the execution engine to reduce the number of bitmap index fragments it has to expand and compare.

In a similar vein, if we execute this:

```
select
        small_vc
from
        t1
where
        n2 = 2          -- one in 20, clustered data
and     n4 = 2          -- one in 25, clustered data
;
```

we get an execution plan from `autotrace` as follows:

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=8 Card=20 Bytes=340)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=8 Card=20 Bytes=340)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP AND
   4    3         BITMAP INDEX (SINGLE VALUE) OF 'T1_I4'
   5    3         BITMAP INDEX (SINGLE VALUE) OF 'T1_I2'
```

In this case, the final cost is 8, a little lower because of the size of the indexes. We still expect to acquire 20 rows from the table, so our estimate for the table-related cost is still 6, but the indexes are so small that the cost of visiting each index is just one block per index, for a total of 2, which is then scaled up to 2.2; and indeed we see the following in the last lines of the `10053 trace`:

```
******** Bitmap access path accepted ********
Cost: 8 Cost_io: 8 Cost_cpu: 0.000000 Selectivity: 0.002000
Not believed to be index-only.
  BEST_CST: 8.47  PATH: 20  Degree:  1
```

Note, by the way, that in this fragment of the trace file, 8.47 = 2.2 + 6.27, and in the previous fragment we have 12.87 = 6.6 + 6.27. The optimizer really is multiplying the index component by 1.1, and is giving a nonintegral value to visiting table blocks.

Again, though, the 80/20 approximation used here starts to break down if pressed too hard, with diverging results as data sizes increase and the number of indexes used goes up.

Low Cardinality

If you have several bitmap indexes on a table, then Oracle is capable of using as many of them as is necessary to make the query run efficiently, but won't necessarily use all the ones that appear to be relevant. This leads to an important question: when does Oracle decide that there is no point in using one more bitmap index? The answer is (approximately, and ignoring the effect of bugs) when the cost of scanning one more set of leaf blocks is larger than the reduction in the number of table blocks that will be hit.

For example, assume you have a table that lists six attributes for people: sex (2 values), color of eyes (3 values), color of hair (7 values), home town (31 values), age band (47 values), and work classification (79 values). Now consider the test case in script `bitmap_cost_03.sql` in the online code suite, which builds a table of 800MB (107,543 blocks) holding details of 36 million people, with a query for rows where

```
        sex_code = 1
and     eye_code = 1
and     hair_code = 1
and     town_code = 15
and     age_code = 25
and     work_code = 40
```

If you had built individual bitmap indexes on all six columns, what is the probability that Oracle would use all six of them for this query? Answer—almost certainly zero. In fact, in my test case, Oracle used the three higher precision indexes, and ignored the three low precision indexes.

A COMMON MISCONCEPTION

The classic example for explaining bitmap indexes is the column holding a *male / female* flag. Unfortunately, it's an example of a bitmap index that I would not expect to see used for general queries.

Ironically, if you want to query this table to count the number of males and females, there are versions of Oracle where it might be quicker to build, query, and drop an index than it would be to run an aggregate query against the table directly. That's only an observation, by the way, not a strategic directive—building an index on the fly is generally not a good idea. In the versions of Oracle that did not support query rewrite, the index might have been useful to answer queries that were simply counting occurrences, but now a summary table declared as a materialized view would probably be the more sensible option.

Why is a column with a very small number of distinct values often a bad choice for a bitmap index? Let's review the preceding example (the script to rebuild the entire test may take about 30 minutes to run, and needs about 1.2GB of space). The six indexes should have the statistics shown in Table 8-4 under 9.2.0.6.

Table 8-4. *Index Statistics from a 36,000,000 Row Example*

Column	blevel	leaf_blocks	distinct_keys	leaf_blocks / distinct_keys (plus blevel)
sex_code	2	1,340	2	670 (672)
eye_code	2	2,010	3	670 (672)
hair_code	2	4,690	7	670 (672)
town_code	2	5,022	31	162 (164)
age_code	2	5,241	47	112 (114)
work_code	2	5,688	79	72 (74)

If you work out the average number of rows identified by just the town_code, age_code, and work_code specified in the preceding example, you will see that the optimizer will decide that these three columns are sufficient to restrict table visits to 36,000,000 / (31 * 47 * 79) = 313 rows. So with the worst possible scattering of data (rather than the assumed 80/20 split), the work needed to acquire these rows would be 313 table block visits.

Even if the next best index (hair_code with seven distinct values) reduced the number of table visits from a worst-case 313 down to 0, it wouldn't be worth using the index because it would require visiting an extra 672 index blocks—far more than the number of table block visits that would be saved.

As a rough test, I used the dbms_stats package to modify the leaf_block count for the hair_code index, and I had to bring it down to exactly 1,127 blocks—so that leaf_blocks / distinct_keys was down to 161—before the optimizer considered it worthwhile using it. (If the table had been sorted on hair_code to start with, the index would have held 675 leaf blocks rather than 4,690 leaf blocks, so the test was a reasonable one.)

Given the time it took to build the data, hacking the statistics with the dbms_stats package (see script hack_stats.sql in the online code source) a couple of dozen times to find the break point seemed a reasonable idea. Although it conveniently showed how the optimizer is very cunning in making its choice of bitmap indexes, this example also highlighted more deficiencies in my approximation of the costing algorithm.

The optimizer chose just the three indexes mentioned and reported the total cost of the query as 314; but if you check the sum of (blevel + leaf_blocks * *effective index selectivity*) for those indexes it comes to 352 (164 + 114 + 74). The final cost is less than the sum of its parts—and that's before you multiply by the 1.1 scaling factor and add on the cost of visiting the table!

It took me some time to spot what was going on, but when I used the dbms_stats package to modify the blevel of one of the used indexes, the change in cost was out of proportion to the change I had made to the blevel. This led me to decide that the final reported cost ignored the statistics of two of the indexes, and used the values from the cheapest index three times.

```
Approximate cost (target 314) =
 74  * 1.1 * 3       +          (cheapest index, times three, scaled)
 0.8 * 313 / 335     +          (80% of the rows, packed at 335 rows per block)
 0.2 * 313           =          (20% of the rows, scattered into individual blocks)
 244.2 + 62.6 + 0.75 =  307.55  (An error of 2.1%)
```

The strategy of always using the values from cheapest index looks to me like a bug, but it could be deliberate. However, it does have some odd side effects. When I used the `dbms_stats.set_index_stats` procedure to change the number of `leaf_blocks` recorded for the `hair_code` index from 4,690 down to (exactly) 1,127, the optimizer decided to use it without needing a hint, and did a four-index `bitmap and`.

However, when I ran this modified test, the reported cost of using the fourth index to resolve the query was 336, whereas the reported cost of using the three indexes had previously been only 314. In other words, the optimizer had voluntarily selected an extra index despite the fact that this *increased* the cost of the query—and the optimizer is always supposed to select the cheapest plan unless hinted otherwise.

When I saw this anomaly, I decided that it was a bit risky to assume that it was a genuine effect, rather than a strange consequence of a slightly hacked data dictionary—so I rebuilt the test case, sorting the data on the `hair_code` column, and tried again (see script `bitmap_cost_03a.sql` in the online code suite). This had exactly the same effect as hacking the statistics. The optimizer selected four indexes for the execution plan, even though it reported the three index option as cheaper when I dropped the fourth index or disabled it with a `no_index()` hint. There is clearly something wrong in the code for costing (or perhaps reporting the cost of) bitmap indexes.

As a quick check of the theory that very low cardinality indexes are not generally useful, I reran the original query a few more times, with various hints to disable some of the indexes. The results, shown in Table 8-5, did not entirely conform to my expectations.

Table 8-5. *Cost Variations As We Disable Selected Indexes*

Hint	Indexes Used	Cost
`no_index(t1 i4)`	`i6, i5, i3, i2 (work, age, hair, eyes)`	428
`no_index(t1 i5)`	`i6, i4, i3, i2, i1 (work, town, hair, eyes, sex)`	485
`no_index(t1 i5 i1)`	`i6, i4, i3, i2 (work, town, hair, eyes)`	481

As you can see, as soon as I disable the index on `age_code` (index `i5` with 47 distinct values), the optimizer decided to use the index on `sex_code` (index `i1` with just two distinct values). Despite my earlier comments that a *male/female* index would not generally be used, in this case it was, possibly because it had been given the cost of index `i6`, the `work_code` index, and not its own cost.

Notice, however, that when we disable the `sex_code` index, the calculated cost of the query becomes *lower*. In fact, the actual resource consumption for running the query was also significantly lower, taking 2,607 logical I/Os when the bad index was used, and only 2,285 when it was ignored.

Of course, there could be a rationale for making this choice—and this may be embedded in some part of the calculation that I haven't worked out yet. Bitmap indexes tend to be very small compared to the table they index; for example, my biggest index is less than 6,000 blocks on a table of more than 100,000 blocks. You may have enough memory to cache all your critical bitmap indexes. In those circumstances, it might make sense for the optimizer to trade a large number of logical I/Os on bitmap indexes against a small number of physical I/Os against the table. In my test case, the increase from 2,285 logical I/Os to 2,607 logical I/Os would have allowed Oracle to decrease the number of table block visits from 16 to 8. That extra 320 logical I/Os doesn't seem such a bad deal against a reduction in 8 physical I/Os.

There was one other change in behavior that appeared from the big test case—the reported cost of the query did not change much as I altered `db_file_multiblock_read_count`. Perhaps this is because the effect is very small once the data sets get large. Perhaps the error introduced by the (possible) bug in the handling of multiple indexes is so significant that the (possible) bug relating to `db_file_multiblock_read_count` is hidden. Perhaps it's because there is some special code for small data sets. In any case, it was nice to discover that for large data sets (which is where you tend to use bitmap indexes used, anyway), the calculations seem to be a little more stable than they are for smaller data sets.

Null Columns

It is always important to tell Oracle as much about your data as possible—`not null` columns are particularly important, as they can make a big difference to the possibilities open to the optimizer. Bitmap indexes are a good case in point. Consider the following query, based on the original sample data set, and note the `not equal` of the first condition (this is an extract from the script `bitmap_cost_04.sql` in the online code suite):

```
select
        small_vc
from
        t1
where
        n1 != 2
and     n3  = 2
;
```

Depending on the exact details of the table definition, you would find that there are two possible execution plans for this query:

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1'
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP MINUS
   4    3         BITMAP INDEX (SINGLE VALUE) OF 'T1_I3'
   5    3         BITMAP INDEX (SINGLE VALUE) OF 'T1_I1'
```

and

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1'
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP MINUS
   4    3         BITMAP MINUS
   5    4           BITMAP INDEX (SINGLE VALUE) OF 'T1_I3'
   6    4           BITMAP INDEX (SINGLE VALUE) OF 'T1_I1'
   7    3         BITMAP INDEX (SINGLE VALUE) OF 'T1_I1
```

Note, especially, the occurrence of two `bitmap minus` operations in the second plan. There are two of them because bitmap indexes include entries for completely `null` keys.

BITMAP MINUS

To perform a `bitmap minus`, Oracle takes the second bitmap and negates it—changing ones to zeros, and zeros to ones. The `bitmap minus` operation can then be performed as a `bitmap and` using this negated bitmap.

If we have declared `n1` to be `not null`, Oracle can resolve the query by finding the bitmap for the predicate `n1 = 2`, negating it, and then using the result in a `bitmap and` with the bitmap for `n3 = 2`, giving the first execution plan.

If we have not declared `n1` as `not null`, then the bitmap we get from the first step will include bits for rows where `n1` is `null`—so Oracle has to acquire the bitmap for `n1 is null`, negate it, and perform a further `bitmap and` with the intermediate result.

This extra step will require some work at run time. So if you know that a column is never supposed to be `null`, here is another reason for including that little bit of information in the table definition.

You will notice that I have not reported the costs in the execution plans—this was a deliberate omission designed to avoid confusion. 8*i* behaves quite reasonably and costs the first plan (with a single `bitmap minus` step) as cheaper than the second plan (with two `bitmap minus` steps).

Unfortunately, 9*i* and 10*g* both seemed to think that doing a second `bitmap minus` step would make the query significantly *cheaper* than doing just the one `bitmap minus` step. This could be a reasonable assumption, but for the fact that the statistics indicate that there were no rows with a null for the relevant column, so there would be little chance of reducing the number of visits to the table. Again, we seem to have some circumstances where the algorithms for bitmap costing are not entirely correct.

There are other problems with `null` values and bitmap indexes. Currently there seems to be a bug with the `bitmap or` mechanism that loses track of the number of `nulls` in a column when two bitmap indexes are **OR**'ed together. (See script `bitmap_or.sql` in the online code suite.)

```
create table t1
as
with generator as (
        select  --+ materialize
                rownum         id
        from    all_objects
        where   rownum <= 3000
)
select
        /*+ ordered use_nl(v2) */
        decode( mod(rownum-1,1000), 0, rownum - 1, null )  n1,
        decode( mod(rownum-1,1000), 0, rownum - 1, null )  n2,
        lpad(rownum-1,10,'0')                              small_vc
from
        generator        v1,
        generator        v2
where
        rownum <= 1000000
;

create bitmap index t1_i1 on t1(n1);
create bitmap index t1_i2 on t1(n2);

--        Collect statistics using dbms_stats here

select
        small_vc
from
        t1
where
         n1 = 50000
;

select
        small_vc
from
        t1
where
         n1 = 50000
or       n2 = 50000
/*
        (n1 = 50000 and n1 is not null)
or      (n2 = 50000 and n2 is not null)
*/
;
```

In this example, there are exactly 1,000 rows where column n1 is not null, and it has exactly 1,000 distinct values, and Oracle can see this in the statistics collected by the gather_table_stats() call. Column n2 is defined to be identical to column n1.

In the first query, the optimizer will determine that n1 has a density of 1/1,000, with a total of 1,000 non-null rows, and decide that the cardinality will be 1. If you changed n1 to n2 in this query, the calculated cardinality would be the same.

But when you run the query with the predicate

```
where   n1 = 50000 or n2 = 50000
```

the computed cardinality is 1,999. Oracle seems to have applied the density to the total number of rows in the table, and not allowed for the nulls. So (using the standard formula for an and of two predicates) we have 1/1,000 of the million rows accepted for the n1 predicate, plus 1/1,000 of the million rows accepted for the n2 predicate; minus one row in the overlap.

Add one is not null predicate, and the computed cardinality drops to 1,000; add the second is not null predicate, and the cardinality drops to the (correct) value of 1.

CPU Costing

Any discussion of bitmap index costing would be incomplete without some notes on what happens when you invoke CPU costing.

Given that CPU costing is a strategic feature, it's a good idea to be prepared for the day when you have to make the change. And if you had to guess, you might assume that CPU costing will have more impact on execution plans using bitmap indexes (and the **B-tree to bitmap conversion**) because of the work that must be done to convert between bits and rowids.

We'll use the same system statistics as we did in the Chapter 4 (see script bitmap_cost_05.sql in the online code suite):

```
alter session set "_optimizer_cost_model" = cpu;
begin
        dbms_stats.set_system_stats('MBRC',8);
        dbms_stats.set_system_stats('MREADTIM',20);
        dbms_stats.set_system_stats('SREADTIM',10);
        dbms_stats.set_system_stats('CPUSPEED',350);
end;
/

alter system flush shared_pool;
```

With these figures in place, we can run a couple of the queries that we used earlier on in the chapter against our original data set, and see how things change. Here, for example, are the execution plans against column n4 (clustered data, 25 different values, bitmap index) and n6 (clustered data, 25 different values, B-tree index), with their original cost, and with CPU costing—all plans are from 9.2.0.6:

```
Execution Plan - bitmap index on clustered column
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=114 Card=400 Bytes=5600)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=114 Card=400 Bytes=5600)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP INDEX (SINGLE VALUE) OF 'T1_I4'

Execution Plan - bitmap index on scattered column with CPU costing
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=128 Card=400 Bytes=5600)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=128 Card=400 Bytes=5600)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP INDEX (SINGLE VALUE) OF 'T1_I4'

Execution Plan - B-tree index on clustered column
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=54 Card=400 Bytes=5600)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=54 Card=400 Bytes=5600)
   2    1     INDEX (RANGE SCAN) OF 'T1_I6' (NON-UNIQUE) (Cost=9 Card=400)

Execution Plan - B-tree index on clustered column with CPU costing
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=55 Card=400 Bytes=5600)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=55 Card=400 Bytes=5600)
   2    1     INDEX (RANGE SCAN) OF 'T1_I6' (NON-UNIQUE) (Cost=10 Card=400)
```

It looks as if our prediction is correct. The cost of the bitmap query has gone up by 14, the cost of the B-tree query has gone up by just 1. Of course, we know that the B-tree query cost has been calculated on the basis of visiting about 45 table blocks, compared to the bitmap query cost that is based on visiting about 100 table blocks—but what Oracle actually does by the time it gets to the table is the same irrespective of whether it got there from a B-tree or a bitmap index. So most of the difference in cost must come from the bitmap somehow ... or so we assume!

And this is where (a) `autotrace` really shows up its deficiencies, and (b) we get a nasty surprise. If we do a proper `explain plan`, we get separate `io_cost` and `cpu_cost` columns (we could also look at the `10053 trace` for even finer detail).

- The B-tree cost of 55 comes from `io_cost` = 54, `cpu_cost` component = 1 (`cpu_cost` = 577,401).
- The bitmap cost of 128 comes from `io_cost` = 127, `cpu_cost` component = 1 (`cpu_cost` = 1,063,986).

Two things stand out in these figures: first, the actual CPU cost has roughly doubled, which corresponds roughly to the factor of 2 in the estimated number of I/Os (from 54 to 114)—the work involved in acquiring the right to access a buffered block is CPU-intensive. But the CPU required to handle the bitmap expansion is apparently not considered to be dramatically different from the CPU required to binary chop through an index block.

Second, and more significantly, the code for calculating the I/O cost of bitmap indexes gives you different figures for I/O when CPU costing is enabled—the original cost for our example was 114, but it's just gone up to 127. In general, the cost seems to go up when CPU costing is enabled, and this could have two significant effects. First, when you enable CPU costing, some of your queries may suffer a dramatic change of execution plans because different execution plans suddenly appear to be cheaper. Alternatively, some of your execution plans may look unchanged at first sight (it's easy to miss the sudden appearance of an extra bitmap index), but do more work because Oracle is actually using an extra bitmap index at a point where it had previously not seemed to be worthwhile. (Of course, given the buggy behavior described earlier, using an extra index might actually make some queries go faster.)

But the really extraordinary thing about the effects of enabling CPU costing with bitmap indexes is that the figures are still affected by changes to the `db_file_multiblock_read_count`. This really seems like a bug, when compared with the costing of B-tree indexes, where CPU costing switches to using the `mbrc` and `mreadtim` values for any calculations based on multiblock reads. If it's a bug, it will be fixed, of course, and when it's fixed, some execution plans will change.

Interesting Cases

I have pulled together a few of the slightly more obscure facets of bitmap indexes into this section simply to make sure you are aware of the options. In all cases, though, the arithmetic is basically nothing new, so I won't be spending much time describing the details.

Multicolumn Indexes

All the examples in the chapter have been single column indexes, which is probably a realistic reflection of most production systems. Since the most significant benefit of using bitmap indexes comes from the way that you can combine them in an arbitrary fashion, there is rarely any point in precombining columns inside an index.

However, you may have a couple of columns that are always queried simultaneously. For example, you may have a couple of columns that are in some way dependent on each other, or possibly hold critical status values.

Depending on the way in which the different values for the columns are distributed through the table, you may find that a single bitmap index on a pair of columns is actually smaller than the sum of the sizes of the two individual bitmap indexes. In such cases, it might be a very good idea to create a multicolumn bitmap index.

The arithmetic involved with multicolumn indexes is no different from the stuff you have already seen. You simply apply the Wolfgang Breitling formula (which I showed you in Chapter 4) to predicates that are available on the index to work out how many of the index leaf blocks you are going to visit, and then to work out what fraction of the table you will have to visit. (So that's just using what you learned about ordinary B-tree indexes.) Then you combine the table selectivities you get from each index to work out the final fraction of the table that you are going to visit, work out the number of rows, and apply the standard 80/20 split for bitmap indexes.

Bitmap Join Indexes

One of the interesting enhancements to bitmap indexes in 9*i* was the addition of the **bitmap join index**—an index that could hold entries for the rows in one table, but had key values that were taken from another table (or tables). For example (see script `bitmap_cost_06.sql` in the online code suite):

```
create bitmap index fct_dim_name on fact_table(dim.dim_name)
from
        dim_table        dim,
        fact_table       fct
where
        dim.id = fct.dim_id
;

create bitmap index fct_dim_par on fact_table(dim.par_name)
from
        dim_table        dim,
        fact_table       fct
where
        dim.id = fct.dim_id
;
```

These index definitions demonstrate two potential benefits:

- The first example gives us an index on a very large fact table that uses a long dimension name—which did not, however, have to be stored millions of times in the fact table.
- The second example shows us an index that can access the fact table based on a query against an **attribute** of a dimension table that (as the column name suggests) may have far fewer distinct values than the dimension ID, and therefore may be a much smaller, more desirable, index. (We assume that this is also an attribute that the users frequently use to identify and summarize the data.)

Personally, I am not convinced that the bitmap join index adds a lot of value once you have managed to use simple bitmap indexes well—but I can imagine there would be a few cases where the technology can be used effectively.

Having got through the details of why and how you create a bitmap join index, though, the arithmetic involved is unchanged. (Although the optimizer is still allowed to consider the option for doing a join, rather than using the index.) Consider this query:

```
select
        count(fct.id)
from
        dim_table        dim,
        fact_table       fct
where
        dim.par_name = 'Parent_001'
and     fct.dim_id = dim.id
;
```

```
Execution Plan (9.2.0.6 - autotrace)
----------------------------------------------------------
SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2149 Card=1 Bytes=9)
 SORT (AGGREGATE)
   TABLE ACCESS (BY INDEX ROWID) OF 'FACT_TABLE' (Cost=2149 Card=10000 Bytes=90000)
     BITMAP CONVERSION (TO ROWIDS)
       BITMAP INDEX (SINGLE VALUE) OF 'FCT_DIM_PAR'
```

As you can see, Oracle has decided that this query will return 10,000 rows (`card=10000`). Taking the usual 80/20 split, this gives us 2,000 rows assumed to be widely scattered, and 8,000 rows tightly packed. The table in question consists of 1,000,000 rows and 30,303 blocks—so 33 rows per blocks.

The data therefore requires 2,000 + 8,000/33 = 2,242 block visits—which is a little bit over the top for the cost of 2,149 that has been reported, especially when you add in roughly another 7 for the index block visits. However, it falls inside the range of values you can get by adjusting the `db_file_multiblock_read_count`, so I'm not too worried by the difference.

In case you're wondering, even in 9*i* the optimizer will calculate the cost of the two-table join before examining the cost of this single table access path.

Bitmap Transformations

The final note on the arithmetic of bitmap indexes covers the two transformations that can take place involving bitmaps. The basis of both transformations is something that we've been looking at all through this chapter—the frequently occurring execution plan line `bitmap conversion (to rowids)`.

A bitmap index is essentially a (cunningly packaged) two-dimensional array of ones and zeros. Each column in the array corresponds to one of the distinct values for the index key, each row in the array corresponds to a specific row location in the table. Oracle has a simple piece of arithmetic it can use to say, "Row X in the array corresponds to the Nth row of the Mth block of the table; conversely, the Pth row of the Qth block in the table corresponds to row Z in the array."

The arithmetic that translates an array entry into table entry is the *bitmap conversion* arithmetic and it can go either way, *at any point* in the execution plan. Normally, we only see it being used to convert from the array to the table, `bitmap conversion (to rowids)`, at the very last moment before we acquire data from the table—but there are other options.

One of these options is quite well known, and actually caused a few problems in the upgrade from 8*i* to 9*i*: the **B-tree to bitmap conversion**, reported in a plan as `bitmap conversion (from rowids)`. For example:

```
select
        small_vc
from
        t1
where
        n1 = 33
and     n2 = 21
;
```

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
SELECT STATEMENT Optimizer=ALL_ROWS (Cost=206 Card=400 Bytes=6800)
  TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=206 Card=400 Bytes=6800)
    BITMAP CONVERSION (TO ROWIDS)
      BITMAP AND
        BITMAP CONVERSION (FROM ROWIDS)
          INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=41)
        BITMAP CONVERSION (FROM ROWIDS)
          INDEX (RANGE SCAN) OF 'T1_I2' (NON-UNIQUE) (Cost=41)
```

In this example (see script `bitmap_cost_07.sql` in the online code suite), we have two B-tree indexes on the table `t1`. The optimizer has decided that neither of the two B-tree indexes (`t1_i1` on column `n1`, and `t1_i2` on column `n2`) is very efficient as the single access path to the table, but it has calculated that the number of rows identified by combining the two predicates will be small.

So, the optimizer has worked out a path that uses index `t1_i1` to acquire the set of rowids where `n1 = 33`, and `t1_i2` to acquire the set of rowids where `n2 = 21`. But rowids can always be converted to bits in the bitmap array. So the optimizer converts each list of rowids to an array of bits using `bitmap conversion (from rowids)`—and then has two arrays that can be put through a `bitmap and`, exactly as if they had come from a pair of normal bitmap indexes. From there on, the plan is just like an ordinary bitmap plan—we convert the bits back into rowids using `bitmap conversion (to rowids)` and visit the table.

The arithmetic used by the optimizer simply puts together a few things we are already familiar with—the cost of an index range scan (before visiting the table), the combined selectivity of two predicates, and the bitmap-related 80/20 distribution rule.

- Both our indexes have 1,947 leaf blocks, a `blevel` of 2, and 50 distinct values—so the index component of the Wolfgang Breitling formula is 2 + `ceil(1947/50)` = 41.
- The selectivity of the two predicates is 1/50, so the combined selectivity is 1/2,500. Since there happen to be 1,000,000 rows in the sample table, the cardinality of the result is 1,000,000 / 2,500 = 400.
- According to the 80/20 rule, 320 rows will be packed, and the 80 rows will be scattered widely. The 1,000,000 rows in the test table cover 17,855 blocks—56 rows per block—for a total of 5.7 blocks of packed data, and 80 blocks of scattered data.
- The total cost should therefore be near (41 * 1.1 * 2) + 80 + 5.7 = 175.9.

Unfortunately, 175.9 isn't really all that close to the value of 206 that was reported. So it's back to the drawing board and a search for inspiration. After a few simple experiments, I started to play with the index `blevel`, using `dbms_stats.set_index_stats`, and found that when I increased the `blevel` of one of the indexes by 10, the cost of the query went up by 14. I followed this up by changing the number of `leaf_blocks` so that the cost of an index range scan went up by 10—sure enough, the cost of the query went up by 14.

So there seems to be a different fudge factor involved with B-tree to bitmap conversions. So based on my figures, the cost of this execution plan should be as follows:

```
   Sum(index range scan * 1.4)       +
      blocks for 20% scattered rows  +
         blocks for 80% packed rows  =
            41 * 1.4 * 2 + 80 + 5.7  = 200.5
```

This is a lot closer to our target of 206, and well within the limits of that ever-present variation we get by changing the value of `db_file_multiblock_read_count`, so I'm happy to leave the approximation there, and move on to the other bitmap transformation.

The second bitmap transformation is something I've only come across very recently (in fact, in a question on the AskTom web site, at `http://asktom.oracle.com`). Like many execution plans in Oracle, it was obvious after I saw it that I should have known that it could happen (a case of "20-20 hindsight"), it was just that I had never expected that it would happen. If Oracle can convert between bitmap entries and rowids, there is no technical reason why it shouldn't do so at any point in an execution plan, so the following execution plan is perfectly legal (see script `bitmap_cost_08.sql` in the online code suite):

```
select
        d1,
        count(*)
from
        t1
where
        n1 = 2
and     d1 between to_date('&m_today', 'DD-MON-YYYY')
               and to_date('&m_future','DD-MON-YYYY')
group by
        d1
;

Execution Plan (9.2.0.6)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=48 Card=4 Bytes=44)
1   0   SORT (GROUP BY) (Cost=48 Card=4 Bytes=44)
2   1     VIEW OF 'index$_join$_001' (Cost=41 Card=801 Bytes=8811)
3   2       HASH JOIN
4   3         BITMAP CONVERSION (TO ROWIDS)
5   4           BITMAP INDEX (RANGE SCAN) OF 'T1_D1'
6   3         BITMAP CONVERSION (TO ROWIDS)
7   6           BITMAP INDEX (SINGLE VALUE) OF 'T1_N1'
```

Note especially how we start with two bitmap indexes, acquire some leaf block data from each in turn, and then effectively turn the results into an in-memory B-tree index. Once we have two B-tree index sections, we can do an index hash join between them. This example, of course, is exactly the opposite to the previous example, where we started with B-tree indexes, acquired some leaf block data, and converted to in-memory bitmap indexes.

As ever, we can put together the stuff we have learned so far to work out how this plan achieves its final cost. And in this case, the bit we are most interested in is the working that gets us to the result of the hash join (the `index$_join$_001` view in line 2).

Whenever you have to work with problems like this, there are usually several different approaches you could take, and the easiest demonstration of what's going on here is to run the query with the `10053 trace event` enabled. When you examine the trace file, you find that Oracle is simply working out the usual index access cost to get at the leaf blocks of the two indexes, with no special scaling factor, and no little extra add-on to allow for the `bitmap conversion (to rowids)`. After that, the cost of the hash join is simply the normal costing for hash joins (which you will see in Chapter 12).

An alternative strategy to determine that Oracle is simply using the normal index costing is to use `dbms_stats.set_index_stats` to adjust the index statistics. Add 10 to the `blevel` of one index, and the total cost of the query goes up by 10; adjust the number of `leaf_blocks` in an index by an amount that should add 10 to the index cost, and it adds 10 to the total cost of the query.

Summary

Bitmap indexes lose information about data scattering, so the optimizer has to invent some numbers. As soon as the optimizer uses hard-coded constants in place of real information, it is inevitable that some of your queries will do the wrong thing.

I am still not sure of the exact formulae used by Oracle for costing bitmap access—there may even be bugs in the costing algorithms that make the costing unstable. I think the notes and approximations of this chapter should be sufficient to give you a reasonable idea of how the optimizer is going to behave, but there seems to be a surprising fudge factor that depends on the value of `db_file_multiblock_read_count`.

When you move from traditional costing to CPU costing, you may see some execution plans change dramatically, and others stay largely the same but run more slowly because an extra bitmap index has been used (perhaps unnecessarily) to filter data out.

When you combine bitmap indexes, the optimizer seems to report a cost based on the cost of just the cheapest relevant index instead of the cost of the indexes actually used. This has some odd side effects that may mean some queries do too much work because an inappropriate set of indexes has been picked.

It is possible that the apparent bugs in the calculations are actually a deliberate design choice that is supposed to incur high numbers of logical I/Os against bitmap indexes to save on small numbers of physical I/Os against tables. In effect, the costing model may be assuming that you have your bitmap indexes in a large `KEEP` pool and the corresponding tables into a small `RECYCLE` pool. (Warning: this comment is highly speculative, so don't depend on it.)

Keep a close eye on the patch list for any bugs relating to costing of bitmap indexes. Some fixes might have a serious impact on your databases' performance.

Test Cases

The files in the download for this chapter are shown in Table 8-6.

Table 8-6. *Chapter 8 Test Cases*

Script	Comments
bitmap_cost_01.sql	Basic script used in this chapter to build a table with six indexed columns
bitmap_mbrc.sql	Sample to show how db_file_multiblock_read_count affects bitmap costs
bitmap_cost_02.sql	Queries that combine two indexes on the base table from the script bitmap_cost_01.sql
bitmap_cost _03.sql	Script to generate a large table (800MB with 36 million rows) to demonstrate the general uselessness of bitmap indexes on columns of very low distinct cardinality
hack_stats.sql	Sample script showing how to make small changes to existing object-level statistics
bitmap_cost_03a.sql	Repeats bitmap_cost_03.sql, with the hair_code in sorted order
bitmap_cost_04.sql	Effects of nullable columns and bitmap MINUS
bitmap_or.sql	Demonstration of bug in bitmap OR
bitmap_cost_05.sql	Effects of CPU costing
bitmap_cost_06.sql	Example of bitmap join index
bitmap_cost_07.sql	Example of B-tree index to in-memory bitmap conversion plan
bitmap_cost_08.sql	Example of bitmap index to in-memory B-tree conversion plan
setenv.sql	Sets a standardized environment for SQL*Plus

CHAPTER 9

Query Transformation

Nine chapters into the book, and I still haven't got as far as a two-table join. And I'm going to avoid joins for just one more chapter while I discuss features such as **subqueries**, **view merging**, **unnesting**, and the **star transformation**.

The reason for examining some of the more subtle options before looking at joins is that the optimizer tries to restructure the SQL you write, generally turning it into a simpler form consisting of just a straight join between tables, before optimizing it. This means the first step in understanding how the optimizer evaluates an execution plan requires you to work out the structure of the SQL that is actually being optimized. Since this chapter is about transformation mechanisms, there won't be a lot about cost in it.

A key point to remember about some of these transformations and the strange consequences that you sometimes see is that the code driving the decision to transform your SQL is still partly rule-based (or, as they say in the manuals, *heuristically driven*). In one version of the database, you may see a particular execution plan appear because *that's what the rules say*; and then you upgrade and the optimizer works out the cost of two execution plans, one with the transformation and one without, and takes the cheaper one—which may not be the one you saw before the upgrade, and may not be faster.

It seems that a common life cycle of the internal code for a typical query transformation is as follows:

- **Beta-like state**: The internal code exists, and you can make it happen with a (possibly) hidden parameter or undocumented hint.
- **First official publication**: The internal code is enabled by default, but not costed, so the transformation always happens.
- **Final state**: The optimizer works out the cost of the original and the transformed SQL and takes the cheaper option. The hint is deprecated (as, for example, `hash_aj` has been in 10*g*).

In cases like this, you may have to fall back on the `10053 trace` to figure out which state the feature and its supporting code is currently in.

Getting Started

We'll start with an example to demonstrate the way in which sections of optimizer code can change. As ever, we stick with an 8KB block size in locally managed tablespaces, with 1MB uniform extent size, avoid ASSM, and disable system statistics to make the test case reproducible (see script `filter_cost_01.sql` in the online code suite).

```
create table emp(
        dept_no         not null,
        sal,
        emp_no          not null,
        padding,
        constraint e_pk primary key(emp_no)
)
as
with generator as (
        select  --+ materialize
                rownum                  id
        from    all_objects
        where   rownum <= 1000
)
select
        mod(rownum,6),
        rownum,
        rownum,
        rpad('x',60)
from
        generator        v1,
        generator        v2
where
        rownum <= 20000
;
```

In this script, I have created a table of 20,000 employees, scattered over six departments. Each employee has a different identifier, and a different salary. The code sample demonstrates the 9*i* feature of **subquery factoring** (*with subquery*), which I often use to generate a lot of rows from a row source that might otherwise not be large enough. (The 8*i* version of the script creates a scratch table.)

We now run a query to list all the employees who earn more than the average salary for their department, and check the execution plans that we get from `autotrace` in different versions of Oracle. However, to highlight an important point, the SQL uses the `no_unnest` hint to force the optimizer to use a particular execution plan, because in this case the change in the internal code produces a dramatic change in the predicted cost and cardinality of the execution plan.

```
select
        outer.*
from    emp outer
where   outer.sal > (
                select  /*+ no_unnest */
                        avg(inner.sal)
                from    emp inner
                where   inner.dept_no = outer.dept_no
        )
;

Execution Plan (8.1.7.4)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=33 Card=1000 Bytes=72000)
1    0   FILTER
2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=33 Card=1000 Bytes=72000)
3    1     SORT (AGGREGATE)
4    3       TABLE ACCESS (FULL) OF 'EMP' (Cost=33 Card=3334 Bytes=26672)

Execution Plan (9.2.0.6)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=35035 Card=1000 Bytes=72000)
1    0   FILTER
2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=1000 Bytes=72000)
3    1     SORT (AGGREGATE)
4    3       TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=3333 Bytes=26664)

Execution Plan (10.1.0.4)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=245 Card=167 Bytes=12024)
1    0   FILTER
2    1     TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=35 Card=20000 Bytes=1440000)
3    1     SORT (AGGREGATE)
4    3       TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=35 Card=3333 Bytes=26664)
```

You will notice that the cost of the simple tablescan on the `emp` table (line 2) changed from 33 in 8*i* to 35 in 9*i* and 10*g*. One unit of the difference appears because the hidden parameter `_tablescan_cost_plus_one` is set to `true` in the newer versions of Oracle, but `false` in the older version. The other unit is because the tables are slightly different sizes thanks to a change in the behavior of `initrans`.

In 9.2, the `initrans` storage parameter on a table defaults to 2 (even if you set it to 1, and even when the data dictionary claims that it is 1). Moreover, when you do a `create table as select`, the actual number of slots in the **interested transaction list** (**ITL**) of each of the initially created table blocks is three rather than the two dictated by `initrans`. In the example, this tiny difference is just enough to make the `emp` table just one block bigger, which just happens to increase the cost of scanning it by one.

You will notice that the execution plans are the same in all three cases—but the costs and cardinalities show some dramatic differences. Since the cost and cardinality of one line of an execution plan can dictate the optimizer's choice of join orders and join methods for the whole plan, you will appreciate that the changes in calculations used for this operation could have a significant difference for more complex queries as you upgrade your version of Oracle.

Look at the cost for the 8*i* execution plan—it looks as if the optimizer has simply forgotten to factor in the cost of the subquery at all, so the cost of the query is just the cost of the driving full tablescan.

The cost that 9*i* works out does, at least, appear to have some sort of rationale behind it. One component of the 35,035 is the cost of the driving tablescan (35) in line 2. According to the execution plan, the number of rows (`card =`) on the driving tablescan will be 1,000—so the subquery (which is also a tablescan) will apparently have to be executed 1,000 times: 1,000 * 35 = 35,000. Adding these two components, you get the required total of 35,035.

But what is the source of the 1,000 that the optimizer has estimated as the cardinality of the driving tablescan? Unfortunately, in another stage of the calculations, the optimizer has worked out that the cardinality of the *final* result set will be 1,000. This comes from the fact that the only predicate on the driving table is `salary > not yet known result of subquery`, and this predicate has been given the same 5% selectivity as `column > :bind_variable`. Note the circular argument, though: the end result will be 1,000 rows, so the optimizer has assumed that the subquery used to generate the end result will be executed 1,000 times.

Finally, we get to 10*g*—and see that the query cost is 245. We also note that in line 2 of the execution plan the optimizer reports the cardinality of the driving tablescan as 20,000 (the number of rows in the table) rather than 1,000. So where does the cost come from?

You can get the answer by working backward: 245 / 35 = 7, the cost is equivalent to seven tablescans of the `emp` table. Subtract one of these for the driving tablescan, and you can infer that the optimizer has decided that the total impact of the subquery will amount to six tablescans. Why? Because that's what the optimizer is predicting as the actual work the execution engine will have to do.

Remember that there were six departments, and we needed to find the average salary for each department—10*g* is smart enough to work out that it need only execute the subquery six times, building an in-memory reference table of results as it goes. (We will examine this behavior later on in the section "Filter Optimization.") You can even see a clue to this behavior in line 4 of the execution plan, where the optimizer does a full tablescan for the subquery, and reports a cardinality of 3,333—one sixth of the table.

The final cardinality of 167 was a bit of a surprise. But when I first saw it, it seemed too much of a coincidence that if you take 5% of 20,000 (remember that `column > :bind_variable` has a selectivity of 5%), and then divide by 6 (the number of departments) the answer is 167. So I repeated my test code with eight departments—and got a final cardinality of 125, which is 1,000 / 8. The optimizer is dividing out by a related but irrelevant factor when it performs the `filter` operation. (This is better than it could be—there is a hidden parameter in 10*g* called `_optimizer_correct_sq_selectivity`, which is set to `true`; change this to `false`, and the optimizer uses the bind-variable 5% value a second time to calculate the cardinality, bringing it down to 50, instead of using the grouping factor from the subquery.)

You will notice, of course, that whichever version of Oracle you are using, the optimizer has produced a resulting cardinality that is wrong. In many typical data sets, the number of rows with a value *greater than average* will be approximately half the rows (although one British union activist is in the annals of urban legend as stating that he would not rest until

every worker earned more than the average wage). Since we have 20,000 rows in our `emp` table, the optimizer's estimate of 1,000 rows in 8*i* and 9*i* is out by a factor of 10, and the estimate in 10*g* is out by a factor of 60. That's a problem with generic code, of course; the optimizer only sees a subquery, it doesn't "understand" the particularly special effect of the *average* function that happens to be embedded in that subquery.

Evolution

So we find from our experiments with filtering that 8*i* hasn't got a clue about costs, 9*i* uses a circular argument for costing, and 10*g* knows what's going on in terms of cost but does something about the cardinality that is most inappropriate for the specific example.

We will return to subqueries later in this chapter. The critical point I wanted to make with this particular example was that the area of query transformation shows a great deal of evolutionary change as you work through different versions of the optimizer.

Filtering

With the evolution of the optimizer, Oracle is increasingly likely to eliminate subqueries by using various transformations, perhaps making the `filter` operation we saw earlier nearly obsolete. Even so, there are cases where subqueries cannot, or perhaps should not, be transformed away, so it is worth discussing their treatment.

THE MEANING OF "FILTER"

The name `filter` is applied to many different types of operation: for example, table elimination in **partitioned views**, evaluation of the `having` clause in **aggregate queries**, and the evaluation of correlated subqueries shown in this chapter.

Until the `filter_predicates` column appeared in the 9*i* `plan_table`, it was sometimes difficult to decide exactly what a `filter` line in an execution plan was supposed to mean. Make sure you stay up to date when it comes to knowing what's in the `plan_table`. Sometimes a new version gives you extra information in new columns.

Consider the following SQL, taken from the script `push_subq.sql` in the online code suite:

```
select
        /*+ push_subq */
        par.small_vc1,
        chi.small_vc1
from
        parent        par,
        child         chi
where
        par.id1 between 100 and 200
and     chi.id1 = par.id1
and     exists (
```

```
                select
                        /*+ no_unnest */
                        null
                from    subtest         sub
                where
                        sub.small_vc1 = par.small_vc1
                and     sub.id1 = par.id1
                and     sub.small_vc2 >= '2'
        )
;
```

This joins two tables of a parent/child relationship—the data has been set up so that each parent row has eight related child rows. The query uses a subquery based on values found in the parent to eliminate some of the data. (The `no_unnest` hint in the extract is there to make 9*i* and 10*g* reproduce the default behavior of 8*i*.)

When the optimizer can't fold a subquery into the main body of a query, the test performed by the subquery occurs at a very late stage of execution. In this example, the subquery would normally take place after the join to the child table and (because of the way I have designed the example) there will be lots of unnecessary joins to child rows from parent rows that should have been discarded before the join took place, and the workload will be higher than necessary. (It is also possible that in more general cases, the subquery could execute much too often, but this may not occur because of the special `filter` optimization described later on in this chapter.)

In cases like this, you may want to override the default behavior. To do this, you can use the `push_subq` hint, as I have done in the example, to force the optimizer to generate an execution plan that applies the subquery at the earliest possibly moment.

With `autotrace` enabled to produce both the execution plan and the execution statistics, we can see how the plan and statistics change as we introduce the hint (I've only reported the one statistic that actually changed):

```
Execution Plan  (8.1.7.4 WITHOUT push_subq hint - subquery postponed)
----------------------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=22 Card=7 Bytes=126)
1    0   FILTER
2    1     NESTED LOOPS (Cost=22 Card=7 Bytes=126)
3    2       TABLE ACCESS (BY INDEX ROWID) OF 'PARENT' (Cost=4 Card=6 Bytes=54)
4    3         INDEX (RANGE SCAN) OF 'PAR_PK' (UNIQUE) (Cost=2 Card=6)
5    2       TABLE ACCESS (BY INDEX ROWID) OF 'CHILD' (Cost=3 Card=817 Bytes=7353)
6    5         INDEX (RANGE SCAN) OF 'CHI_PK' (UNIQUE) (Cost=2 Card=817)
7    1     TABLE ACCESS (BY INDEX ROWID) OF 'SUBTEST' (Cost=2 Card=1 Bytes=14)
8    7       INDEX (UNIQUE SCAN) OF 'SUB_PK' (UNIQUE) (Cost=1 Card=1)
```

```
Statistics
----------------------
1224  consistent gets


Execution Plan  (8.1.7.4 WITH push_subq hint - early subquery)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=22 Card=7 Bytes=126)
1    0   NESTED LOOPS (Cost=22 Card=7 Bytes=126)
2    1     TABLE ACCESS (BY INDEX ROWID) OF 'PARENT' (Cost=4 Card=6 Bytes=54)
3    2       INDEX (RANGE SCAN) OF 'PAR_PK' (UNIQUE) (Cost=2 Card=6)
4    2       TABLE ACCESS (BY INDEX ROWID) OF 'SUBTEST' (Cost=2 Card=1 Bytes=14)
5    4         INDEX (UNIQUE SCAN) OF 'SUB_PK' (UNIQUE) (Cost=1 Card=1)
6    1     TABLE ACCESS (BY INDEX ROWID) OF 'CHILD' (Cost=3 Card=817 Bytes=7353)
7    6       INDEX (RANGE SCAN) OF 'CHI_PK' (UNIQUE) (Cost=2 Card=817)

Statistics
----------------------
 320  consistent gets
```

As you can see, the number of logical I/Os has dropped significantly. If you run a more thorough analysis of the work done, you will also find that various other measures of the work done (CPU used, latch gets, `'buffer is pinned count'`) drop in a similar fashion—so the benefit you can see in the `consistent gets` is a real benefit. (The costs of the two execution plans for both 9*i* and 10*g* nearly reflect this drop although, as you can see from the preceding example, both execution plans have the same cost in 8*i*.)

Note how the first execution plan has a `filter` operation in line 1, with child operations in lines 2 and 7. This means that too many rows are joined in line 2, and then every row returned by the join is subject to the subquery test (although the `filter` optimization described later in this chapter made this step a negligible overhead).

In the second execution plan, there is no apparent `filter` operation, but you will notice that the `nested loop` at line 1 still has tables `parent` and `child` as its child operations (lines 2 and 6). However, the table called `subtest` now appears in line 4 as a child operation to line 2, which is the table access to table `parent`.

In fact, the execution plan is not telling us the whole truth. But you have to switch to 9*i* and look at the view `v$sql_plan_statistics` to work out what's really happening—and even then, the image you get of the execution plan is misleading. The following plan comes from `v$sql_plan` after running the query with the `push_subq` hint—but the second and third columns show the `last_starts` and `last_output_rows` figures from the dynamic performance view `v$sql_plan_statistics`:

```
ID Starts   Rows Plan (9.2.0.6 - v$sql_plan_statistics - with push_subq hint)
-- ------ ------ ------------------------------------------------------------
 0              SELECT STATEMENT (all_rows)
 1      1      8   TABLE ACCESS (analyzed) CHILD (by index rowid)
 2      1     10     NESTED LOOPS
 3      1      1       TABLE ACCESS (analyzed) PARENT (by index rowid)
 4      1    101         INDEX (analyzed) PAR_PK (range scan)
 5    101      1         TABLE ACCESS (analyzed) SUBTEST (by index rowid)
 6    101    101           INDEX (analyzed) SUB_PK (unique scan)
 7      1      8       INDEX (analyzed) CHI_PK (range scan)
```

Notice that line 6 is executed 101 times, returning 101 rows, and then line 5 is executed 101 times, returning just one row. But why is line 6 executed 101 times? Because line 4 returns 101 rows, and each row has to be checked by the subquery. The subquery is occurring at the earliest possible moment—which is when the *index leaf block* is being read—and only one index entry survives the `filter` test, leading to just one access to the table.

In principle, I think the execution plan ought to look like the following so that you can see more clearly what's really going on:

```
ID Starts     Rows Plan (hypothetical)
-- ------ -------- ---------------------------
 0                SELECT STATEMENT (all_rows)
 1      1        8   TABLE ACCESS (analyzed) CHILD (by index rowid)
 2      1       10     NESTED LOOPS
 3      1        1       TABLE ACCESS (analyzed) PARENT (by index rowid)
 4      1        1         FILTER
 5      1      101           INDEX (analyzed) PAR_PK (range scan)
 6    101        1           TABLE ACCESS (analyzed) SUBTEST (by index rowid)
 7    101      101             INDEX (analyzed) SUB_PK (unique scan)
 8      1        8       INDEX (analyzed) CHI_PK (range scan)
```

I'm not sure how many people would be happy with an execution plan that showed a line between a table and the index that was used to access the table though. So maybe that's why it has been suppressed. In fact, the default mechanisms that 9*i* and 10*g* use for this query are completely different from the mechanisms that 8*i* uses, so the whole problem of how and where to represent filters may become irrelevant in future releases anyway.

THE ORDERED_PREDICATES HINT

The `ordered_predicates` hint is sometimes mentioned in these circumstances. However, this does not allow the optimizer to perform a subquery before a join, its purpose is to control the order that predicates will be applied when the optimizer says, "I am now at table X, and have N single table predicates that are now relevant." The script `ord_pred.sql` in the online code suite demonstrates this.

In passing, it is commonly stated (and even appears in most versions of the *Oracle Performance and Tuning Guide*) that the `ordered_predicates` hint goes after the `where`. This is wrong—it goes after the `select` just like any other hint. The `ordered_predicates` hint is deprecated in 10*g*, unfortunately.

Filter Optimization

In the first example of this chapter—which selects all employees who earn more than the average for their department—I showed that 10*g* had worked out that it needed to execute the correlated subquery only six times, and had therefore worked out a suitable cost for the execution of the query.

In fact, the model used by 10*g* for the calculation reflects the run-time method that Oracle has been using for years—at least since the introduction of 8*i*. The execution plan looks as if the subquery will be executed once for each row returned from the driving table (which is what would have happened in 7.3.4, and is how the mechanism is still described in the 9.2 Performance Guide and Reference [Part A96533-01], page 2-13, where it says, "In this example, for every row meeting the condition of the outer query, the correlated *EXISTS* subquery is executed.") but the run-time code is actually much more economical than that.

As I pointed out earlier, Oracle will attempt to execute the subquery just six times, once per department, remembering each result for future use. For 8*i*, you can only infer this by running the query and looking at the session's statistics for the number of tablescans, logical I/Os, and rows scanned. From 9*i* (release 2) onward, you can run with the parameter `statistics_level` set to `all` and then query `v$sql_plan_statistics` directly for the number of times each line of the execution plan was started.

By building various test cases (for example, script `filter_cost_02.sql` in the online code suite), it is possible to work out what is probably happening in a subquery filter; and I believe that the runtime engine does the following each time it picks a value from the driving table:

```
if this is the first row selected from the driving table
      execute the subquery with this driving value
      retain the driving (input) and return (output) values as 'current values'
      set the 'current values' status to 'not yet stored'.
else
      if the driving value matches the input value from the 'current values'
            return the output value from the 'current values'
      else
            if the status of the 'current values' is 'not yet stored'
                  attempt to store the 'current values' in an in-memory hash-table
                  if a hash collision occurs
                        discard the 'current values'
                  end if
            end if
            probe the hash table using the new driving value
            if the new driving value is in the hash table
                  retrieve the stored return value from the in-memory hash table
                  retain these values as the 'current values'
                  set the 'current values' status to 'previously stored'
            else
                  execute the subquery with the new driving value
                  retain the driving and return (output) values as 'current values'
                  set the 'current values' status to 'not yet stored'.
            end if
```

```
            return the output value from the 'current values'
      end if
end if
```

The most important lesson to learn from this is that a reported execution plan is not necessarily a totally accurate representation of the mechanics of what is really going on. In fact, this is likely to be true in more cases than just the subquery filter—the layout of an execution plan is very clean and simple; it may not always be possible to express exactly what is really going to happen and keep that level of clarity and simplicity in the execution plan.

There are a couple of other interesting side issues in this example. Oracle limits the size of the in-memory hash table (presumably to stop excessive memory consumption in unlucky cases). In 8*i* and 9*i* the limit on the size of the hash table seems to be 256 entries, in 10*g* it seems to be 1,024.

This means the performance of a subquery filter can be affected by the number of different driving values that exist, the order in which they appear in the pass through the driving table, and the actual values. If the hash table is simply too small, or you have driving values that just happen to cause excessive collisions on the hash table, then you may execute the subquery far more frequently than is strictly necessary.

As a simple demonstration of this, try the following with the data created by the script for the first test in this chapter (see script `filter_cost_01a.sql` in the online code suite):

```
update emp
set dept_no = 67                -- first collision value for 9i
-- set dept_no = 432            -- first collision value for 10g
where rownum = 1
;

select
        /*+ no_merge(iv) */
        count(*)
from (
        select  outer.*
        from    emp   outer
        where   outer.sal > (
                        select  /*+ no_unnest */
                                avg(inner.sal)
                        from    emp inner
                        where   inner.dept_no = outer.dept_no
                )
)        iv
;
```

Since I want to execute this query, not just run it through `autotrace`, I've taken the original query and wrapped it in an inline view to count the result set. The `no_merge` hint is there to ensure that the optimizer does not try to get too clever and transform the query before doing the count. I want to make sure that this query does the same amount of work as the original query; the `count(*)` in the outer query is just there to keep the output small. In this form, the `no_merge()` hint references the alias of a view that appears in its `from` clause.

BLOCKING MERGE JOINS

The `no_merge` hint has occasionally been described in the literature (possibly even in the Oracle manuals at one stage) as something that stops the optimizer from doing a merge join. This is not correct. The `no_merge` hint stops the optimizer from trying to get too clever with views (stored and inline) and noncorrelated `in` subqueries. Technically, it stops **complex view merging** from taking place.

If you want to block a merge join, there is no direct hint until you get to 10*g* and the arrival of the `no_use_merge` hint.

Check how long the `count(*)` takes to complete before the update, and then try it after the update. On my 9*i* system running at 2.8 GHz, the query run time changed from 0.01 seconds to 14.22 seconds (all of it CPU).

The value 67 just happens to collide in the hash table with the value 0. The side effect of setting the `dept_no` of just the first row to 67 was that the average salary for `dept_no = 67` went into the hash table, blocking the option for Oracle to store the average salary for `dept_no = 0`, so the subquery got executed 3,332 times, once for every row with `dept_no = 0`. (Because the hash table changes size in 10*g*, the value 432 is the first value that collides with 0.) I can't help thinking that a statistically better strategy would be to replace old values when a collision occurs—but then I would just have to modify my code slightly to produce an example where that strategy would also burn excessive amounts of CPU.

You will also note that the algorithm allows for checking the next row against the previous row to avoid the need to search the hash table. This means you generally get a small CPU benefit if the driving rows happen to be in sorted order. Moreover, if you sort the driving rows, the number of distinct values becomes irrelevant because the subquery executes only when the driving value changes—which may give you a manual optimization option in special cases.

The fact that the actual run-time resource consumption of this execution plan can vary so dramatically with the number and order of different data items is not considered in the cost calculation. Of course, it's a little unfair to make that last comment—I don't think it would be possible to collect the necessary information in a cost-effective fashion, so some *typical scenario* has to be built into the code. On the other hand, it is important to recognize that this is one of those cases where cost and resource consumption may have very little to do with each other.

You will probably see simple subquery filtering less frequently from 9*i* onward because unnesting (controlled largely by the change in the default value of parameter `_unnest_subquery`) happens so much more frequently. However, as you have seen, the `filter` operation can be very efficient, and some people have found that when they upgraded to 9*i*, some of their SQL ran much more slowly because an efficient filter had been turned into a much less efficient unnested join. (If you hit this problem, think about using the `no_unnest` hint that I used in my sample code for 9*i* and 10*g*.) You should also remember that there are cases where the presence of multiple subqueries makes unnesting impossible—if you have cases like this, each remaining subquery will get its own hash table.

Scalar Subqueries

I have picked **scalar subqueries** as my next topic because their implementation is tied so closely to the implementation of the `filter` operation described previously. It would be interesting to

discover whether the filter optimization came about because of the work done by the developers to accommodate scalar subqueries, or whether the scalar subquery became viable because of the optimization method created for filtering subqueries.

Consider the following rewrite of our original query (see `scalar_sub_01.sql` in the online code suite):

```
select
        count(av_sal)
from (
        select  /*+ no_merge */
                outer.dept_no,
                outer.sal,
                outer.emp_no,
                outer.padding,
                (
                        select  avg(inner.sal)
                        from    emp  inner
                        where   inner.dept_no = outer.dept_no
                )       av_sal
        from    emp  outer
)
where
        sal > av_sal
;
```

In this query, I've written a (correlated) subquery in the `select` list of another query. In this context, we refer to this subquery as a scalar subquery, i.e., a query that returns a single value (one column and at most one row—and if the query return no rows, its return value is deemed to be a `null`). In this example, I have also demonstrated the alternative form of the `no_merge` hint—the hint is embedded in the view that should not be merged and so does not need to reference an alias.

Since the subquery is now a column in the main select list, you could easily assume that it had to be executed once for each row returned by the main `select` statement. However, the code behaves just like the filtering code—and when we check the execution plan, we discover why:

```
Execution Plan (autotrace 9.2.0.6)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=35035 Card=1 Bytes=26)
1    0   SORT (AGGREGATE)
2    1     VIEW (Cost=35035 Card=1000 Bytes=26000)
3    2       FILTER
4    3         TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=1000 Bytes=8000)
5    3         SORT (AGGREGATE)
6    5           TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=3333 Bytes=26664)
```

The bit in the middle has (apparently) been transformed to the very same filtered subquery that our first experiment used. Note the funny thing though—this execution plan is from 9*i*, and in the original experiment, I had to use a `no_unnest` hint to force the optimizer to use the preceding execution plan. In this case, the optimizer has transformed once (from a scalar subquery to a filter subquery) but has been unable to perform the transformation that would have unnested the filtered subquery. In principle, that's one of the effects of the `no_merge` hint, and some other queries I tried that were similar to the example did transform into hash joins when I removed the hint. However, none of them would transform in 10*g* (even when hinted with an `unnest` hint); moreover, the sample query actually crashed my session in 9.2.0.6 when I removed the `no_merge` hint. There are possibly a few refinements still needed in the code that handles scalar subqueries.

If you eliminate the `where` clause in the sample query, so that the subquery result is reported but not used in the predicate, you will find that (a) lines 3, 5, and 6 disappear—there is no clue that the subquery has even happened, the code is down to that of a single tablescan—and, (b) the work done by the query is still consistent with executing the scalar subquery just six times.

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=35 Card=1 Bytes=13)
1    0   SORT (AGGREGATE)
2    1     VIEW (Cost=35 Card=20000 Bytes=260000)
3    2       TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=60000)
```

However, to make the point more firmly that the same optimization really is in place, we could do something more dramatic (see `scalar_sub_02.sql` in the online code suite).

```
create or replace function get_dept_avg(i_dept in number)
return number deterministic
as
        m_av_sal        number;
begin
        select  avg(sal)
        into    m_av_sal
        from    emp
        where   dept_no = i_dept
        ;

        return m_av_sal;

end;
/

select
        count(av_sal)
```

```
from (
        select  /*+ no_merge */
                dept_no,
                sal,
                emp_no,
                padding,
                (select get_dept_avg(dept_no) from dual) av_sal
                -- get_dept_avg(dept_no)                 av_sal
        from    emp
)
;
```

I have included two possible lines in the final query—one that calls the function `get_dept_avg()` explicitly, and one that calls it by doing a scalar subquery that accesses the `dual` table.

DETERMINISTIC FUNCTIONS

A deterministic function, according to the requirements of Oracle, is one that is guaranteed to return the same result every time you give it the same set of inputs. If you use **function-based indexes** (or, as they should be described, indexes with **virtual columns**), then the virtual columns have to be defined using deterministic functions.

The point of the deterministic function is that if Oracle can determine that the current call to the function is using the same inputs as the previous call, then it can use the previous result and avoid the call—or so the manuals say. As far as I can tell, this feature of deterministic functions has never been implemented. But the capability can be emulated by scalar subqueries.

If you run the version of the query that uses the scalar subquery, it will run very quickly, executing the subquery just six times. If you run the version that uses the direct function call, it will take much longer to run, and call the function once for every row in the main body of the query. (The fast version will suffer from exactly the same hash-collision problem of the normal subquery if you change one of the `emp` rows to have `dept_no = 67`—or `432` for 10*g*).

This is a fantastic performance benefit. The only drawback is that you can't see the subquery at all in the execution plan, which is identical to the execution plan I showed earlier where the scalar subquery was reported as a column in the output but not used in the `where` clause.

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=35 Card=1 Bytes=13)
1    0   SORT (AGGREGATE)
2    1     VIEW (Cost=35 Card=20000 Bytes=260000)
3    2       TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=60000)
```

Note the complete absence of any clue that there may be anything more to this query than a simple full tablescan of the `emp` table (and the subsequent aggregation, of course).

The really wonderful thing about scalar subqueries, though, is that they make it very easy to find out more about the limitations on the hash table. Remember my comment in the section "Filter Optimizations" that the limit appeared to be 256 in 8*i* and 9*i*, and 1,024 in 10*g*. I had to do a bit of hard work to figure that one out (and it's not the whole story in 10*g* anyway). But look at what you can do with scalar subqueries:

```
select
        (select pack1.f_n(16) from dual) x
from
        dual
;
```

We have a scalar subquery that calls a packaged function with a single numeric input (see script `scalar_sub_03.sql` in the online code suite). The package is defined as follows—note particularly the **global variable** and the way that the two functions increment the global variable before returning their input:

```
create or replace package pack1 as

        g_ct        number(10) := 0;

        function f_n(i in number)   return number;
        function f_v(i in varchar2) return varchar2;

end;
/

create or replace package body pack1 as

function f_n(i in number)   return number
is
begin
        pack1.g_ct := pack1.g_ct + 1;
        return i;
end
;

function f_v(i in varchar2) return varchar2
is
begin
        pack1.g_ct := pack1.g_ct + 1;
        return i;
end
;

end;
/
```

If, instead of using the scalar subquery against table dual, we use it against a table that contains the number from 1 to 16,384 twice (such as the following), we can observe something interesting:

```
create table t1 as
with generator as (
        select  --+ materialize
                rownum         id
        from    all_objects
        where   rownum <= 3000
)
select
        rownum                      n1,
        lpad(rownum,16,'0')         v16,
        lpad(rownum,32,'0')         v32
from
        generator        v1,
        generator        v2
where
        rownum <= 16,384
;

insert /*+ append */ into t1
select * from t1
;

select
        count(distinct x)
from    (
                select  /*+ no_merge */
                        (select pack1.f_n(n1) from dual) x
                from
                        t1
        )
;
```

Since every value from 1 to 16,384 appears in the table (in numeric form, and two zero-padded character forms), when we run the scalar subquery, we are going to have lots of hash collisions on the hash table. But we are also going to fill the hash table fairly rapidly on the first pass through the 16,384 values, and then reuse the `N` values we have saved just once each on the second pass through the 16,384 values in the table. Consequently, by the end of the query, the value of the global variable `pack1.g_ct` will be short of the 32,768 rows in the table by exactly the size of the hash table.

So all we have to do is this:

```
execute pack1.g_ct := 0;

-- execute the query

execute dbms_output.put_line('Hash table size: ' || ( 32768 - pack1.g_ct))
```

and we will get the size of the hash table directly; and the answer is

8*i*	256
9*i*	256
10*g*	Depends on the function's input, output, and the value of parameter `_query_execution_cache_max_size`

You will notice that my package definition includes a function that accepts a character input and returns a character output. I wrote this so that I could check whether the hash table behaved differently when large character strings were involved. In 8*i* and 9*i*, there was no change in the number of entries in the hash table as I went through various options of mixing character and numeric inputs and outputs. But when I ran a version of the query that supplied the zero-padded, 32-character string, returning the same string,the change in 10g was dramatic:

```
execute pack1.g_ct := 0;

select
        count(distinct x)
from    (
                select  /*+ no_merge */
                        (select pack1.f_v(v32) from dual) x
                from
                        t1
        )
;

execute dbms_output.put_line('Hash table size: ' || ( 32768 - pack1.g_ct))

COUNT(DISTINCTX)
----------------
           16384

Hash table size: 16
```

As you can see, the hash table shrank to 16. By selecting a `substr()` of the function return, I could *increase* the size of the hash table by *decreasing* the size of the substring. By supplying character inputs to the function returning a number, I could *decrease* the size of the hash table by *increasing* the size of the input string. It seemed likely that the size of the hash table was controlled by a fixed memory limit, rather than an absolute number of hash entries.

Since the size of the hash table dropped to 16 when I used a function returning an unconstrained string—which really means `varchar2(4,000)`, I looked for any parameter with a default value of 64, or 65,536, and found the parameter `_query_execution_cache_max_size`. Sure enough,

you can increase the size of the hash table up to a limit of 16,384 entries (the size is always a power of two) by increasing this parameter.

The upshot of this series of test is that you may find after upgrading to 10*g* that some queries that include scalar subqueries or filter subqueries run more slowly—even though their execution plans do not change—because their input or output values are quite long. If this is the case, you have two manual options for improving the performance. Try to reduce the total size of the input and output (concatenating strings, explicit substrings), or change the session setting for the `_query_execution_cache_max_size` parameter (although, as ever, you should be aware that messing with hidden parameters is something that should not be done without approval from Oracle support, and isn't necessarily going to be a stable solution on the next upgrade).

Subquery Factoring

I don't have to say much about subquery factoring—you've seen it in action a few times in the book already, and every time you've seen it, I've used a hint to make the optimizer do what I want rather than taking a cost-based decision.

If we take the query I used to create the table for the first example in this chapter, and remove the hint, it looks like this:

```
with generator as (
        select
                rownum   id
        from    all_objects
        where   rownum <= 1000
)
select
        mod(rownum,6),
        rownum,
        rownum,
        rpad('x',60)
from
        generator        v1,
        generator        v2
where
        rownum <= 20000
;
```

The main body of the query refers (twice) to an object I have called `generator`, and the initial section of the query tells us how `generator` is defined. At run time the optimizer has a choice: it can substitute the defining text inline whenever it sees the name `generator`, or it can create a temporary table once to hold the results of running the query that defines `generator` and use that temporary table whenever it sees the name `generator`.

Two hints are available for use with subquery factoring: the `materialize` hint forces the optimizer to create the temporary table, and the `inline` hint forces the optimizer to replace the name with its defining text and optimize the resulting query.

Here are the two possible execution plans generated by the 10*g* `dbms_xplan utility` for the preceding query (see script `with_subq_01.sql` in the online code suite). To keep the plan short,

I have replaced the view `all_objects` in the definition of `generator` with the table `t1`, created as `select * from all_objects`.

```
Execution plan with the /*+ materialize */ hint:
-------------------------------------------------------------------------------------------
| Id  | Operation                  |Name                          |Rows  |Bytes |Cost  |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                              |20000 |      |   97 |
|   1 |  TEMP TABLE TRANSFORMATION |                              |      |      |      |
|   2 |   LOAD AS SELECT           |                              |      |      |      |
|*  3 |    COUNT STOPKEY           |                              |      |      |      |
|   4 |     TABLE ACCESS FULL      |T1                            |44666 |      |   95 |
|*  5 |   COUNT STOPKEY            |                              |      |      |      |
|   6 |    NESTED LOOPS            |                              |20000 |      |    2 |
|   7 |     VIEW                   |                              |    1 |      |    1 |
|   8 |      TABLE ACCESS FULL     |SYS_TEMP_0FD9D662D_543F90     |44666 |  567K|    2 |
|   9 |     VIEW                   |                              |20000 |      |    1 |
|  10 |      TABLE ACCESS FULL     |SYS_TEMP_0FD9D662D_543F90     |44666 |  567K|    2 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter(ROWNUM<=1000)
   5 - filter(ROWNUM<=20000)


Execution plan with the /*+ inline */ hint:
-------------------------------------------------------------------
| Id  | Operation             | Name | Rows  | Bytes | Cost  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |    11 |       |     3 |
|*  1 |  COUNT STOPKEY        |      |       |       |       |
|   2 |   NESTED LOOPS        |      |    11 |       |     3 |
|   3 |    VIEW               |      |     1 |       |     1 |
|*  4 |     COUNT STOPKEY     |      |       |       |       |
|   5 |      TABLE ACCESS FULL| T1   | 44666 |       |    95 |
|   6 |    VIEW               |      |    11 |       |     2 |
|*  7 |     COUNT STOPKEY     |      |       |       |       |
|   8 |      TABLE ACCESS FULL| T1   | 44666 |       |    95 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(ROWNUM<=20000)
   4 - filter(ROWNUM<=1000)
   7 - filter(ROWNUM<=1000)
```

If you check the Rows column from both execution plans, you will see that the numbers don't really make sense. Particularly, the effects of the rownum <= 1000 clauses have not echoed back into the column, and the second execution plan shows a final row count of 11, rather than the 20,000 we are aiming for.

Moreover, you will notice that the final cost of the execution plan with the inline hint is clearly incorrect—each of the two tablescans of t1 table cost 95—and yet the total cost of the query is reported as 3, although it should clearly be at least 190 (2 * 95).

In fact, even though the execution plan with the inline hint has been reported with the lower cost, it is the execution plan where the subquery has been materialized that is used when the query is left to run unhinted.

The reason for these anomalies is a consequence of the rownum predicate. If you check the 10053 trace for these queries, you will find that the optimizer (in 10*g* only) has switched into first_rows(n) optimization. In effect, because my final predicate is rownum <= 20000, the optimizer has run the calculations that would be used if I had included a first_rows(20000) hint in the query.

The trace file shows the optimizer making two passes at the calculations—one that works out the final cardinality that you would get by joining the full generator view to itself (in my case a total of 1,995,239,224 rows) and one that is based on a scaling factor of 0.000010023905893 (which is 20,000 / 1,995,239,224).

I suspect this means the execution plans are displaying some figures that show the full cost of an operation, and some figures that are allowing for the scaled cost of the operation. It is unfortunate that my rather convenient trick for generating data happens to have found a complicated error in the *presentation* of execution plans for subquery factoring.

A Little Entertainment

One of the nice things about subquery factoring is that it allows you to use a divide-and-conquer approach (also known as *peeling the onion*) to complicated problems. In the past, you may have used inline views with the no_merge hint to work your way to a solution—but the no_merge hint is a little restrictive because it does force the optimizer into instantiating intermediate data as temporary data sets, and can be overly aggressive about restricting the options for clever transformations.

With subquery factoring, you can write a complex query in simple pieces, and let the optimizer choose whether to generate the intermediate data sets or construct the more complex, expanded form of the query and optimize it.

The technique is one I used to solve the following puzzle that someone posed on the comp.databases.oracle.server newsgroup a couple of years ago; asking for a single SQL statement that produced the correct answer without preprogramming the answer into the SQL.

Two mathematicians meet at a college reunion and start to talk about their families. Being mathematicians, they cannot help making their conversation a little obscure.

Mathematician X: Do you have any children?

Mathematician Y: Yes, I have three daughters.

Mathematician X: And how old are they?

Mathematician Y: If you multiply their ages, the result is 36.

Mathematician X: That isn't enough information.

Mathematician Y: If you sum their ages, the result is the same as the number of people in this room.

Mathematician X (after glancing around the room): That still isn't enough information.

Mathematician Y: My oldest daughter has a pet hamster with a wooden leg.

Mathematician X: And are the two-year-olds identical twins?

So how can you write a SQL statement that derives the ages of the three girls, and the number of people in the room? You might like to work out the ages, and figure out how many people mathematician X could see in the room before you read any further.

As a first step, you need to know that the basic syntax for subquery factoring allows for expressions like

```
with
        alias1 as (subquery1),
        alias2 as (subquery2),
        ...
        aliasN as (subqueryN)
select
        ...
;
```

So let's build a solution to the mathematicians' puzzle (see script `with_subq_02.sql` in the online code suite).

First we note that there are three daughters whose ages multiply up to 36. We can assume that the ages are complete years, and therefore must have a minimum of 1 and a maximum of 36—so let's generate a table with 36 rows, one for each possible age:

```
with age_list as (
        select  rownum   age
        from    all_objects
        where   rownum <= 36
),
```

But there are three of them, so we need three copies of the table, and when you multiply the three ages together, the total is 36—and then we will be adding the ages together, so let's do that at the same time. Notice that the preceding section of SQL ended with a comma, and the following section simply starts with a new alias (I don't repeat the `with`):

```
product_check as (
        select
                age1.age                            as youngest,
                age2.age                            as middle,
                age3.age                            as oldest,
                age1.age + age2.age + age3.age      as summed,
                age1.age * age2.age * age3.age      as product
```

```
        from
                age_list   age1,
                age_list   age2,
                age_list   age3
        where
                age2.age                              >= age1.age
        and     age3.age                              >= age2.age
        and     age1.age * age2.age * age3.age         = 36
),
```

Note how I've eliminated permutations of each possible result by introducing the predicate `age2.age >= age1.age and age3.age >= age2.age`. This means if, for example, I've listed `(2,3,6)`, I don't also list `(6,3,2)`, `(3,2,6)`, and the other three permutations. At this point, we have all the possible arrangements of ages that multiply up to 36. If we selected from `product_check`, we would get the following output:

```
  YOUNGEST     MIDDLE     OLDEST     SUMMED    PRODUCT
---------- ---------- ---------- ---------- ----------
         1          1         36         38         36
         1          2         18         21         36
         1          3         12         16         36
         1          4          9         14         36
         1          6          6         13         36                **
         2          2          9         13         36                **
         2          3          6         11         36
         3          3          4         10         36
```

We now get told that the sum of the ages matches the number of people in the room—but mathematician Y looks around, counts the number of people in the room, and cannot deduce the ages! So the (unknown) number of people in the room must identify at least two rows in the `product_check` set—in other words, there must be at least two rows that sum to the same value (see the rows marked with an **). So let's narrow the output from `product_check` to just those rows without asking for them explicitly:

```
summed_check as (
select
        youngest, middle, oldest, summed, product
from
        (
        select
                youngest, middle, oldest, summed, product,
                count(*) over(partition by summed) ct
        from    product_check
        )
where       ct > 1
)
```

In this subquery, I've used the analytic version of the `count()` function, partitioning over the sum of ages (column `summed`). The count will be greater than one only for the sums that appear more than once—so at this point we finally say

```
select  *
from    summed_check
;

  YOUNGEST     MIDDLE     OLDEST     SUMMED    PRODUCT
---------- ---------- ---------- ---------- ----------
         1          6          6         13         36
         2          2          9         13         36
```

We now know (as mathematician Y would have observed by looking around) that there were 13 people in the room. But mathematician Y is still stuck with two rows—until he hears about the hamster (with the completely irrelevant wooden leg) and the problem is resolved by a common convention (in the UK, at least) about the ages of twins. The oldest girl has a pet hamster—but the first row doesn't have an oldest girl, only a youngest girl and two older twins. The second row has an oldest girl and two younger twins, so the final, complete query should be as follows:

```
with age_list as (
        select  rownum   age
        from    all_objects
        where   rownum <= 36
),
product_check as (
        select
                age1.age                            as youngest,
                age2.age                            as middle,
                age3.age                            as oldest,
                age1.age + age2.age + age3.age      as summed,
                age1.age * age2.age * age3.age      as product
        from
                age_list    age1,
                age_list    age2,
                age_list    age3
        where
                age2.age                        >= age1.age
        and     age3.age                        >= age2.age
        and     age1.age * age2.age * age3.age   = 36
,
summed_check as (
select
        youngest, middle, oldest, summed, product
```

```
from
        (
        select
                youngest, middle, oldest, summed, product,
                count(*) over(partition by summed)   ct
        from    product_check
        )
where   ct > 1
)
select  *
from    summed_check
where   oldest > middle
;
```

Of course, it's not really a serious piece of SQL (check out Tom Kyte's web site, `http://asktom.oracle.com`, for a demonstration of how to use this technology to arrange a golf tournament if you want a more practical example), but it does show the principles.

You will notice that in this example, I used each subquery in the subsequent subquery—this is just one of many possible options. There is a requirement that every named subquery has to be used at least once; otherwise you receive the following Oracle error:

```
ORA-32035: unreferenced query name defined in WITH clause
```

But apart from this requirement, you can be very flexible with the named subqueries. My trivial example of generating a large data set used the same subquery twice—in general, any level of subquery could use many copies of any, or even all, of the subqueries in the levels above it. (In DB2, a subquery can even reference itself, but that feature is not yet available in Oracle.)

Complex View Merging

In Chapter 1, you saw an example of complex view merging (see script `view_merge_01.sql` in the online code suite). We created an aggregate view over a table, and then joined that view to another table:

```
create or replace view avg_val_view
as
select
        id_par,
        avg(val)        avg_val_t1
from    t2
group by
        id_par
;

select
        /* no_merge(avg_val_view) */         --  include a + to block merging.
        t1.vc1,
        avg_val_t1
```

```
from
        t1,
        avg_val_view
where
        t1.vc2 = lpad(18,32)
and     avg_val_view.id_par = t1.id_par
;
```

Faced with this query, the optimizer has two options available to it to optimize the query: create a result set for the aggregate view and then join it to the base table, or expand the view out to join the two base tables and then aggregate the join:

```
Execution Plan (9.2.0.6 hinted to instantiate the view)
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=15 Card=1 Bytes=95)
  HASH JOIN (Cost=15 Card=1 Bytes=95)
    TABLE ACCESS (FULL) OF 'T1' (Cost=2 Card=1 Bytes=69)
    VIEW OF 'AVG_VAL_VIEW' (Cost=12 Card=32 Bytes=832)
      SORT (GROUP BY) (Cost=12 Card=32 Bytes=224)
        TABLE ACCESS (FULL) OF 'T2' (Cost=5 Card=1024 Bytes=7168)

Execution Plan (9.2.0.6 allowed to merge the view)
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=14 Card=23 Bytes=1909)
  SORT (GROUP BY) (Cost=14 Card=23 Bytes=1909)
    HASH JOIN (Cost=8 Card=32 Bytes=2656)
      TABLE ACCESS (FULL) OF 'T1' (Cost=2 Card=1 Bytes=76)
      TABLE ACCESS (FULL) OF 'T2' (Cost=5 Card=1024 Bytes=7168)
```

Checking the `10053 trace` file (see Chapter 14 for more on the CBO trace file) for the unhinted query in 8*i*, you would find the following two `general plans` sections (from which I've listed just the headlines—typically each of the `general plans` starts with a new `Join order[1]`). This shows you that 8*i* considered only the option to create the aggregate view and join to the first table. You will also notice that when evaluating strategies for creating the aggregate view, the first of the **general plans** includes a recosting step to consider the use of a convenient index on table `t2` to calculate the aggregate without performing a sort—and this leads to an extra `Join order[1]`.

```
Join order[1]: T2 [T2]
****** Recost for ORDER BY (using join row order) *******
Join order[1]: T2 [T2]

Join order[1]: T1 [T1] AVG_VAL_VIEW [AVG_VAL_VIEW]
Join order[2]: AVG_VAL_VIEW [AVG_VAL_VIEW] T1 [T1]
```

Checking the trace file for the unhinted query in 9*i*, you would find the following single `general plans` section. After the upgrade (in which the default value for parameter `_complex_view_merging` changed from `false` to `true`), the optimizer considers only the option for merging the view and performing the join before aggregating. You will notice there is no reference to

recosting—it isn't relevant under this strategy. When you add the `no_merge()` hint, the trace file nearly switches back to the 8*i* version, although it doesn't try recosting for the index on `t2`—and that's an interesting little detail in its own right.

```
Join order[1]:  T1[T1]#0  T2[T2]#1
Join order[2]:  T2[T2]#1  T1[T1]#0
```

Checking the trace file for the unhinted query in 10*g*, you would find the following four `general plans` sections. And in this trace, you can see that the optimizer has calculated the cost of both options, and then selected the cheaper one. In fact, there were two sets of calculations for joining the two base tables. The only difference between the last two sets of calculations seemed to be the absence of a cost for carrying the join column—and I haven't been able to work out the significance of what that represents.

```
Join order[1]:  T2[T2]#0

Join order[1]:  T1[T1]#0  AVG_VAL_VIEW[AVG_VAL_VIEW]#1
Join order[2]:  AVG_VAL_VIEW[AVG_VAL_VIEW]#1  T1[T1]#0

Join order[1]:  T1[T1]#0  T2[T2]#1
Join order[2]:  T2[T2]#1  T1[T1]#0

Join order[1]:  T1[T1]#0  T2[T2]#1
Join order[2]:  T2[T2]#1  T1[T1]#0
```

Pushing Predicates

Some restrictions exist on view merging, and these vary from version to version of Oracle. However, even when the optimizer is not allowed to merge a view into the main body of a query, there are cases in some nested loop joins where the optimizer can push a **join predicate** into a view. This results in many small view instantiations, rather than one large view instantiation. Although I have seen pushed predicates being reported in execution plans from time to time, I found it quite hard to create a realistic example when I needed one, so I've fallen back on a limitation in the optimizer to demonstrate the feature (see `push_pred.sql` in the online code suite). The most likely case where pushing predicates will occur is when you do outer joins into multitable views:

```
create or replace view v1 as
select
        t2.id1,
        t2.id2,
        t3.small_vc,
        t3.padding
from
        t2,
        t3
where
        t3.id1 = t2.id1
```

```
and     t3.id2 = t2.id2
;

select
        t1.*,
        v1.*
from
        t1,
        v1
where
        t1.n1 = 5
and     t1.id1 between 10 and 50
and     v1.id1(+) = t1.id1
;
```

The execution plan for this query shows the predicate pushing in action. In this case, I have used the 9*i* `dbms_xplan` package to show the plan, as the `filter_predicates` and `access_predicates` columns of the `plan_table` are important.

```
-------------------------------------------------------------------------------
| Id  | Operation                      |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |             |     1 |   240 |     5 |
|   1 |  NESTED LOOPS OUTER            |             |     1 |   240 |     5 |
|*  2 |   TABLE ACCESS BY INDEX ROWID  | T1          |     1 |   119 |     3 |
|*  3 |    INDEX RANGE SCAN            | T1_PK       |    42 |       |     2 |
|   4 |   VIEW PUSHED PREDICATE        | V1          |     1 |   121 |     2 |
|*  5 |    FILTER                      |             |       |       |       |
|   6 |     NESTED LOOPS               |             |     1 |   129 |     3 |
|*  7 |      INDEX RANGE SCAN          | T2_PK       |     1 |     9 |     2 |
|   8 |      TABLE ACCESS BY INDEX ROWID| T3         |     1 |   120 |     1 |
|*  9 |       INDEX UNIQUE SCAN        | T3_PK       |     1 |       |       |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("T1"."N1"=5)
   3 - access("T1"."ID1">=10 AND "T1"."ID1"<=50)

   5 - filter("T1"."ID1"<=50 AND "T1"."ID1">=10)

   7 - access("T2"."ID1"="T1"."ID1")
       filter("T2"."ID1">=10 AND "T2"."ID1"<=50)

   9 - access("T3"."ID1"="T2"."ID1" AND "T3"."ID2"="T2"."ID2")
       filter("T3"."ID1">=10 AND "T3"."ID1"<=50)
```

Note especially line 4, with the operation `view pushed predicate`. This is the critical line that tells you that the optimizer is pushing predicates.

You will also note, however, that lots of filter predicates are listed, and that there are four separate occurrences of filter predicates of the form `colX >= 10 and colX <= 50`. The filter predicates in lines 5, 7, and 9 are actually redundant and disappear in 10*g*—in fact, the `filter` operation in line 5 of the execution plan itself disappears completely in 10*g*. However, it is quite convenient that these predicates appeared in this example, as it allows me to point out that they are not there as the result of predicate pushing, rather they are there as a result of **transitive closure**, and could appear even in cases where you don't see a `pushed predicate` line. Technically, pushing predicates is about join predicates, which in this example is the `access` predicate of line 7.

General Subqueries

It is very easy to request information using language that translates into subqueries.

- Give me a breakdown of sales from the store that had the *highest profit in June*.
- Give me the phone number of all customers whose zip code is *in the Carlton advertising region*.
- List all employees who earn more than the *average for their department*.

Depending on the way the question is phrased and your familiarity with the way the underlying data is structured, you might take these English-language queries and pick one of several different ways to express them in SQL.

In some cases, you may decide to do a very direct translation from the English because the strategy for translation is very visible—perhaps there is an obvious sense of *do X and then do Y with the result* that hints at a subquery structure. For example, the third item invites you to think of a simple query to work out the average salary for a `department`, and then use it to filter a query against `employees`:

```
Step 1:
        select avg(salary)
        from employees
        where department = {X}
        ;

Step 2:
        select *
        from employees emp1
        where salary > (
                        select avg(salary)
                        from employees emp2
                        where emp2.department = emp1.department
               )
        ;
```

Conversely, you may decide that some request can be rephrased to produce a much simpler SQL statement, even if the English equivalent isn't quite so direct. Consider the Carlton example—you might very well start with a query against some table that allowed you to identify all the zip codes covered by the Carlton advertising region, and then use this as an input to a more complex query:

```
Step 1:
        select distinct zip_code from tableX where agent = 'CARLTON';

Step 2:
        select tel from customers where zip_code in (
                select /* distinct */ zip_code from tableX where agent = 'CARLTON'
        );
```

On the other hand, it is probably much easier to spot in this case that a simple join between tables would be a perfectly satisfactory and comprehensible way of acquiring the information, provided we knew that each zip code appeared no more than once per agent—and had included that fact as a constraint on the database. (Note the use of the `distinct` in the first of the two preceding queries. The mechanism of an `in` subquery makes this implicit, which is why I have commentedit out in the second query.)

```
select  tel
from
        tableX,
        customers
where
        tableX.agent = 'CARLTON'
and     customers.zip_code = tableX.zip_code
;
```

When we translate user requirements into SQL, we have a great deal of scope in how we express ourselves. Ideally, of course, we want to express the problem in the way that fits our natural language most easily while allowing the optimizer to operate the resulting query in the most efficient fashion.

■ **Caution** From time to time, I see cases where programmers have converted joins to subqueries, or subqueries to joins—and have produced a SQL statement that is not logically identical to the original. Be very careful with subqueries and rewrites.

Looking at things from the opposite view point: if we fire the following query at a general purpose RBDMS, what could it do to resolve it?

```
select * from tableX where z_code in ('A','B','C');
```

One option would be to scan every row from `tableX`, and check the column `z_code` to see if it was an `'A'`, `'B'`, or `'C'`—a filtering operation. Another option might be to make use of a

convenient index that makes it very cheap to "find all the 'A's, then find all the 'B's, then find all the 'C's"—a mechanism that I usually refer to as a *driving* operation.

The same possibility exists when the query becomes slightly more complex:

```
select * from tableX where z_code in (other query);
```

Should the RDBMS get all the rows from `tableX`, and operate the `other query` once for each row, or should it somehow try to operate the `other query` once and use its result set to drive into `tableX`?

If everything I've said in this section makes perfect sense to you, then you've understood all there is to know (from a conceptual point of view) about subquery transformation. There are ways to write SQL that are nice, easy, and intuitively obvious for the natural language speaker—there are ways to operate SQL that allows the database engine to perform efficiently. The better the optimizer is at translating between equivalent SQL statements, the less work you have to do to tailor your queries to the needs of the database engine. Subquery transformation is one of the growth areas in the Oracle optimization code aimed squarely at this issue.

Subquery Parameters

Just as an indication of how hard it is to keep track of what's going on with subqueries, and what special effects might, or might not, work, I've produced a little table of the parameters, Table 9-1, relating to subquery transformation. (I may have missed a few—sometimes the names and descriptions don't give much of a clue to purpose.)

Table 9-1. *Parameters for Handling Subqueries Keep Changing*

Name	8*i*	9*i*	10*g*	Description
_unnest_notexists_sq	n/a	single	n/a	Unnests `not exists` subquery with one or more tables if possible.
_unnest_subquery	false	true	true	Enables unnesting of correlated subqueries.
_ordered_semi-join	true	true	true	Enable ordered semi-join (exists) subquery.
_cost_equality_semi_join	n/a	true	true	Enables costing of equality semi-join (exists).
_always_anti_join	nested_loops	choose	choose	Always use this method for anti-join (not exists) when possible.
_always_semi_join	standard	choose	choose	Always use this method for semi-join (exists) when possible.
_optimizer_correct_sq_selectivity	n/a	n/a	true	Forces correct computation of subquery selectivity.
_optimizer_squ_bottomup	n/a	n/a	true	Enables unnesting of subqueries in a bottom-up manner.

Table 9-1. *Parameters for Handling Subqueries Keep Changing*

Name	8*i*	9*i*	10*g*	Description
_distinct_view_unnesting	n/a	n/a	false	Enables unnesting of in subquery into "select distinct" view.
_right_outer_hash_enable	n/a	n/a	true	Enables right outer (including semi and anti) hash join.
_remove_aggr_subquery	n/a	n/a	true	Enables removal of subsumed aggregate subqueries.

The number of parameters that didn't exist before 10*g* should give you some pause for thought—how many old rules of thumb are you going to have to forget regarding things you shouldn't do in SQL?

Then take a look at that parameter _unnest_notexists_sq. Why has it ceased to exist in 10*g*? Does this mean that the optimizer has only recently been enhanced to handle all possible not exists subqueries, and no longer needs a parameter to keep it in check? Or does it mean that the optimizer has decided to stop unnesting not exists subqueries? (There is an example in the section "Anti-join Anomaly" that shows the answer to this question is no.)

What about the parameter _cost_equality_semi_join, which appeared in 9*i*? Why are there two notes on MetaLink (258945.1 and 144967.1) that seem to state that 9*i* will unnest without costing, when this parameter suggests that there are some cases where the cost is examined? Perhaps it's because semi-joins (and anti-joins) aren't really considered to be examples of unnesting. Perhaps it's simply that even the MetaLink analysts have a hard time keeping up with all the changes.

Whatever this list says to you, it tells me that it's hard work keeping up with the new details and features that keep appearing in the optimizer—particularly in the area of subquery transformation. The whole area of subquery manipulation is one where I regularly have to remind myself that "I've never seen it" isn't the same as "it doesn't happen." It's also the area where transformed execution plans can be hardest to read (think about my hypothetical plan in the earlier discussion on filtering, with its missing filter line).

Categorization

Before going on, we ought to do a little categorization of subqueries, just so that we have some common understanding of the types of query that we will be looking at, and to get a broad-brush picture of the types of subquery where the options for transformation are (currently) restricted.

When dealing with problems of subqueries that are misbehaving, I tend to split them up into a few mechanical areas, as shown in Table 9-2. This isn't intended as a scientific, or even formal, categorization; it's just my personal way of reminding myself of the highlights of what to expect from different types of subquery.

Table 9-2. *A Rough Classification of Subquery Types*

Category	Characteristics
Correlated/ Noncorrelated	A correlated subquery references columns from an outer query block. Correlated subqueries can often be transformed into joins; noncorrelated subqueries have some chance of becoming driving subqueries.
Simple/Complex	Simple subqueries contain just a single table. Complex subqueries contain many tables, through either joins or subqueries of subqueries. There are things that the optimizer can do with simple subqueries that cannot be applied to complex subqueries.
Aggregate	If a simple (single table) subquery contains some aggregation, then there are some restrictions on how the optimizer may be able to transform them.
Single-row	A subquery that returns (at most) a single row—which often means that it can become a driving point in the query.
In/Exists	`in` subqueries can be rewritten as `exists` subqueries. These can then be transformed into semi-joins.
Not in/Not exists	`not in` subqueries can be rewritten as `not exists` subqueries. With certain restrictions, these can then be transformed into anti-joins. Critically, `not in` is not the opposite of `in`—and `null` columns cause problems.

You will note, of course, that a single subquery can easily fall across several of these categories—there is nothing stopping a correlated subquery from being an aggregate, returning at most one row for a nonexistence test. (But at least, when I see one like that, I know that I may have to think about it very carefully if it's not doing what I expect.)

In this volume, I'm going to stick to simple subqueries, play about a little bit with `in/exists`, `not in/not exists`, and say a little bit about unnesting, semi-joins, and anti-joins.

So let's go back to script `filter_cost_01.sql`, which started this chapter, and modify it to allow 9*i* to do what it wants with our very first test case. What does the execution plan look like? See script `unnest_cost_01.sql` in the online code suite.

```
Execution Plan (9.2.0.6 with no hints - an unnest occurs)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=100 Card=1000 Bytes=98000)
1    0   HASH JOIN (Cost=100 Card=1000 Bytes=98000)
2    1     VIEW OF 'VW_SQ_1' (Cost=64 Card=6 Bytes=156)
3    2       SORT (GROUP BY) (Cost=64 Card=6 Bytes=48)
4    3         TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=160000)
5    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)

Execution Plan (9.2.0.6 when we forced a FILTER)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=35035 Card=1000 Bytes=72000)
1    0   FILTER
2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=1000 Bytes=72000)
3    1     SORT (AGGREGATE)
4    3       TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=3333 Bytes=26664)
```

The optimizer has taken our original query and rewritten it to avoid the filter. In effect, the optimizer has executed the following statement (the `no_merge` hint is there just to make sure that Oracle doesn't decide to use some trick of complex view merging to expand the query, and do something completely different with it):

```
select
        /*+ ordered use_hash(outer) */
        outer.*
from
        (
                select
                        /*+ no_merge */
                        dept_no,
                        avg(inner.sal)     avg_sal
                from    emp inner
                group by
                        dept_no
        )    inner,
        emp  outer
where
        outer.dept_no = inner.dept_no
and     outer.sal > inner.avg_sal
;
```

Note the appearance of the line `view of vw_sq_1` in the execution plan. This is an indication that the optimizer has done some query unnesting. Alternative names for views you can get as a result of unnesting are `vw_nsq_1` and `vw_nso_1` (with a change in the numeric bit for complicated SQL that manages to produce more than one unnesting action).

Those of you who have read *Oracle Performance Tuning 101* by Gaja Vaidyanatha et al. (Osborne McGraw-Hill, 2001) will recall at this point that one of the authors' suggestions for dealing with certain types of subquery was to convert the subquery into an inline view and put it into the body of the query. This was such a good idea that the optimizer now does it automatically. It happens more or less inevitably in 9*i*, but is a cost-based decision in 10*g*.

The test case produces exactly the same final plan for 10*g* as it does for 9*i*, including the cardinality of 1,000 rather than the 167 that 10*g* produced when doing a filter (so the cardinality is still wrong with its *bind variable 5%*, but it is closer). However, when you look at the `10053 trace`, you find that 9*i* has produced just two `general plans` sections, whereas 10*g* has produced ten sections before producing a final course of action, suggesting that the decision to unnest was driven by the cost.

The first section in the 9*i* trace generated the strategy for instantiating the inline aggregate view, and the second section worked out how to join the aggregate view to the other table.

The trace file for 10*g* started with the same two sections, and then seemed to have a section calculating the effect of joining the two tables before doing an aggregation (which I thought should have been blocked by the `no_merge` hint), followed by various repetitions of very similar calculations. Lurking within the later calculations were two sections relating to costing the query using the old 8*i* `filter` option—which was more expensive than the unnesting option, and therefore ignored (although in a secondary experiment, I changed the data so that the `filter` was cheaper than the `unnest`, and the filter was chosen automatically).

Script unnest_cost_02.sql in the online code suite has an example where the optimizer in 10g seems to choose to unnest, even though the filter option has a lower cost. However, the chosen path is actually reported as a semi-join rather than a simple join after unnesting, so there may be some heuristic (i.e., rule) that blocks filters in favor of semi-joins. Examination of the 10053 trace file shows that only one possible execution method (a join) was considered—so there must have been a transformation applied before the option to use a filter has been considered. Hang on to your no_unnest hint—you may need it from time to time.

The script unnest_cost_01a.sql in the online code suite shows a couple of variations on the theme of the average salary question. The first is a noncorrelated single row subquery—instead of checking employees with a salary greater than the average for their department, we find the employees with a salary higher than the average for the company:

```
select
        outer.*
from
        emp outer
where
        outer.sal >
                (
                        select
                                avg(inner.sal)
                        from    emp inner
                )
;
```

As usual, 8*i* does its filtering thing, and forgets to allow for the cost of the subquery. In fact, it is barely possible to notice the difference between the execution plan for the original *better than average for the department* query and the modified query in 8*i*.

```
Execution Plan (8.1.7.4 autotrace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=34 Card=1000 Bytes=72000)
1  0   FILTER
2  1     TABLE ACCESS (FULL) OF 'EMP' (Cost=34 Card=1000 Bytes=72000)
3  1     SORT (AGGREGATE)
4  3       TABLE ACCESS (FULL) OF 'EMP' (Cost=34 Card=20000 Bytes=100000)
```

As with the filter execution for the correlated subquery we ran at the start of the chapter, we have an execution plan that appears to say we will scan the emp table and calculate the average once per row. The only difference between this plan and the plan for the correlated subquery is that the cardinality for the aggregation tablescan on line 4 is 20,000 (for the full table) rather than the 3,334 for each department reported in the first execution plan.

When we move on to 9*i*, we see the following changes:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=35 Card=1000 Bytes=72000)
1  0   TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=1000 Bytes=72000)
2  1     SORT (AGGREGATE)
3  2       TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=100000)
```

Look very carefully at this execution plan—the indenting is critical. Working from the bottom up, line 3 is a tablescan that feeds its result set to line 2, which sorts it for an aggregate function—in fact, there will be no real sorting, just a running sum and count to produce the company average. Line 2 feeds the average *just once* to line 1 as an unknown, but fixed, value. So line 1 uses the *5% for a bind variable* calculation to estimate that 1,000 rows out of the 20,000 will be returned by the tablescan.

Note the final cost in line 0. It's 35, the cost of just one tablescan—which means it's wrong. This changes in 10*g*, where the plan is identical, except for a final (correct) cost of 70.

Another variation on the averages theme (`unnest_cost_01a.sql` still) is to introduce an extra restriction, represented by an extra table. What if I am only interested in the results for a particular group of departments? I might end up with a (not very sensible) query like the following:

```
select
        outer.*
from
        emp outer
where
        outer.dept_no in (
                select  dept_no
                from    dept
                where   dept_group = 1
        )
and
        outer.sal > (
                select  avg(inner.sal)
                from    emp  inner
                where
                        inner.dept_no = outer.dept_no
                and     inner.dept_no in (
                                select  dept_no
                                from    dept
                                where   dept_group = 1
                        )
        )
;
```

In this case, the SQL selects only the people from the departments in group 1 from the full list of employees by using a noncorrelated subquery in both the outer query block and the inner query block. A more sensible approach, perhaps, would have entailed using a two-table join (assuming it was logically equivalent) in both the outer and inner query blocks.

Despite the silliness of the query, the optimizer copes well in 10*g*. I've used `dbms_xplan` in this example to help identify which copies of `emp` and `dept` in the original query correspond to which copies in the execution plan. But the help comes only from examining the predicate information section carefully. (A column called `instance` in the `plan_table` lets you identify multiple copies of the same table in a query very easily—unfortunately, none of the Oracle tools report it.)

```
-------------------------------------------------------------------
| Id  | Operation             | Name    | Rows  | Bytes | Cost  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT      |         |   500 | 51500 |    98 |
|*  1 |  HASH JOIN            |         |   500 | 51500 |    98 |
|   2 |   VIEW                | VW_SQ_1 |     6 |   156 |    59 |
|   3 |    SORT GROUP BY      |         |     6 |    78 |    59 |
|*  4 |     HASH JOIN         |         | 10000 |   126K|    38 |
|*  5 |      TABLE ACCESS FULL| DEPT    |     3 |    15 |     2 |
|   6 |      TABLE ACCESS FULL| EMP     | 20000 |   156K|    35 |
|*  7 |   HASH JOIN           |         | 10000 |   751K|    38 |
|*  8 |    TABLE ACCESS FULL  | DEPT    |     3 |    15 |     2 |
|   9 |    TABLE ACCESS FULL  | EMP     | 20000 |  1406K|    35 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("DEPT_NO"="OUTER"."DEPT_NO")
       filter("OUTER"."SAL">"VW_COL_1")
   4 - access("INNER"."DEPT_NO"="DEPT_NO")
   5 - filter("DEPT_GROUP"=1)
   7 - access("OUTER"."DEPT_NO"="DEPT_NO")
   8 - filter("DEPT_GROUP"=1)
```

Looking at this output, we can see that the optimizer has turned the outer subquery into a simple hash join in lines 7, 8, and 9. Looking at line 2, we can see that the optimizer has also unnested a subquery, which turns out to be our average salary subquery—and inside that subquery, the optimizer (at lines 4, 5, and 6) has also turned our silly inner subquery construct into a simple hash join.

Even 8*i* does something similar. It manages to convert both subqueries into hash joins, although it then uses its standard `filter` mechanism to deal with the rest of the query.

Unfortunately, 9*i* does something a little surprising—possibly trying to be too clever with the wrong dataset. This is the 9*i* execution plan:

```
-------------------------------------------------------------------------------
| Id  | Operation               |  Name       | Rows  | Bytes |TempSpc| Cost  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |             |   833K|    71M|       |   262K|
|*  1 |  FILTER                 |             |       |       |       |       |
|   2 |   SORT GROUP BY         |             |   833K|    71M|  1687M|   262K|
|*  3 |    HASH JOIN            |             |    16M|  1430M|       |   100 |
|   4 |     TABLE ACCESS FULL   | EMP         | 20000 |   156K|       |    35 |
|*  5 |     HASH JOIN           |             | 30000 |  2402K|       |    44 |
|   6 |      MERGE JOIN CARTESIAN|            |     9 |    90 |       |     8 |
|*  7 |       TABLE ACCESS FULL | DEPT        |     3 |    15 |       |     2 |
|   8 |       BUFFER SORT       |             |     3 |    15 |       |     6 |
|*  9 |        TABLE ACCESS FULL| DEPT        |     3 |    15 |       |     2 |
|  10 |      TABLE ACCESS FULL  | EMP         | 20000 |  1406K|       |    35 |
-------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("OUTER"."SAL">AVG("INNER"."SAL"))
   3 - access("INNER"."DEPT_NO"="OUTER"."DEPT_NO" AND
              "INNER"."DEPT_NO"="DEPT"."DEPT_NO")
   5 - access("OUTER"."DEPT_NO"="DEPT"."DEPT_NO")
   7 - filter("DEPT"."DEPT_GROUP"=1)
   9 - filter("DEPT"."DEPT_GROUP"=1)
```

I am not going to try to explain exactly what the optimizer has done here—critically, though, it obeyed the subquery unnesting directive (parameter `_unnest_subquery = true`) and unnested everything in sight, and then used complex view merging to try to find the optimum four-table join order, postponing the calculation of average salary to the last possible moment.

Suffice it to say that on test runs, 8*i* and 10*g* managed to complete the query in the proverbial subsecond response time, whereas 9*i* took 1 minute and 22 seconds of solid CPU on a machine running at 2.8 GHz.

When I set `_unnest_subquery` to `false`, 9*i* followed the 8*i* execution plan; when I set parameter `_complex_view_merging` to `false`, it followed the 10*g* execution plan. Conversely, when I set the 10*g* parameter `_optimizer_squ_bottomup` to `false`, 10*g* generated the disastrous 9*i* execution plan—even when I rewrote the query to turn the subqueries on `dept` to joins (and that was a surprise that I'm going to have to investigate one day).

Semi-Joins

Let's take a step backward from the complexity of the previous example, and focus instead on just the requirement to list all the employees from a specific group of departments. I'd like to write this (see script `semi_01.sql` in the online code suite) as follows:

```
Select  emp.*
from    emp
where   emp.dept_no in (
               select  dept.dept_no
               from    dept
               where   dept.dept_group = 1
        )
;
```

You will appreciate (based on an intuitive understanding of how employees and departments usually work) that this is probably a silly way to write this query, and a simple join would work better. Assuming I had designed the tables correctly—which really means insisting that departmental codes are unique within the department table—the optimizer could come to the same conclusion. This is the default execution plan in 9*i* when that critical condition is met:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=38 Card=10000 Bytes=770000)
1    0   HASH JOIN (Cost=38 Card=10000 Bytes=770000)
2    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=3 Bytes=15)
3    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)
```

The optimizer managed to turn the subquery approach into a simple join approach. If I hadn't created the dept table with the appropriate uniqueness constraint, the optimizer would still have managed to find the join by unnesting:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=44 Card=10000 Bytes=770000)
1    0   HASH JOIN (Cost=44 Card=10000 Bytes=770000)
2    1     SORT (UNIQUE)
3    2       TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=3 Bytes=15)
4    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)
```

In a very slight variation, 8*i* would have inserted one extra line to the plan, between the hash join and the sort, identifying the inline view from the unnest as vw_nso_1. If we had then blocked the unnesting (with the no_unnest hint), the optimizer would have fallen back to the filter mechanism:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=47 Card=3333 Bytes=239976)
1    0   FILTER
2    1     TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=35 Card=20000 Bytes=1440000)
3    1     TABLE ACCESS (FULL) OF 'DEPT' (TABLE) (Cost=2 Card=1 Bytes=5)
```

There is still one remaining option that the optimizer could play: the semi-join—a mechanism reserved exclusively for existence tests. Wait a moment, you say, this query doesn't have an existence test it has an in subquery. But an in can always be transformed into an exists, and once it has been transformed, the optimizer may choose to use a semi-join on it—and that semi-join could be a nested loop, merge, or hash semi-join. When hinted, these are the plans we get from 8*i* and 9*i*:

```
Execution Plan (9.2.0.6 autotrace with nl_sj hint in subquery)
----------------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=40035 Card=10000 Bytes=770000)
1    0   NESTED LOOPS (SEMI) (Cost=40035 Card=10000 Bytes=770000)
2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)
3    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=2 Bytes=10)

Execution Plan (9.2.0.6 autotrace with merge_sj hint in subquery)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=282 Card=10000 Bytes=770000)
   1    0   MERGE JOIN (SEMI) (Cost=282 Card=10000 Bytes=770000)
   2    1     SORT (JOIN) (Cost=274 Card=20000 Bytes=1440000)
   3    2       TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)
   4    1     SORT (UNIQUE) (Cost=8 Card=3 Bytes=15)
   5    4       TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=3 Bytes=15)
```

```
Execution Plan (9.2.0.6 autotrace with hash_sj hint in subquery)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=63 Card=10000 Bytes=770000)
   1    0   HASH JOIN (SEMI) (Cost=63 Card=10000 Bytes=770000)
   2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)
   3    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=3 Bytes=15)
```

So what is a semi-join? If we take our subquery example about finding employees in a set of departments, we might rewrite it as follows:

```
select  emp.*
from
        emp,
        dept
where
        dept.dept_no = emp.dept_no
and     dept.dept_group = 1
;
```

But there could be a logical problem with this rewrite. Unless the combination of (`dept_no, dept_group`)—or at least the `dept_no`—is constrained to be unique across the `dept` table, a simple join could produce multiple copies of each relevant `emp` row. If you are not very careful as you switch between joins and subqueries, you can get wrong answers.

This is where the semi-join comes in: it deals with cases where the suitable uniqueness constraint simply does not exist. A semi-join is like a simple join, but once a row from the outer table has joined to one row from the inner table, no further processing is done for that outer row. In its nested loop form, it's like having an inner loop that always stops after one success, thus saving resources.

Of course, as you have just seen, within the limitations of the semi-join, you can have nested loop, merge, or hash semi-joins—but if you look carefully at the 9*i* version of the hash semi-join, you will see that the join has operated *the wrong way round*. It has built the in-memory hash from the larger data set, and then probed it with the smaller data set. Semi-joins and anti-joins (like traditional outer joins) are constrained in 9*i* to operate in a very specific order—the table from the subquery has to appear *after* the table in the main query.

However, when we move to 10*g*, where an enhancement to the hash join code allows an **outer hash join** to operate in either direction, we find an interesting little change. The execution plan (in fact the default execution plan in my test case) is as follows:

```
Execution Plan (10.1.0.4 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=38 Card=10000 Bytes=770000)
1    0   HASH JOIN (RIGHT SEMI) (Cost=38 Card=10000 Bytes=770000)
2    1     TABLE ACCESS (FULL) OF 'DEPT' (TABLE) (Cost=2 Card=3 Bytes=15)
3    1     TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=35 Card=20000 Bytes=1440000)
```

Note the (`right semi`) in line one, compared to the (`semi`) in the 9*i* execution plan. Semi-joins and anti-joins, like outer joins, are no longer constrained to operate in a specific order in 10*g* if they are executed as hash joins, so the smaller dataset can always be the one chosen to

build the hash table. Note also that we are back to the original cost of the hash join we had when the dept_no was declared unique.

Anti-Joins

The semi-join can deal with exists subqueries (or in subqueries that get transformed into exists subqueries). There is another special operation—the anti-join—that has been created to deal with not exists (or not in) subqueries.

Given the very special piece of inside information that our departments fall into exactly two groups, we could identify employees in department group 1 by rewriting our peculiar query so that it selected employees who were *not* in department group 2 (see script anti_01.sql in the online code suite).

```
select  emp.*
from    emp
where   emp.dept_no not in (
               select  dept.dept_no
                from    dept
                where   dept.dept_group = 2
        )
;
```

The first thing we discover is that, no matter how we hint it, the optimizer will stubbornly produce the following plan if either the emp.dept_no or dept.dept_no is allowed to be null. Something is blocking any transformation if nulls are allowed (we will examine this in a minute).

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2035 Card=1000 Bytes=72000)
1    0   FILTER
2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=1000 Bytes=72000)
3    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=1 Bytes=5)
```

If we want the optimizer to be clever with not in subqueries, we need to ensure that the columns at both ends of the comparison may not be null. Of course, one way of doing this is to add a not null constraint to the two columns—but adding a predicate dept_no is not null at both ends of the query would also work.

Assuming that we do have the necessary not null constraints at both ends of the comparison, we get the following extra execution plans from 9*i* (in my case the hash anti-join appeared as the default action, the other two options had to be hinted):

```
Execution Plan (9.2.0.6 - with nl_aj hint in subquery)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=40035 Card=10000 Bytes=770000)
1    0   NESTED LOOPS (ANTI) (Cost=40035 Card=10000 Bytes=770000)
2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)
3    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=2 Bytes=10)
```

```
Execution Plan (9.2.0.6 - with merge_aj hint in subquery)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=282 Card=10000 Bytes=770000)
1    0   MERGE JOIN (ANTI) (Cost=282 Card=10000 Bytes=770000)
2    1     SORT (JOIN) (Cost=274 Card=20000 Bytes=1440000)
3    2       TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)
4    1     SORT (UNIQUE) (Cost=8 Card=3 Bytes=15)
5    4       TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=3 Bytes=15)

Execution Plan (9.2.0.6 - with hash_aj hint in subquery)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=63 Card=10000 Bytes=770000)
1    0   HASH JOIN (ANTI) (Cost=63 Card=10000 Bytes=770000)
2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)
3    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=3 Bytes=15)
```

And you notice once more in 9*i* that the hash join has been done the expensive way (hashing the larger table). Again, 10*g* comes to the rescue with the following execution plan where the table order has been reversed, and the (`anti`) has been replaced by (`right anti`):

```
Execution Plan (10.1.0.4 - with hash_aj hint in subquery)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=38 Card=10000 Bytes=770000)
1    0   HASH JOIN (RIGHT ANTI) (Cost=38 Card=10000 Bytes=770000)
2    1     TABLE ACCESS (FULL) OF 'DEPT' (TABLE) (Cost=2 Card=3 Bytes=15)
3    1     TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=35 Card=20000 Bytes=1440000)
```

We can explain the anti-join by thinking how we could rewrite the `not in` subquery as a join. The following query is an appropriate rewrite, and gives us the critical clue as to how the anti-join is behaving:

```
select
        emp.*
from
        emp,
        dept
where
        dept.dept_no(+) = emp.dept_no
and     dept.dept_group(+) = 2
and     dept.dept_no is null
;
```

The effect of this query is to join every employee to their department if their department is in group 2, but, because of the outer join, we also preserve every other employee row; then we discard all rows where the join actually found a department. Consequently, what we have left is a single copy of every employee who is not a member of a department in group 2, and (through our human awareness of the meaning of the data) is therefore a member of a department in group 1. Again, we have done too much work to get a result that could (in general) be wrong—especially if there is no suitable uniqueness constraint on the `dept` table. The anti-join is simply

the implementation of this outer join approach, except it discards an outer row the moment it finds a match (thus avoiding redundant joins to the inner table).

Anti-join Anomaly

In much the same way that the optimizer can transform `in` subqueries to `exists` subqueries, it can also transform `not in` subqueries into `not exists` subqueries. But sometimes it won't, for no apparent reason. Compare the following two queries—given the definition of the tables (see `book_subq.sql` in the online code suite) they are logically equivalent:

```
select  book_key
from    books
where   NOT EXISTS (
                    select  null
                    from    sales
                    where   sales.book_key = books.book_key
        )
;

select  book_key
from    books
where   book_key NOT IN (
                    select  book_key
                    from    sales
        )
;
```

In 10*g* the optimizer will use a hash anti-join to execute the first query. Moreover,10*g* will transform the second query into the first query and do a hash anti-join on that as well.

On the other hand, 9*i* behaves differently—and in a totally counterintuitive way. It will transform the second query into the first query, and execute a hash anti-join. But it won't perform a hash anti-join on the first query unless it is hinted (and the hash anti-join is much cheaper than the `filter` operation that appears by default). You may recall in the table of optimizer parameters earlier on that we saw a 9*i* parameter called `_unnest_notexists_sq`, with a default value of `single`. Strangely, this query seems to be described perfectly by that parameter—but the single existence subquery is not unnested.

ANTI-JOIN BUGS IN 9*i*

There are several bugs listed on MetaLink relating to anti-joins in 9*i*, typically reported as fixed in 10.2. Generally these bugs are of the *wrong results* category, so be a little cautious about execution plans that report anti-joins. It's worth getting a list of the current bugs with their **rediscovery information** just in case.

The moral of this story is that the optimizer can do all sorts of clever transformations between different ways of writing queries—so it's a nice idea to write the query in a way that makes sense to you as the first approach; but sometimes a transformation that you might

expect simply doesn't happen—so sometimes you have to be prepared to write your queries in a way that helps the optimizer along a bit.

Nulls and Not In

We still have to address the issue of nulls blocking the anti-join (and associated) transformations. There are two ways of understanding the issue. The first is the informal observation that the anti-join is approximately equivalent to the outer join show in the previous section—but the outer join does an equality comparison between the dept_no columns of the emp and dept tables, and nulls always produce problems when you start comparing them in any way other than is null or is not null.

The second, slightly more formal, approach is to note the comment in the SQL Reference manual that

```
colX not in ('A', 'B', 'C')
```

is equivalent to

```
colX != 'A' and colX != 'B' and colX != 'C'
```

The string of **AND**s means that every one of the individual conditions in the list must be checked and must evaluate to true for the entire expression to evaluate to true. If a single condition evaluates to false or null, then the expression is not true.

As an example of the errors that can occur if you forget this, script notin.sql in the online code suite sets up a case with the following results:

```
select * from t1 where n1 = 99;

        N1 V1
---------- --------------------
        99 Ninety-nine

1 row selected.

select * from t2 where n1 = 99;

no rows selected

select  *
from    t1
where   t1.n1 not in (
              select t2.n1 from t2
        )
;

no rows selected
```

The first query shows that there is a row in table t1 with n1 = 99. The second query shows that there are no corresponding rows with n1 = 99 in table t2. The last query shows that there

are no rows in table t1 that do not have a corresponding row in table t2—despite the fact that we've clearly just reported one.

The problem is that there is a row in table t2 where n1 is null—as soon as this occurs, the final query can never return any data because every row from table t1 that is tested gets to this null row and tests the condition t1.n1 = null (which is not the same as t1.n1 is null), which never returns true.

The solution, of course, is to include a not null predicate at both ends of the test:

```
select  *
from    t1
where   t1.n1 is not null
and     t1.n1 not in (
              select t2.n1 from t2
              where t2.n1 is not null
        )
;

        N1 V1
---------- --------------------
        99 Ninety-nine

1 row selected.
```

Because of the problem of comparisons with null, the optimizer is only able to transform the standard not in subquery if the not null predicates or constraints are in place. Otherwise, the only possible execution plan is the filter option; moreover, the clever optimization that we saw for the filter option is not (currently) implemented for the case where the filter is there to deal with a not in.

This is another reason for defining database constraints. If you know that a column should never be null, then tell the database about it, otherwise bad data (in the form of a null) may get into that column one day, and make any existing not in subqueries suddenly return the wrong answer for *no apparent reason*.

Just to complete the catalog, you might note that the (apparently opposite) expression

```
colX in ('A', 'B', 'C')
```

is equivalent to

```
colX = 'A' or colX = 'B' or colX = 'C'
```

The string of **OR**s means that just one of the individual tests in the list needs evaluate to true (after which Oracle need check no further) for the entire expression to evaluate to true. Apart from the benefit of being able to take a shortcut on the test, this does mean that the presence of a null somewhere in the test is not a problem. This shows us that, counterintuitively, the two operators in and not in are not exact opposites of each other.

The ordered Hint

As the optimizer becomes increasingly clever with subqueries, you may find that code that used to work efficiently suddenly starts to misbehave because you've told it to! Hints—even the

simplest ones—can be very dangerous, and here's an example showing how things can go badly wrong (see script `ordered.sql` in the online code suite). It starts with a query that needed a little bit of hinting in 8*i* to make the join between `t1` and `t3` take place in the right order:

```
select
        /*+ ordered push_subq */
        t1.v1
from
        t1, t3
where
        t3.n1 = t1.n1
and     exists (
                select        t2.id
                from                t2
                where                t2.n1 = 15
                and             t2.id = t1.id
        )
and     exists (
                select        t4.id
                from                t4
                where                t4.n1 = 15
                and             t4.id = t3.id
        )
;
```

This happens to work perfectly in 8*i*, with the optimizer producing the following plan:

```
Execution Plan (8.1.7.4)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=103 Card=1812 Bytes=45300)
   1    0   FILTER
   2    1     HASH JOIN (Cost=103 Card=1812 Bytes=45300)
   3    2       TABLE ACCESS (FULL) OF 'T1' (Cost=51 Card=1000 Bytes=16000)
   4    3         TABLE ACCESS (BY INDEX ROWID) OF 'T2' (Cost=2 Card=1 Bytes=8)
   5    4           INDEX (UNIQUE SCAN) OF 'T2_PK' (UNIQUE) (Cost=1 Card=1)
   6    2       TABLE ACCESS (FULL) OF 'T3' (Cost=51 Card=1000 Bytes=9000)
   7    1     TABLE ACCESS (BY INDEX ROWID) OF 'T4' (Cost=2 Card=1 Bytes=8)
   8    7       INDEX (UNIQUE SCAN) OF 'T4_PK' (UNIQUE) (Cost=1 Card=1)
```

Unfortunately, when you upgrade to 9*i*, the optimizer still obeys your hints—and you haven't put enough of them into the SQL to deal with the new, improved options that 9*i* uses. Remember that query transformation takes place before the other stages of optimization.

The optimizer will unnest the two subqueries in your original statement. Unfortunately, subquery unnesting seems to take place from the bottom up, and the resulting inline views are inserted from the top down in your `from` clause. This means the table order in the transformed query is now `t4`, `t2`, `t1`, `t3`—and then there's that ordered hint to apply! (The `push_subq` hint is ignored, because after 9*i* has finished its transformation, there are no outstanding subqueries to push.) So the new execution plan—thanks to your hinting—is as follows:

```
Execution Plan (9.2.0.6)
----------------------------------------------------------
 0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=36 Card=1 Bytes=41)
 1   0   NESTED LOOPS (Cost=36 Card=1 Bytes=41)
 2   1     NESTED LOOPS (Cost=29 Card=7 Bytes=224)
 3   2       MERGE JOIN (CARTESIAN) (Cost=22 Card=7 Bytes=112)
 4   3         SORT (UNIQUE)
 5   4           TABLE ACCESS (BY INDEX ROWID) OF 'T4' (Cost=4 Card=3 Bytes=24)
 6   5             INDEX (RANGE SCAN) OF 'T4_N1' (NON-UNIQUE) (Cost=1 Card=3)
 7   3         BUFFER (SORT) (Cost=18 Card=3 Bytes=24)
 8   7           SORT (UNIQUE)
 9   8             TABLE ACCESS (BY INDEX ROWID) OF 'T2' (Cost=4 Card=3 Bytes=24)
10   9               INDEX (RANGE SCAN) OF 'T2_N1' (NON-UNIQUE) (Cost=1 Card=3)
11   2       TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=1 Card=1 Bytes=16)
12  11         INDEX (UNIQUE SCAN) OF 'T1_PK' (UNIQUE)
13   1     TABLE ACCESS (BY INDEX ROWID) OF 'T3' (Cost=1 Card=1 Bytes=9)
14  13       INDEX (UNIQUE SCAN) OF 'T3_PK' (UNIQUE)
```

This is not (necessarily) a good thing—note particularly the **Cartesian** merge join between the two unrelated tables `t2` and `t4` that used to be in your subqueries. Don't blame Oracle for this one—you put in the hints, and hints are obeyed if they're legal. You just didn't put in enough hints to future-proof your code.

Star Transformation Joins

It is worth mentioning the star transformation briefly, as this is a fantastic example of how the optimizer can rewrite a query in a way that produces the same result set from the same set of tables while allowing a completely different order of activity to occur. Script `star_trans.sql` in the online code suite creates the following table:

```
create table fact1 (
        id,
        mod_23,
        mod_31,
        mod_53,
        small_vc,
        padding
)
partition by range (id) (
        partition p_0500000 values less than( 500001),
        partition p_1000000 values less than(1000001),
        partition p_1500000 values less than(1500001),
        partition p_2000000 values less than(2000001)
)
```

```
as
with generator as (
        select  --+ materialize
                rownum         id
        from    all_objects
        where   rownum <= 3000
)
select
        /*+ ordered use_nl(v2) */
        rownum                          id,
        20 * mod(rownum - 1, 23)        mod_23,
        20 * mod(rownum - 1, 31)        mod_31,
        20 * mod(rownum - 1, 53)        mod_53,
        lpad(rownum - 1,20,'0')         small_vc,
        rpad('x',200)                   padding
from
        generator     v1,
        generator     v2
where
        rownum <= 2000000
;
```

The `fact1` table is range-partitioned on an `ID` column and includes three highly repetitive columns. I am going to create three bitmap indexes on this table, one for each of those repetitive columns, and then create matching dimension tables for each of them. Each dimension table will have a **primary key** declared.

To make things more interesting, each dimension table is going to hold 20 times as many distinct values as it needs for this `fact1` table, and there will be an extra column (a *repetitions* column) with a name like `rep_nn` in each dimension table that I will use in an extra predicate when joining the dimensions to the `fact1` table.

Here, for example, is a typical dimension table, and an example of the type of query you might run:

```
create table dim_23
as
select
        rownum - 1           id_23,
        mod(rownum - 1,23)   rep_23,
        lpad(rownum - 1,20)  vc_23,
        rpad('x',2000)       padding_23
from
        all_objects
where
        rownum <= 23 * 20
;

alter table dim_23 add constraint dim_23_pk primary key(id_23);
```

```
--      Collect statistics using dbms_stats here

select
        dim23.vc_23,
        dim31.vc_31,
        dim53.vc_53,
        fact1.small_vc
from
        dim_23,
        dim_31,
        dim_53,
        fact1
where
        fact1.mod_23 = dim_23.id_23
and     dim_23.rep_23 = 10
/*                                */
and     fact1.mod_31 = dim_31.id_31
and     dim_31.rep_31 = 10
/*                                */
and     fact1.mod_53 = dim_53.id_53
and     dim_53.rep_53 = 10
;
```

There are two dramatically different strategies that the optimizer could use for this query. One strategy will try to find an ordering of the four tables that joins each table to the next in the most efficient way, possibly coming up with an execution plan that scans the `fact1` table, and uses nested loop joins or hash joins to each of the dimension tables in turn to eliminate the unwanted data. For example, the optimizer may simply smash its way through all the data doing a three-step hash join as follows:

```
Execution Plan (autotrace 10.1.0.4)
-----------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=10608 Card=423404 Bytes=49538268)
1  0  HASH JOIN (Cost=10608 Card=423404 Bytes=49538268)
2  1    TABLE ACCESS (FULL) OF 'DIM_23' (TABLE) (Cost=25 Card=20 Bytes=560)
3  2    HASH JOIN (Cost=10576 Card=486914 Bytes=43335346)
4  3      TABLE ACCESS (FULL) OF 'DIM_31' (TABLE) (Cost=33 Card=20 Bytes=560)
5  4      HASH JOIN (Cost=10536 Card=754717 Bytes=46037737)
6  5        TABLE ACCESS (FULL) OF 'DIM_53' (TABLE) (Cost=55 Card=20 Bytes=560)
7  6        PARTITION RANGE (ALL) (Cost=10469 Card=2000000 Bytes=66000000)
8  7          TABLE ACCESS (FULL) OF 'FACT1' (TABLE) (Cost=10469 Card=2M Bytes=66M)
```

This execution plan scatters the three-dimension table into memory (probably using an area of memory of at least half the `hash_area_size` for each in-memory hash table if you aren't using the automatic `workarea_size_policy` feature), and then reads each partition from the fact table in turn, probing the three hashed dimension tables before reporting (or discarding) the `fact1` row. Consequently, the cost of the query is close to the cost of performing the full

tablescan across all the partitions. (The cost of scanning and hashing the dimension tables and the cost of the in-memory probes are likely to be negligible by comparison.)

The other strategy is to recognize that there are three bitmap indexes that could allow us to create a very efficient access path to the very small number of rows (just 53 in the example) that we want from the `fact1` table, following which we can join the three dimension tables back very efficiently to pick up any relevant dimension details that we needed.

In effect, the optimizer does the following rewrite on our query, with the extraordinary feature that it uses each dimension table twice:

```
select
        dim23.vc_23,
        dim31.vc_31,
        dim53.vc_53,
        v1.small_vc
from
        dim_23,
        dim_31,
        dim_53,
        (
        select
                mod_23, mod_31, mod_53, small_vc
        from
                fact1
        where
                fact1.mod_23 in (select id_23 from dim_23 where dim_23.rep_23 = 10)
        and     fact1.mod_23 in (select id_31 from dim_31 where dim_31.rep_31 = 10)
        and     fact1.mod_53 in (select id_53 from dim_53 where dim_53.rep_53 = 10)
        )        v1
where
        dim_23.id_23 = v1.mod_23
and     dim_31.id_31 = v1.mod_31
and     dim_53.id_53 = v1.mod_53
;
```

So we should see one part of the execution plan doing a `bitmap and` of the three bitmap indexes on the main fact table, and three subsequent joins (possibly nested loop, merge, or hash joins) to add back the extra columns from the dimension tables. Sure enough, one of the possible plans for this query (when the parameter `star_transformation_enabled` is set to `temp_disable`) is as follows:

```
Execution Plan (autotrace 10.1.0.4)
----------------------------------------------------------
SELECT STATEMENT Optimizer=ALL_ROWS (Cost=256 Card=11)
  HASH JOIN (Cost=256 Card=11)
    HASH JOIN (Cost=230 Card=13)
      HASH JOIN (Cost=174 Card=34)
        TABLE ACCESS (FULL) OF 'DIM_31' (TABLE) (Cost=33 Card=20)
        PARTITION RANGE (ALL) (Cost=139 Card=53)
```

```
+++       TABLE ACCESS (BY LOCAL INDEX ROWID) OF 'FACT1' (TABLE) (Cost=139 Card=53)
            BITMAP CONVERSION (TO ROWIDS)
              BITMAP AND

**              BITMAP MERGE
**                BITMAP KEY ITERATION
**                  BUFFER (SORT)
**                    TABLE ACCESS (FULL) OF 'DIM_53' (TABLE) (Cost=55 Card=20)
**                  BITMAP INDEX (RANGE SCAN) OF 'FACT1_53' (INDEX (BITMAP))

**              BITMAP MERGE
**                BITMAP KEY ITERATION
**                  BUFFER (SORT)
**                    TABLE ACCESS (FULL) OF 'DIM_23' (TABLE) (Cost=25 Card=20)
**                  BITMAP INDEX (RANGE SCAN) OF 'FACT1_23' (INDEX (BITMAP))

**              BITMAP MERGE
**                BITMAP KEY ITERATION
**                  BUFFER (SORT)
**                    TABLE ACCESS (FULL) OF 'DIM_31' (TABLE) (Cost=33 Card=20)
**                  BITMAP INDEX (RANGE SCAN) OF 'FACT1_31' (INDEX (BITMAP))

      TABLE ACCESS (FULL) OF 'DIM_53' (TABLE) (Cost=55 Card=20)
    TABLE ACCESS (FULL) OF 'DIM_23' (TABLE) (Cost=25 Card=20)
```

(Note—to fit the page width, I have eliminated the `bytes=nnnnn` entries from the cost details, leaving only the cost and cardinality.)

THE STAR_TRANSFORMATION AND TEMPORARY TABLES

The parameter `star_transformation_enabled` can take three values: `false` (the default), `true`, and `temp_disable`. When a dimension table exceeds 100 blocks, your session will create an in-memory **global temporary table** to hold the filtered dimension data if you simply enable star transformations by setting the parameter to `true`. This has been known to cause problems in the past, which is why the option to disable temporary tables also exists.

The limit of 100 blocks is controlled by the hidden parameter `_temp_tran_block_threshold`, and the absolute value is independent of the block size of the tablespaces that your dimension tables are in. This is another reason to be cautious about moving objects to tablespaces of different block sizes. Change a dimension table from one block size to another, and you've changed the number of blocks—so the optimizer may change from using a global temporary table to using the base table (or vice versa) for no apparent reason, and the effect may not be beneficial.

I have broken the plan down a little bit to make it easier to see critical operations. Note how the lines marked with the double-asterisk, **, are just three copies of the same structure—the method of deciding how each dimension table identifies sections of the corresponding

bitmap on the fact1 table. We scan each dimension table in turn to pick up the primary key values—the primary key values (when sorted) allow us to walk along the corresponding bitmap index of the fact table, picking up the relevant bitmap sections and merging them into a single bitstream. After we have done this for all three dimensions, we have three bitstreams that we can apply the bitmap and operation to, converting any resulting bits to their corresponding rowid.

The line marked with the +++ is particularly important. In this line, the optimizer tells us how many relevant rows it thinks there are in the fact table: card = 53 (which happens to be correct thanks to the rather contrived way that I constructed the data). Compare this with the cardinality reported from the brutal hash join: card = 423404. The difference is quite significant. In fact, the divergence gets worse, because the final cardinality reported for the complete query is only 11 according to the star transformation plan. The discrepancy arises from two distinct factors.

- First, there seems to be a bug in the code that handles the bitmap and portion of the plan. The reason for the strange names of the dimensions is that I had 23, 31, and 53 distinct values of each column appearing in the fact1 table (plus a lot of other values in the dimension tables to pad them up a bit). The predicates I had on the dimension tables selected 20 rows each, so in theory they should have selected 20 sections from each of the bitmaps on the fact table. However, the optimizer's arithmetic is based on the absolute selectivity of the bitmap indexes, and doesn't allow for the actual number of bitmap sections identified by each of the dimension tables. I had 2,000,000 rows in the fact table, and the selectivities of my three bitmap indexes were 23, 31, and 53, respectively, so the optimizer calculated the cardinality of the bitmap access step as 2,000,000 / (23 * 31 * 53) = 52.925. This happens to be the correct answer in my example, but only because I had rigged my dimension tables so that 19 of the 20 values that I had chosen would not appear in the fact table.

- The second factor that comes into play arises from the fact that star transformations obey normal join arithmetic. After using the bitmap and mechanism to identify the starting set in the fact1 table, the optimizer simply applied the standard join arithmetic to the task of joining back the three dimension tables. And that's highly questionable, of course, because we've already done that arithmetic once to derive the starting number of rows—every row we join back is doing the join to extend the row length, and there is no question of rows being eliminated or multiplied by the second join pass. Star transformations get the wrong cardinality because they apply join selectivities twice!

We don't even have to know how the join arithmetic works to see this second factor in action; we can just use ratios (horror, shock, gasp) to prove the point. In the first execution plan (without the star transformation), I started with 2,000,000 rows, and after three joins the cardinality was down to 423,404—a factor of 4.7236. In the execution plan for the star join, I have an estimated 53 rows as the starting cardinality after the bitmap and before doing the joins back to the dimensions. Divide this by the same 4.7236, and you get 11.2—and the final cardinality after the joins is 11.

If you check the 10053 trace file when you have enabled star transformations, you will find that there are several sections of general plans. The first is the normal (no-transformation) plan, followed by separate plans for each of the dimension tables in turn (each with its own join order[1]), and finally this heading:

```
**************************************
STAR TRANSFORMATION PLANS
**********************
```

After this, you get the first of the join orders (again starting with `join order[1]`) for the join-back. The strange thing about it, though, is that there are no clues in the trace file about the arithmetic that has been used to determine the cardinality of the target data set in the fact table. All you get is a sudden appearance of an assumed cardinality (the 53 rows in my example) with no justification.

Just as a closing thought on star transformations, you might like to wonder why I used such an odd data pattern to demonstrate the effects it can have. First, of course, I wanted to make the divergence in the calculations, and its cause, very obvious. Second, the construction mimics a fairly common design error—many people put lots of little dimensions into a single table with a `type` column.

If you adopt this strategy, then a single column in the fact table corresponds to a subset of the available values in the dimension table, and the peculiar selectivity of the `type` column results in the optimizer exaggerating the difference between the cardinality of the ordinary join and the star transformation join. My predicate `rep_23 = 10` might be your predicate `ref_type = 'COUNTRY CODE'`.

Star Joins

For the sake of completeness, I will mention star joins very briefly. They don't really belong in this chapter, as there is no rewrite or transformation involved in a star join. However, people sometimes confuse star transformations and star joins, so it's worth mentioning them at the same time to eliminate the confusion.

A star join is simply a special case of evaluating a join order that takes advantage of a multi-column index on the fact table to drive a query through a Cartesian join of its dimension tables. There are probably some special circumstances where this could be useful, but I have yet to see a production example where it was appropriate to take advantage of the feature. (Of course, you have to remember that the star join was made available in Oracle 7, before the implementation of the star transformation—so it may simply be an example of outdated technology.)

The script `star_join.sql` in the online code suite has a simple example (with no data) to show this in action. We have a fact table with the following definition:

```
create table fact_tab (
        id1          number        not null,
        id2          number        not null,
        id3          number        not null,
        small_vc     varchar2(10),
        padding      varchar2(100),
        constraint f_pk primary key (id1,id2,id3)
);
```

The three `IDn` columns correspond to the unique ID columns of three separate dimension tables. When we run a query such as the following:

```
select
        /*+ star */
        d1.p1, d2.p2, d3.p3,
        f.small_vc
from
        dim1         d1,
        dim2         d2,
        dim3         d3,
        fact_tab     f
where
        d1.v1 = 'abc'
and     d2.v2 = 'def'
and     d3.v3 = 'ghi'
and     f.id1 = d1.id
and     f.id2 = d2.id
and     f.id3 = d3.id
;
```

we get an execution plan like this one:

```
Execution Plan (autotrace 9.2.0.6)
----------------------------------------------------------
 0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 Card=1 Bytes=127)
 1   0   NESTED LOOPS (Cost=6 Card=1 Bytes=127)

 2   1     MERGE JOIN (CARTESIAN) (Cost=6 Card=1 Bytes=81)
 3   2       MERGE JOIN (CARTESIAN) (Cost=4 Card=1 Bytes=54)
 4   3         TABLE ACCESS (FULL) OF 'DIM1' (Cost=2 Card=1 Bytes=27)
 5   3         BUFFER (SORT) (Cost=2 Card=1 Bytes=27)
 6   5           TABLE ACCESS (FULL) OF 'DIM2' (Cost=2 Card=1 Bytes=27)
 7   2       BUFFER (SORT) (Cost=4 Card=1 Bytes=27)
 8   7         TABLE ACCESS (FULL) OF 'DIM3' (Cost=2 Card=1 Bytes=27)

 9   1     TABLE ACCESS (BY INDEX ROWID) OF 'FACT_TAB'
10   9       INDEX (UNIQUE SCAN) OF 'F_PK' (UNIQUE)
```

As you can see from lines 2 through 8, we have a Cartesian merge join between the three dimension tables, followed (in lines 1, 9, and 10) by a unique index scan of the primary key index to get to the fact table. It is worth pointing out that in this particular example (which actually contained no data), the optimizer found this path without the assistance of the `star` hint.

There was an important lesson in the `10053 trace` files for this test: in the absence of a `star` hint, the 9*i* trace file showed 24 join orders being investigated (4 * 3 * 2 * 1 = 24, so this was the fullest possible list)—and the plan printed previously happened to come from the first join order examined. With the hint in place, the usual `general plans` section of the trace file was missing, and instead we had a section labeled

```
**************************************
STAR PLANS
**********************
```

This was followed by just six plans (3 * 2 * 1) that all had the `fact_tab` in the last position—in other words, the optimizer only tried to find an execution plan that was a proper star join, and didn't consider any others. The `star` hint doesn't help the optimizer to *find* that star join execution plan; it stops the optimizer from wasting time on other options. Hints, generally, are designed to limit the actions of the optimizer, and the `star` hint demonstrates this very clearly.

As a little follow-up, I also created a test case with seven dimension tables. Interestingly, when hinted with the `star` hint, the optimizer examined all 5,040 possible join orders—ignoring the fact that the parameter `_optimizer_max_permutations` was set to 2,000.

In the absence of the `star` hint, 9*i* examined 10 join orders under the heading `general plans`, one join order under the head `star plans` (8*i* did all 5,040 at this point), and four join orders under the heading `additional plans`. It seems the `star` hint allows the optimizer to bypass the normal short-circuit code that makes a trade-off between the extra time required to check more join orders and the risk of missing a better execution plan. Mind you, the optimizer managed to get through all 5,040 join orders in 0.27 seconds (they were all simple join orders with no extra options for indexed access paths, and it was a 2.8 GHz CPU).

The Future

Sometimes, you can see where Oracle is going by looking at hidden parameters and undocumented functions. This section describes a couple of features that you should not use on a production system—but you might like to know about them so that you can understand what's happening when they hit you in a future release.

There could be many ways in SQL to specify a single query, and as time passes the optimizer is enhanced to find new ways to convert safely and efficiently between texts that are structured very differently but produce the same result. Consider the following query (see `intersect_join.sql` in the online code suite):

```
select n2 from t1 where n1 < 3
intersect
select n2 from t2 where n1 < 2
;
```

In principle, provided you deal with the problems of checking equality between `null` columns, you could rewrite this as follows:

```
select  distinct t1.n2
from    t1, t2
where   t1.n1 < 3
and     t2.n1 < 2
and     t2.n2 = t1.n2
;
```

Fortunately, the problem of comparing `nulls` can be solved (at least internally) with the undocumented function `sys_op_map_nonnull()`, which seems to return a value that is deemed to be of the type that matches its input parameter even though it is a value that is not normally considered to be legal for that type (typically a single `0xFF`).

In the preceding example, we need only change the last predicate to `sys_op_map_nonnull(t2.n2) = sys_op_map_nonnull(t1.n2)` to make the second form of the query a valid rewrite of the first query.

Check the execution plan for the first query, and you get

```
Execution Plan (10.1.0.4 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=30 Card=30 Bytes=600)
1  0   INTERSECTION
2  1     SORT (UNIQUE) (Cost=10 Card=45 Bytes=360)
3  2       TABLE ACCESS (BY INDEX ROWID) OF 'T1' (TABLE) (Cost=4 Card=45 Bytes=360)
4  3         INDEX (RANGE SCAN) OF 'T1_I1' (INDEX) (Cost=2 Card=45)
5  1     SORT (UNIQUE) (Cost=20 Card=30 Bytes=240)
6  5       TABLE ACCESS (FULL) OF 'T2' (TABLE) (Cost=14 Card=30 Bytes=240)
```

But set the hidden parameter `_convert_set_to_join` to `true` (this is a 10*g* parameter only), and the plan changes to

```
Execution Plan (10.1.0.4 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=25 Card=33 Bytes=528)
1  0   SORT (UNIQUE) (Cost=25 Card=33 Bytes=528)
2  1     HASH JOIN (Cost=19 Card=33 Bytes=528)
3  2       TABLE ACCESS (FULL) OF 'T2' (TABLE) (Cost=14 Card=30 Bytes=240)
4  2       TABLE ACCESS (BY INDEX ROWID) OF 'T1' (TABLE) (Cost=4 Card=45 Bytes=360)
5  4         INDEX (RANGE SCAN) OF 'T1_I1' (INDEX) (Cost=2 Card=45)
```

The `hash join` execution plan is exactly what you get if you rewrite the query manually from the intersection form to the join form. And if you run a full execution plan report against the plan table, you will find that the `access_predicates` on the hash join is, indeed

```
sys_op_map_nonnull("t2"."n2") = sys_op_map_nonnull("t1"."n2")
```

With this new parameter set, the optimizer can also transform queries using the `minus` set operator into queries using anti-joins. The transformation effectively changes the first query that follows into the second query—and aren't you glad you won't have to do it by hand in the future, as it's too easy to make mistakes with this type of rewrite.

```
select n2 from t1 where n1 < 3
minus
select n2 from t2 where n1 < 2
;

select  distinct n2
from    t1
where   n1 < 3
```

```
and     not exists (
                select  null
                from    t2
                where   n1 < 2
                and     sys_op_map_nonnull(t2.n2) = sys_op_map_nonnull(t1.n2)
        )
;
```

Summary

Before doing any arithmetic, the optimizer may rewrite your query in a form that is easier to model and potentially has more optimization options available. Sometimes it is better to block the rewrite, and there are hints available to allow you to do this. Increasingly, though, the optimizer will run cost calculations for the original query and the restructured query, so in theory such blocking activity should become unnecessary. (I understand that in 10*g* release 2, all transformations are driven by cost—this is generally a good thing, but you may find that some complex queries may now take longer to optimize.)

There are cases where the execution plan cannot adequately describe exactly what the execution engine is going to do. Moreover, the work that has to be done by the run-time engine may vary dramatically because of minor differences in the target data set, so a plan that seems to work well on even a carefully generated test set of data may behave very badly on production data.

The optimizer's ability to transform queries can be seriously affected by the presence (or usually absence) of `not null` constraints. Make sure that if a column is mandatory, it is declared as such to the database.

Similarly, the presence, or absence, of uniqueness constraints can make a difference to the legal options and table ordering when the optimizer is trying to transform subqueries.

In general, if you are going to write SQL with subqueries, you might as well start by writing code that reads like the natural language version of a problem, as this may be the easiest for other people to understand. Generally, the optimizer can transform such queries into more efficient representations. Sometimes, though, you will find that an "obvious" transformation will not take place, and you will have to do a manual transformation of your code. If you do, make sure that it is logically the same as the original.

Star transformations are a popular join strategy for data warehouses. But it is possible that when you enable the feature, some of your execution plans may show dramatic changes in cardinality. This may be due to a collision between your design strategy for dimension tables and the arithmetic the optimizer does for the driving `bitmap and`.

Every time you upgrade, even on a point release, there may be a couple of new transformations that have been enabled. Usually this means that the optimizer can find a better way of operating a query efficiently—occasionally a new transformation may be a bad idea for your specific data set. If the performance of a critical query changes dramatically (for better or worse) on an upgrade, make sure you can check the old execution plan against the new execution plan—there may be a new transformation option that you need to know about.

Test Cases

The files in the download for this chapter are shown in Table 9-3.

Table 9-3. *Chapter 9 Test Cases*

Script	Comment
filter_cost_01.sql	Example of forcing a filter subquery path
push_subq.sql	Example of pushing subqueries (8.1 only)
ord_pred.sql	Demonstration of the ordered_predicates hint
filter_cost_02.sql	Subquery filter going faster with sorted driving table
filter_cost_01a.sql	How bad luck can affect the performance of a filtered subquery
scalar_sub_01.sql	Uses an inline scalar subquery instead of a traditional correlated subquery
scalar_sub_02.sql	Uses a scalar subquery to fake "determinism" in function calls
scalar_sub_03.sql	Uses a scalar subquery to investigate the hash table
with_subq_01.sql	Simple example of scalar subqueries used to generate large data sets
with_subq_02.sql	More complex example of using scalar subqueries to simplify problems.
view_merge_01.sql	Demonstration of changes due to complex_view_merging
push_pred.sql	Demonstration of pushing join predicates
unnest_cost_01.sql	The first filter example, with the SQL written with a *manual unnest*
unnest_cost_02.sql	A filter that looks as if it should take place (in 10*g*) but doesn't
unnest_cost_01a.sql	As unnest_cost_01.sql, with modifications in which averages are required
semi_01.sql	Examples leading up to the semi-join
anti_01.sql	Examples leading up to the anti-join
book_subq.sql	Example of an anomaly where not in does something that not exists cannot
notin.sql	Highlights the problem of nulls and not in subqueries
ordered.sql	Example of how the ordered hint can cause problems on upgrade
star_trans.sql	Example of a star transformation
star_join.sql	Simple demonstration of a star join (not star transformation)
intersect_join.sql	Demonstration of how 10*g* can convert a set operation to a join (or anti-join)
setenv.sql	Sets a standardized environment for SQL*Plus

CHAPTER 10

Join Cardinality

What is the maximum number of tables that Oracle can join at once? You may be surprised to learn that the answer is two. It doesn't matter how many tables you have in your query, Oracle will only work on two objects at a time in a join. In fact, you could even argue that the optimizer doesn't have a long-term strategy for joins, it simply takes what it's got at any one point and joins on the next available table to see what happens.

Of course, this description is a little fanciful—but it's not far from the truth. To perform a five-table join, the optimizer picks a starting table and joins on one table; it takes the intermediate result and joins on one more table; it takes the intermediate result ... and so on, until all five tables have been used; so it is perfectly true that Oracle can only join two tables.

But there is a strategy—which is to make a sensible first guess at a good join order, and then work methodically through a series of permutations of that join order in a way that tries to minimize the risk of wasting time on really bad join orders while maximizing the chance of finding a better join order.

There are three things to consider with joins: how the optimizer calculates the join cardinality, how the optimizer calculates the join cost, and how the execution of each type of join is actually done.

In this chapter, we will focus only on the calculation of cardinality.

Basic Join Cardinality

Oracle Corp. published a pair of formulae for join cardinality in MetaLink note 68992.1 dated March 1999 and last updated (as of time of writing) in April 2004. The latest incarnation of the formula for the selectivity of a join is as follows:

```
Sel = 1 / max[(NDV(t1.c1), NDV(t2.c2)] *
        ((card t1 - #  t1.c1 NULLS) / card t1) *
        ((card t2 - #  t2.c2 NULLS) / card t2))
```

with the join cardinality then given as

```
Card (Pj) = card(t1) * card(t1) * sel(Pj)
```

These formulae aren't completely consistent in style and need a little translation—in particular, the expression `card t1` that appears in the selectivity formula does not mean the same thing as the `card(t1)` that appears in the cardinality formula. To clarify the meaning, I have rewritten the formulae as follows:

```
Join Selectivity =
        ((num_rows(t1) - num_nulls(t1.c1)) / num_rows(t1)) *
        ((num_rows(t2) - num_nulls(t2.c2)) / num_rows(t2)) /
        greater(num_distinct(t1.c1), num_distinct(t2.c2))

Join Cardinality =
        Join Selectivity *
        filtered cardinality(t1) * filtered cardinality(t2)
```

To translate: assume we are joining tables `t1` and `t2` on columns `c1` and `c2`, respectively. We check `user_tables` for the `num_rows` in each of the two tables and check view `user_tab_col_statistics` for the `num_nulls` and `num_distinct` for the two columns, and slot these values into the formula for join selectivity. (Later you will see that there is a variation of the formula that uses the `density` rather than the `num_distinct`.)

The join cardinality, however, includes the *filtered cardinality* of each table. Remember that a general SQL statement may have some predicates other than the join predicates. These extra *filter predicates* are the ones that should be applied to the tables first to derive the filtered cardinality—and it is possible that some of the join predicates may also function as filter predicates.

The easiest way to explain the formula is through an example. The following is an extract from the script `join_card_01.sql` in the online code suite. As usual, my demonstration environment starts with an 8KB block size and 1MB extents, locally managed tablespaces, manual segment space management, and system statistics (CPU costing) disabled.

```
create table t1 as
select
        trunc(dbms_random.value(0, 25 ))        filter,
        trunc(dbms_random.value(0, 30 ))        join1,
        lpad(rownum,10)                         v1,
        rpad('x',100)                           padding
from
        all_objects
where   rownum <= 10000
;

create table t2 as
select
        trunc(dbms_random.value(0, 50 ))        filter,
        trunc(dbms_random.value(0, 40 ))        join1,
        lpad(rownum,10)                         v1,
        rpad('x',100)                           padding
from
        all_objects
where   rownum <= 10000
;

--        Collect statistics using dbms_stats here
```

```
select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t1.filter = 1
and     t2.join1 = t1.join1
and     t2.filter = 1
;
```

In all the test cases in this chapter, I have used the `dbms_random.value()` procedure to generate randomized but predictable data, using the `(low, high)` parameters and truncating the result to control the number of distinct values that will appear in the various filter and join columns. In this example, we have the following:

`t1.filter` 25 values

`t2.filter` 50 values

`t1.join1` 30 values

`t2.join1` 40 values.

The filter columns are the easy step—given that both tables hold 10,000 rows, the filtered cardinality of `t1` will be 400 (10,000 rows divided by 25 distinct values), and the filtered cardinality of `t2` will be 200 (10,000 rows divided by 50 distinct values).

Since there are no null values in either table, the formulae for join cardinality give us the following:

```
Join Selectivity =
        (10,000 - 0) / 10,000) *
        (10,000 - 0) / 10,000) /
        greater(30, 40) =
        1/40

Join Cardinality = 1/40 * (400 * 200) = 2000
```

Sure enough, when we run the query through `autotrace`, we see the following plan:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=57 Card=2000 Bytes=68000)
   1    0   HASH JOIN (Cost=57 Card=2000 Bytes=68000)
   2    1     TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=200 Bytes=3400)
   3    2     TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=400 Bytes=6800)
```

This shows the filtered cardinality of table `t1` as 400, the filtered cardinality of `t2` as 200, and the cardinality of the join as 2,000, as we predicted. The formulae seem to work.

We can complicate the test (see `join_card_02.sql` in the online code suite) by making every 20th row of `t1` hold a null value for its join column, and every 30th row of `t2` hold a null for its join column:

```
update t1 set join1 = null
where mod(to_number(v1),20) = 0;

--        500 rows updated

update t2 set join1 = null
where mod(to_number(v1),30) = 0;

--        333 rows updated
```

With this change to the data (and after collecting statistics), we just have one more detail to include in the join selectivity:

```
Join Selectivity =
        (10,000 - 500) / 10,000) *
        (10,000 - 333) / 10,000) /
        greater(30, 40) =
        0.022959125
Join Cardinality = 400 * 200 * 0.022959125 = 1,836.73
```

And again, we see that the cardinality predicted by the formula matches the cardinality of 1,837 reported by `autotrace`.

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=57 Card=1837 Bytes=62458)
   1    0   HASH JOIN (Cost=57 Card=1837 Bytes=62458)
   2    1     TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=200 Bytes=3400)
   3    1     TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=400 Bytes=6800)
```

As a final complication, we can introduce some null values to the filter columns (see `join_card_03.sql` in the online code suite) as well as the nulls we put into the join columns. The following code introduces 200 nulls to the filter column in table `t1`, and 100 nulls to table `t2`:

```
update t1 set filter = null
where mod(to_number(v1),50) = 0;

--  200 rows updated

update t2 set filter = null
where mod(to_number(v1),100) = 0;

--  100 rows updated
```

We already have a join selectivity of 0.22959125 from our previous example, so all we have to do is work out the filtered cardinality of each of the tables when the columns with the filter predicates have some nulls. But I showed you how to do that in Chapter 3. The formula is

```
Adjusted (computed) Cardinality = Base Selectivity * (num_rows - num_nulls)
```

- For table t1 we have 1/25 * (10,000 – 200) = 392.
- For table t2 we have 1/50 * (10,000 – 100) = 198.

Putting these values into the formula for join cardinality, we get

```
Join Cardinality = 392 * 198 * 0.022959125 = 1,781.995
```

Compare this with the autotrace output, noting also the cardinality for each of the separate tablescans:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=57 Card=1782 Bytes=60588)
   1    0   HASH JOIN (Cost=57 Card=1782 Bytes=60588)
   2    1     TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=198 Bytes=3366)
   3    1     TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=392 Bytes=6664)
```

So the formulae seem to work—at least, for very simple cases. (Strangely, the 8*i* calculations go slightly wrong, with a cardinality of 199 rather than 198 on table t2, leading to a join cardinality of 1791. This is probably just an example of simple computational error—dividing by 5 is a common cause of errors in the Nth decimal place.)

Unfortunately there are problems that still need addressing—lots of them. Let's try to find a few questions about the limitations of the formulae.

- What are you supposed to do if you have two or more join columns?
- What do you do about a join condition that includes a range scan?
- How do you join a third table?
- Why doesn't the formula seem to allow for cases where the ranges of the join columns only have a partial overlap?
- Do histograms have any effect?

With a little patience and experimentation, we can find the answers to some of these questions. As usual, it turns out that the code contains many rational strategies, some decisions that don't seem entirely reasonable, some special cases, and some things that look like bugs.

Biased Joins

Let's finish this section with just one very simple case where the optimizer "breaks" the rules—or rather, uses a rule we didn't know about. Let's go all the way back to the simplicity of our first example (no nulls, simple equality), but produce a test that applies a filter to just one of the two tables in the join (see script join_card_01a.sql in the online code suite):

```
create table t1 as
select
        trunc(dbms_random.value(0, 100 ))      filter,
        trunc(dbms_random.value(0,  30 ))      join1,
        lpad(rownum,10)                        v1,
        rpad('x',100)                          padding
from
        all_objects
where   rownum <= 1000
;

create table t2 as
select
        trunc(dbms_random.value(0, 100 ))      filter,
        trunc(dbms_random.value(0,  40 ))      join1,
        lpad(rownum,10)                        v1,
        rpad('x',100)                          padding
from
        all_objects
where   rownum <= 1000
;

--        Collect statistics using dbms_stats here

select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t2.join1 = t1.join1
-- and     t1.filter = 1
and     t2.filter = 1
;
```

In this test, I have 10,000 rows in each table, and a filter column with 100 distinct values. The query shown filters only on table `t2` and has the filter on table `t1` commented out, but the complete script for the test case has a second query that filters only `t1`, commenting out the filter on `t2`. There are 30 distinct values in the join column for `t1` and 40 for `t2`.

If you check the selectivity formula, it shouldn't really matter which table has the filter condition on it; the selectivity is only looking at the `num_distinct` on the join columns:

```
Join Selectivity =
        ((num_rows(t1) - num_nulls(t1.c1)) / num_rows(t1)) *
        ((num_rows(t2) - num_nulls(t2.c2)) / num_rows(t2)) /
        greater(num_distinct(t1.c1), num_distinct(t2.c2))
```

in this case:

```
Join Selectivity =
        (1000 / 1000) * (1000 / 1000) / greater(30, 40) =
        1/40 =
        0.025
```

Since the filter condition is the same (one value in 100) on each of the tables, the join cardinality is going to come out the same, whichever table has the filter applied:

```
Join Cardinality =
        Join Selectivity *
        filtered cardinality(t1) * filtered cardinality(t2) =
        0.025 * 10 * 1000 =
        250
```

So here are the two execution plans:

```
Execution Plan (9.2.0.6 Filter on just T1)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=11 Card=250 Bytes=7750)
   1    0   HASH JOIN (Cost=11 Card=250 Bytes=7750)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=5 Card=10 Bytes=170)
   3    1     TABLE ACCESS (FULL) OF 'T2' (Cost=5 Card=1000 Bytes=14000)

Execution Plan (9.2.0.6 Filter on just T2)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=11 Card=333 Bytes=10323)
   1    0   HASH JOIN (Cost=11 Card=333 Bytes=10323)
   2    1     TABLE ACCESS (FULL) OF 'T2' (Cost=5 Card=10 Bytes=170)
   3    1     TABLE ACCESS (FULL) OF 'T1' (Cost=5 Card=1000 Bytes=14000)
```

Change the filter table, and you change the cardinality of the result. Filtering on table `t1`, we get a cardinality of 250, as we inferred from the formula. Filter on table `t2`, and the cardinality changes to 333. Can you guess where that came from? Is it a coincidence that 1,000 / 30 = 333?

One simple variation on the join cost algorithm appears when there is a filter predicate at just one end of the join—we may use the `num_distinct` values from the table at the other end of the join, not the greater of the two `num_distinct`. (And I suspect that this statement is probably just a very special case of a more general rule about how filter predicates may affect the way that the optimizer decides which table to use as the source of `num_distinct`.)

When we have a filter only on `t2`, we use the `num_distinct` of 30 from table `t1`, rather than the larger `num_distinct` of 40 required by the standard formula.

Join Cardinality for Real SQL

Unless your system has been cursed with single-column synthetic keys all over the place, you will probably write some SQL that involves two, or more, columns being used in the join condition between two tables. So how do you extend the basic formula to deal with multicolumn joins? The strategy is exactly the same one you would use for multiple predicates on a single table—just apply the formula once for each predicate in turn and multiply up to derive the final join selectivity. (And then wait for the trap that 10*g* introduces with its *sanity checks*.)

Consider the following extract (see `join_card_04.sql` in the online code suite):

```
create table t1 as
select
        trunc(dbms_random.value(0, 30 ))        join1,
        trunc(dbms_random.value(0, 50 ))        join2,
        lpad(rownum,10)                         v1,
        rpad('x',100)                           padding
from
        all_objects
where   rownum <= 10000
;

create table t2 as
select
        trunc(dbms_random.value(0, 40 ))        join1,
        trunc(dbms_random.value(0, 40 ))        join2,
        lpad(rownum,10)                         v1,
        rpad('x',100)                           padding
from
        all_objects
where   rownum <= 10000
;

--        Collect statistics using dbms_stats here

select  t1.v1, t2.v1
from
        t1,
        t2
where
        t2.join1 = t1.join1
and     t2.join2 = t1.join2
;
```

When we run this through `autotrace` (and 9*i* and 8*i* should produce the same cardinality, though the costs may differ), we get the following:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=60 Card=50000 Bytes=1700000)
   1    0   HASH JOIN (Cost=60 Card=50000 Bytes=1700000)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=10000 Bytes=170000)
   3    1     TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=10000 Bytes=170000)
```

The cardinalities on `t1` and `t2` are clearly correct; we haven't filtered out any rows on the tablescans. Consequently, our join cardinality of 50,000 must come from 100,000,000 (i.e., 10,000 * 10,000) times a join selectivity of 1/2,000—but where in our data definitions can we find that "magic" 2,000?

If we think about applying the selectivity formula twice, we'll soon spot it. The critical component in our case is the bit that goes

```
greater(num_distinct(t1.c1), num_distinct(t2.c2))
```

We have two join components, `t1.join1 = t2.join1` and `t1.join2 = t2.join2`. In Chapter 3 you learned that you combine selectivities on single tables by multiplying them together (at least, that's what you do when the individual predicates have an `and` between them). You do exactly the same with the join selectivities:

```
Join Selectivity =
        {join1 bit} *
        {join2 bit} =

        (10,000 - 0) / 10,000) *
        (10,000 - 0) / 10,000) /
        greater(30, 40)          *         -- uses the t2 selectivity (1/40)

        (10,000 - 0) / 10,000) *
        (10,000 - 0) / 10,000) /
        greater(50, 40)          =         -- uses the t1 selectivity (1/50)

        1/40 * 1/50 =
        1/2000                (as required)
```

So all we've done is examine each part of the join in turn, and use the most selective value in each case—even if it means the selectivity comes from a different table at each step of the calculation.

■ **Note** You may have come across an old bug where the optimizer would fail to notice repeated predicates (for example, add an extra `t2.join2 = t1.join2` to the preceding query, and the cardinality would come out at 1,000 instead of 50,000 because the optimizer had done the arithmetic for the `join2` columns twice). Various aspects of this problem have been dealt with in 9*i*.

Then, just as you think you're getting the hang of how things work, you run the test on 10*g* and the `autotrace` output comes up with a different answer:

```
Execution Plan (10.1.0.4 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=60 Card=62500 Bytes=2125000)
   1    0   HASH JOIN (Cost=60 Card=62500 Bytes=2125000)
   2    1     TABLE ACCESS (FULL) OF 'T1' (TABLE) (Cost=28 Card=10000 Bytes=170000)
   3    1     TABLE ACCESS (FULL) OF 'T2' (TABLE) (Cost=28 Card=10000 Bytes=170000)
```

How did the computed cardinality of 50,000 in the 9*i* plan change into the 62,500 in the preceding execution plan? If you're really good at mental arithmetic, you may be able to guess

the answer really quickly—that's one reason why it's good to make sure you construct experiments where nice, easy numbers are likely to drop out. When in difficulty, I often try working backwards—in this case, from the join cardinality to the join selectivity:

```
62,500 = 100,000,000 * join selectivity.
Join selectivity = 62,500 / 100,000,000 = 1/1,600
```

Can we find 1,600 anywhere in our test case? Yes we can: 1/1,600 = 1/40 * 1/40.

The optimizer has used two selectivities from the same side of the join instead of one from each side. In fact, we can see this confirmed in the `10053 trace` file. There are two indicators.

Following the `single table access path` section, we see

```
Table:  T1  Multi-column join key card: 1500.000000
Table:  T2  Multi-column join key card: 1600.000000
```

In fact, if you also have multicolumn indexes that might be relevant, this part of the trace file also reports the number of distinct keys in these indexes with the description `Concatenated index card`.

At the tail end of the `NL Join` section (where the join cardinality is worked out), we then see the following:

```
Using multi-column join key sanity check for table T2
Revised join selectivity: 6.2500e-004 = 5.0000e-004 * (1/1600) * (1/5.0000e-004)
Join Card:  62500.00 = outer (10000.00) * inner (10000.00) * sel (6.2500e-004)
```

Again, with suitable multicolumn indexes in place, the first of these lines might refer to the `concatenated index cardinality` for table `t2`. It is an interesting thought that you may choose to add a column to an index in order to make some queries index-only. A side effect of this is that this 10*g* sanity check will then cease to be used for any joins that might involve just the columns from the original index definition.

As far as I can tell, this substitution always takes place (at least on nice, easy two-table joins). The optimizer works out (perhaps for historical reasons only) the *worst case* selectivity by taking the greater of the two `num_distinct` values for each pair of joined columns in turn. Then it works the equivalent figure for each table in turn by looking only at the values for `num_distinct` on that table, and actually uses whichever table-based figure gives the smaller value for selectivity. (In this case, 1/1,600 is smaller than 1/1,500.) The parameter `_optimizer_join_sel_sanity_check` is probably the one that controls this behavior.

As a final wrinkle, if the `multi-column join key cardinality` for an individual table goes below 1/`num_rows` for that table, then the optimizer appears to substitute 1/`num_rows` instead of the calculated value.

Extensions and Anomalies

Given our understanding of the basic calculation for well-behaved queries and nice data, we can now move on to some of the many variations on a theme that can appear.

Joins by Range

One special case that we haven't examined yet is what the optimizer does about joins that are range-based joins. The answer is simple: the optimizer uses a predetermined fixed value as the selectivity of this join predicate—just as it did with single tables and bind variables.

For example, rerun script `join_card_01.sql`, but change the single join predicate from

```
t2.join1 = t1.join1
```

to

```
t2.join1 > t1.join1
```

The `autotrace` output shows that the cardinality has changed from 2,000 to 4,000. The selectivity of 1/40 (from the `t2.join1` end of the join) has been replaced by a flat 5% (1/20). You might note, in passing, that the join mechanism has switched from a hash join to a merge join. Hash joins can work only for equalities; they cannot be used for range-based joins.

```
Execution Plan (autotrace 9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=68 Card=4000 Bytes=136000)
   1    0   MERGE JOIN (Cost=68 Card=4000 Bytes=136000)
   2    1     SORT (JOIN) (Cost=34 Card=200 Bytes=3400)
   3    2       TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=200 Bytes=3400)
   4    1     SORT (JOIN) (Cost=35 Card=400 Bytes=6800)
   5    4       TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=400 Bytes=6800)
```

Now change the predicate to

```
t2.join1 between t1.join1 - 1 and t1.join1 + 1
```

and the join cardinality changes to 200. Remember that the optimizer treats `between :bind1 and :bind2` as a pair of independent predicates and multiplies 1/20 by 1/20 to get a selectivity of 1/400 (0.025%). It has done exactly the same here.

The selectivity of the join has changed from the original 1/40 to 1/400, so the cardinality has dropped by the same factor of 10 from 2,000 to 200.

```
Execution Plan (autotrace 9.2.0.6)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=68 Card=200 Bytes=6800)
   1    0   MERGE JOIN (Cost=68 Card=200 Bytes=6800)
   2    1     SORT (JOIN) (Cost=34 Card=200 Bytes=3400)
   3    2       TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=200 Bytes=3400)
   4    1     FILTER
   5    4       SORT (JOIN)
   6    5         TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=400 Bytes=6800)
```

Of course, you can't help noticing that the original join clause `t2.join1 = t1.join1` with a calculated cardinality of 2,000 is going to identify a proper subset of our new range-based join `t2.join1 between t1.join1 - 1 and t1.join1 + 1`, which has been given a calculated cardinality of 200. Clearly there is a problem of consistency that needs to be addressed in the strategy for calculating range-based joins.

I shall explain the significance of the `filter` operation on line 4 in Chapter 13. You might notice, though, a little reporting error that has made the `sort(join)` line below the `filter` lose its cost figure—which should 34 (as in line 2)—even though the `merge join` on line 1 does know about it.

Not Equal

The next special case is inequalities: `t1.join1 != t2.join1`. The same rule that we learned for single-table selectivity applies here—the selectivity (`not(t1.join1 = t2.join1)`) is simply 1 – selectivity (`t1.join1 = t2.join1`). For example, if we have the following query (based on `join_card_04.sql` again):

```
select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t2.join1 != t1.join1          -- (30 / 40 values for num_distinct)
and     t2.join2 != t1.join2          -- (50 / 40 values for num_distinct)
;
```

where we used 1/40 * 1/50 in our calculations when both predicates were equalities, we would now use 39/40 * 49/50 because we now have two inequalities.

There are some quirky side effects to this—bugs, even. Consider a query with a **disjunct** (**OR**) between two predicates:

```
select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t2.join1 = t1.join1          -- (30 / 40 values for num_distinct)
or      t2.join2 = t1.join2          -- (50 / 40 values for num_distinct)
;
```

All three major versions of Oracle report the cardinality of this query as 2,125,000 (see `join_card_05.sql` in the online code suite), but the only version where `autotrace` gives you a sensible clue about what is going on is 10*g*, where the `autotrace` output looks like this:

```
Execution Plan (10.1.0.4 autotrace)
----------------------------------------------------------
 0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=120 Card=2125000 Bytes=72250000)
 1    0   CONCATENATION
 2    1     HASH JOIN (Cost=60 Card=2000000 Bytes=68000000)
 3    2       TABLE ACCESS (FULL) OF 'T1' (TABLE) (Cost=28 Card=10000 Bytes=170000)
 4    2       TABLE ACCESS (FULL) OF 'T2' (TABLE) (Cost=28 Card=10000 Bytes=170000)
 5    1     HASH JOIN (Cost=60 Card=125000 Bytes=4250000)
```

```
6    5       TABLE ACCESS (FULL) OF 'T1' (TABLE) (Cost=28 Card=10000 Bytes=170000)
7    5       TABLE ACCESS (FULL) OF 'T2' (TABLE) (Cost=28 Card=10000 Bytes=170000)
```

Notice how line 2 shows a cardinality of 2,000,000. In 8*i* and 9*i*, this line reports a cardinality of 125,000, which means they should show a total cardinality of 250,000 in line 0—but both versions then show a total cardinality of 2,125,000.

The problem with the final cardinality is that it is clearly wrong. In fact, if you examine the `10053 trace`, you find that at its very first step the optimizer got the cardinality right (at 4,450,000) and then went through a long-winded process to get to the wrong answer. (If you put the `/*+ no_expand */` hint into this query, Oracle will do a single join, and the execution plan will show the correct cardinality—but the join will be a horrendous nested loop operation.)

USING HINTS

There are cases where you may want to use hints to force the optimizer into using a particular mechanism for a join so that it calculates the correct cardinality. There are fairly frequent reports on the various Oracle forums of examples of SQL where a feature (or enhancement) needs to be disabled by a hint because the feature is not appropriate for every case.

The 2,000,000 that appears in the first hash join probably comes from the predicate `t2.join2 = t1.join2`, using the selectivity of 1/50. I suspect that the 125,000 in the second hash join is generated by considering a selectivity of 1/40 on the join predicate `t2.join1 = t1.join1`, but then introducing an inappropriate, and incorrectly applied, factor of 5% (the *bind selectivity factor*) to eliminate rows that have previously been identified by the `join2` predicate.

There are two ways you could find out the right answer for the cardinality; one of them is simply to rewrite the query in an equivalent form, show here with its execution plan:

```
select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t2.join2 = t1.join2
union all
select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t2.join1 = t1.join1
and     t2.join2 != t1.join2
;
```

```
Execution Plan (10.1.0.4 autotrace)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=120 Card=4450000 Bytes=139300000)
1    0   UNION-ALL
2    1     HASH JOIN (Cost=60 Card=2000000 Bytes=56000000)
3    2       TABLE ACCESS (FULL) OF 'T1' (TABLE) (Cost=28 Card=10000 Bytes=140000)
4    2       TABLE ACCESS (FULL) OF 'T2' (TABLE) (Cost=28 Card=10000 Bytes=140000)
5    1     HASH JOIN (Cost=60 Card=2450000 Bytes=83300000)
6    5       TABLE ACCESS (FULL) OF 'T1' (TABLE) (Cost=28 Card=10000 Bytes=170000)
7    5       TABLE ACCESS (FULL) OF 'T2' (TABLE) (Cost=28 Card=10000 Bytes=170000)
```

As you can see, when written as an explicit `union all`, rather than allowing the optimizer to generate its implicit concatenation, you get a very different cardinality on the second half of the query.

The alternative way to decide that the cardinality of 2,125,000 is incorrect and that 4,450,000 is right is to fall back on the formula for combining predicates that we first saw in Chapter 3:

```
The selectivity of (predicate1 OR predicate2) =
    selectivity of (predicate1) +
    selectivity of (predicate2) -
    selectivity of (predicate1 AND predicate2)  -- or you count the overlap twice
```

From the test cases we've used so far (mainly `join_card_01.sql` and `join_card_04.sql`), we already know the individual selectivities:

```
t1.join1 = t2.join1                                1/40
t1.join2 = t2.join2                                1/50
t1.join1 = t2.join1 and t1.join2 = t2.join2        1/2000
```

Hence the selectivity of the `t1.join1 = t2.join1 or t1.join2 = t2.join2` should be as follows:

```
1 / 40 + 1 / 50 - 1 / 2000 =  89 / 2000 = 0.0445
```

Apply that to the 100,000,000 rows that is the unfiltered Cartesian join of the two tables, and the join cardinality will be 4,450,000—as required.

Clearly, there are cases where the code paths used to work out join selectivities and join cardinalities are not self-consistent. You may need to take some action to work around the problems this can cause.

Overlaps

There is a problem built into the implementation. So far we've been using well-behaved data, and I'm about to change that in one important respect. The nice feature of our data is that the number of distinct values in the join columns has matched up quite well, and the range (low/high values) of the joins columns has also matched up quite nicely. In other words, our two tables are *supposed* to join properly.

Let's see what happens to the join cardinality when the overlap is less convenient. The example (see `join_card_06.sql` in the online code suite) creates two tables of 10,000 rows each, and then executes a `select` statement with a single, simple, predicate:

```
t1.join1 = t2.join1
```

The `join1` columns both have 100 distinct values, and at the start of the test their values range from 0 to 99. But we run the test time and time again, rebuilding table `t1` each time and changing the low/high values for `t1.join1` by a different offset from the `t2.join1` value each time so that we can report tests where `t1.join1` ranging from –100 to –1, or from 100 to 199.

Table 10-1 shows the effects on the cardinality from running a few tests on 9*i* or 10*g*. Remember that column `t2.join1` is always built with values from 0 to 99.

Table 10-1. *Join Cardinality Problems*

t1.join1 Low Value	t1.join1 High Value	Computed Cardinality	Actual Rows
–100	–1	1	0
–50	49	1,000,000	486,318
–25	74	1,000,000	737,270
0	99	1,000,000	999,920
25	124	1,000,000	758,631
50	149	1,000,000	513,404
75	174	1,000,000	262,170
99	198	1,000,000	10,560
100	199	1	0

Spot the issue—until the point where the tables *fail to overlap* at all, the computed cardinality does not change from assuming that the overlap is 100%. (In fact, 8*i* is even worse—it doesn't even notice that the overlap has disappeared completely and keeps reporting a cardinality of 1,000,000.) If you want a graphic image of what is happening, Figure 10-1 represents the 0, 50, and 100 state of the two tables—visually it is *obvious* that the join is likely to be less successful as the two tables slide past each other, but the arithmetic simply doesn't change.

If you are joining two tables with the expectation of eliminating rows because rows in one table are not supposed to exist in the other table, then the optimizer may well produce an inappropriate execution plan because its rules of calculation do not match your knowledge of the data.

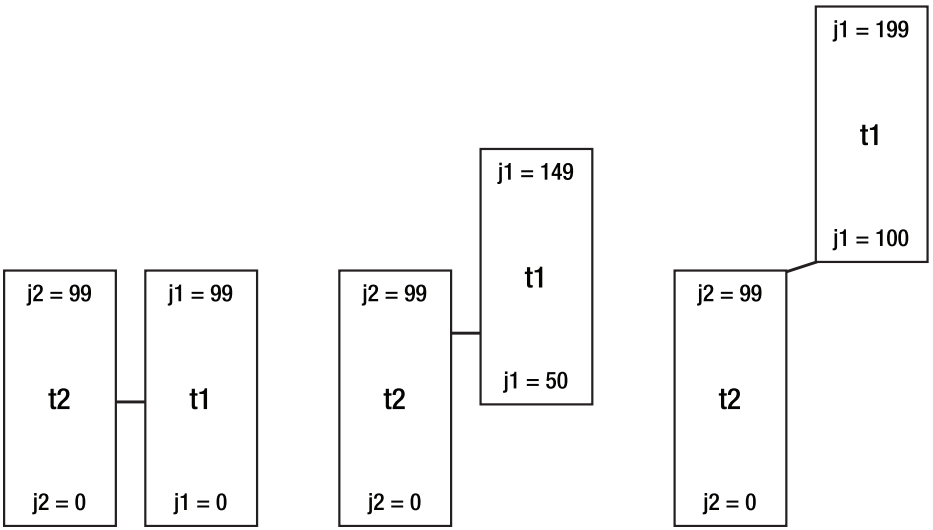

Figure 10-1. *Join cardinality—anomaly*

Histograms

I haven't been entirely truthful about overlaps, because sometimes the optimizer will use alternative arithmetic to calculate the join selectivity, and the alternative seems to recognize the effect of the overlapping ranges. Consider the results shown in Table 10-2, again from 9*i*/10*g*, based on exactly the same data as we used in the previous section, but with a carefully selected histogram built on the two columns at either end of the join.

Table 10-2. *Histograms Helping Join Cardinality*

t1.join1 Low Value	t1.join1 High Value	Computed Cardinality	Actual Rows
–100	–1	1,000,000	0
–50	49	1,000,000	486,318
–25	74	1,000,000	737,270
–22	79	622,025	767,437
0	99	1,007,560	999,920
25	124	767,036	758,631
50	149	538,364	513,404
75	174	286,825	262,170
99	198	1,000,000 (10,615)	10,560
100	199	1,000,000	0

With the histograms in place, I achieved reasonable, though not fantastic, accuracy over a wide range of overlaps. With the histograms I chose initially, the computed cardinality went back to 1,000,000 when the low value on `t1` went below –22 or above 77.

You will notice that in the 99 row, I have shown a join cardinality of 10,615 in parentheses. When I changed my histogram definitions, the optimizer got accurate figures, as `t1.join1` varied all the way from –99 to +99, before flipping back to a computed cardinality of 1,000,000 when `t1.join1` hit +/–100. Histograms look helpful—although they do introduce a new problem when the data sets don't overlap at all.

So what was special about my final choice of histograms, and why did one set of histogram definitions give better results than another? After all, given the way I have been generating my data using the `dbms_random.value()` procedure, there aren't likely to be any extreme values.

The answer is that one set of histograms was built with 85 buckets, the other was built with a nominal 254.

CHOOSING THE RIGHT NUMBER OF BUCKETS

Recall that 254 buckets is (currently) the maximum number of buckets you can request for a histogram. The histogram will be a **frequency histogram** when the number of buckets exceeds the number of different values.

If a histogram on a column is going to be beneficial, you may as well ask for the maximum bucket count as a first guess at the best number of buckets—the marginal cost of the extra buckets (the default is 75) is not significant for the potential benefit you could gain from the extra precision.

Since I had only 100 different values in the columns, the histograms with 254 buckets actually collapsed to become high-precision frequency histograms with 100 endpoints and an exact picture of the current data content—so the optimizer was able to make very good use of them, even in the join.

The histograms with 85 buckets were the more common height balanced histograms, and the critical feature was that one of them (in fact the one at the `t2` end of the join) did show a *popular value*. It seems that if

- You are running 9*i* or 10*g*,
- And you have histograms at both ends of an equality join,
- And at least one histogram either is a frequency histogram or shows a popular value,

then the optimizer has some method (somehow comparing the histogram data to estimate the number of rows and distinct values in each table in the overlap) that makes it possible to allow for joins where there is only a partial overlap in the ranges of values in the columns being joined. Don't take this as a directive to build histograms all over the place, though. It is useful to know, however, so that you can test the effects, when you find those few critical pieces of SQL where you can see a join cardinality going badly wrong.

FREQUENCY HISTOGRAMS AND DBMS_STATS

There is a problem getting a frequency histogram out of the `dbms_stats` package. I have examples of data sets with only 100 distinct values, but the SQL used by Oracle in 9*i* and 10*g* (until 10.2) to generate the histogram failed to build a frequency histogram until I requested 134 buckets in the baseline test.

There are a couple of side issues to consider.

First, 8*i* also takes advantage of histograms in this situation, but the calculations used must be different because the results are much less accurate unless the histogram is the perfect frequency distribution histogram. Table 10-3, for example, shows the results you get from 8*i* when using the same 85-bucket histograms as we did previously for 9*i* and 10*g*.

Table 10-3. *Histograms in 8*i *Give Different Join Arithmetic*

t1.join1 Low Value	T1.join1 High Value	Computed Cardinality	Actual Rows
–100	–1	1,000,000	0
–50	49	147,201	486,318
–25	74	350,499	737,270
–10	89	497618	895,925
0	99	616,242	999,920
25	124	447,791	758,631
50	149	328,109	513,404
75	174	167,392	262,170
99	198	41,034	10,560
100	199	1,000,000	0

The other issue is the problem of finding a histogram definition that just happens to work if you can't get a frequency distribution histogram. Basically, it is a question of choosing the right number of buckets to make a popular value visible if there is such a value. The optimizer seems to check for popular values by comparing the number of buckets defined against the number of endpoints stored. If you check view `user_tab_histograms` for a specific column, then the maximum value for column `endpoint_number` (assuming you haven't switched from a height balanced histogram to a frequency distribution histogram) will be one less than the number of rows stored if there are no popular values.

To show how awkward it can be to find the *right* number of buckets for a histogram, look at Table 10-4, which was generated from the data used for the tests in `join_card_06.sql`. It lists a range of bucket counts and shows whether or not each bucket count managed to identify a popular value—remember, all I am doing in this test is changing the number of buckets in the

histogram, I am not changing the data itself. The last column in the table shows the effect that the choice of bucket count had on our test query when the overlap between the two tables was fixed at 50%. The results are actually taken from the 8*i* tests, but similar (slightly less dramatic) results also came out of the 9*i* and 10*g* tests.

Table 10-4. *Picking the Right Bucket Count Is a Matter of Luck*

Bucket Count	Found Popular	Computed Cardinality at 50% Overlap
76	Yes	367,218
77	No	1,000,000
78	Yes	348,960
79	Yes	355,752
80	No	1,000,000
81	Yes	346,040
82	No	1,000,000
83	Yes	336,838

The problem with depending on histograms to get correct join cardinalities is that you dare not regenerate the histograms in case the data has just changed enough to hide all popular values for the number of columns you have selected. On the other hand, you dare not leave the histograms unchanged if the data keeps changing in a way that moves the high (or low) values, as the size of the overlap is the thing that you are trying to capture with the histogram.

■ **Note** It is an interesting point that the optimizer can use the `density` from `user_tab_columns` instead of 1/`num_distinct` when applying filter predicates but doesn't seem to do so when applying join predicates (at least not directly; it is possible that the join selectivity is derived from the `density` in some way that I haven't yet worked out).

It's a difficult problem to address—essentially it comes down to knowing your data, recognizing the few critical cases, and tailoring a specific mechanism to each of them.

Transitive Closure

It doesn't matter how much you know, there are always new examples that need further investigation. What happens if your query includes an extra predicate against the join columns, such as the following example?

```
select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t1.join1 = 20                   -- 30 distinct values
and     t2.join1 = t1.join1             -- 40 / 30 distinct values
and     t2.join2 = t1.join2             -- 40 / 50 distinct values
;
```

This is the data set that we used in `join_card_04.sql`, but the actual test script is `join_card_07.sql` in the online code suite. The comments show the number of distinct values in each column, in the order that the column names appear on the line.

The original execution plan (i.e., without the constant predicate `t1.join1 = 20`) looked like this:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=60 Card=50000 Bytes=1700000)
   1    0   HASH JOIN (Cost=60 Card=50000 Bytes=1700000)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=10000 Bytes=170000)
   3    1     TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=10000 Bytes=170000)
```

With the extra predicate in place, we now have a plan like this:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=57 Card=1667 Bytes=56678)
   1    0   HASH JOIN (Cost=57 Card=1667 Bytes=56678)
   2    1     TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=250 Bytes=4250)
   3    1     TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=333 Bytes=5661)
```

How has the optimizer managed to produce such a dramatic difference in the cardinalities all the way through the plan? (They are the right values, by the way.) The trick is revealed when you look at a more detailed execution plan (from the `dbms_xplan` package, for example).

```
-----------------------------------------------------------------------
| Id  | Operation            |  Name       | Rows  | Bytes | Cost  |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |  1667 | 56678 |    57 |
|*  1 |  HASH JOIN           |             |  1667 | 56678 |    57 |
|*  2 |   TABLE ACCESS FULL  | T2          |   250 |  4250 |    28 |
|*  3 |   TABLE ACCESS FULL  | T1          |   333 |  5661 |    28 |
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
```

```
1 - access("T2"."JOIN2"="T1"."JOIN2")
2 - filter("T2"."JOIN1"=20)
3 - filter("T1"."JOIN1"=20)
```

Note particularly the second line of `Predicate Information`. There is a predicate, `t2.join1 = 20` ..., but we didn't have that one in the original SQL, the optimizer derived it using a mechanism known as **transitive closure**. You will also note that there is no longer a predicate `t2.join1 = t1.join1`; in the process of transitive closure, this predicate became redundant and was thrown away.

Transitive closure means the optimizer is allowed to infer the following: If

```
colB = colA
```

and

```
colA = {constant X}
```

then

```
colB = {constant X}
```

In our case, we have

```
t2.join1 = t1.join1
```

and

```
t1.join1 = 20
```

so

```
t2.join1 = 20
```

and the join condition can be discarded.

So we have changed our SQL from a two-column join with one filter predicate to a single-column join with two filter predicates. So let's reprise the standard form of the basic formula, and put the numbers in—remembering that in our example, both the join columns are called `join2`, and both the filtering columns are called `join1`:

```
Join Selectivity =
        ((num_rows(t1) - num_nulls(t1.c1)) / num_rows(t1)) *
        ((num_rows(t2) - num_nulls(t2.c2)) / num_rows(t2)) /
        greater(num_distinct(t1.c1), num_distinct(t2.c2))

Join Cardinality =
        join selectivity *
        filtered cardinality(t1) * filtered cardinality(t2)

Join Selectivity =
        ((10000 - 0)/ 10000) *
        ((10000 - 0)/ 10000) /
        greater(40, 50) =             -- t1.join2, t2.join2 (num_distinct)
        1/50
```

```
Join Cardinality =
        (1 / 50) *
        10000/30 * 10000/40 =          -- t1.join1 has 30 values, t2.join1 has 40
        333 * 250 / 50 =
        1,666.66
```

You get the same result from 9*i* and 10*g*. (You may recall that when we first looked at the two-column join, we saw that 10*g* had a special *sanity check* that took both selectivities from a single table. But in this case the process of transitive closure has eliminated one of our two join predicates, so this sanity check does not take place.)

As ever, though, it is possible to break things. What's going to happen if you explicitly add the extra, redundant, predicate, so that the `where` clause now looks like this:

```
where
        t1.join1 = 20                  -- 30 distinct values
and     t2.join1 = t1.join1            -- 40 / 30 distinct values
and     t2.join2 = t1.join2            -- 40 / 50 distinct values
and     t2.join1 = 20                  -- 40 distinct values
```

This is what you get from `dbms_xplan` for 9*i* (the computed cardinality is 52 in 10*g*):

```
-------------------------------------------------------------------------
| Id  | Operation            |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |    42 |  1428 |    57 |
|*  1 |  HASH JOIN           |             |    42 |  1428 |    57 |
|*  2 |   TABLE ACCESS FULL  | T2          |   250 |  4250 |    28 |
|*  3 |   TABLE ACCESS FULL  | T1          |   333 |  5661 |    28 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):

   1 - access("T2"."JOIN1"="T1"."JOIN1" AND "T2"."JOIN2"="T1"."JOIN2")
   2 - filter("T2"."JOIN1"=20)
   3 - filter("T1"."JOIN1"=20)
```

Obviously something nasty has happened because the computed cardinality is down from a reasonably accurate 1,667 to a dangerously inaccurate 42.

Check the `Predicate Information`—the predicate `t2.join1 = t1.join1` has reappeared. The optimizer did not need to use transitive closure to generate the predicate `t2.join1 = 20`, so the join predicate never got eliminated. In effect, this means that the optimizer double-counts the selectivity on `t2.join1`, once for the join selectivity (the access), and once for the filtered cardinality (the filter). Repeating the arithmetic we did when we first saw a two-column selectivity, the join selectivity for this query is

```
Join Selectivity =
        {join1 bit} *
        {join2 bit} =
```

```
        (10,000 - 0) / 10,000) *
        (10,000 - 0) / 10,000) /
        greater(30, 40) *

        (10,000 - 0) / 10,000) *
        (10,000 - 0) / 10,000) /
        greater(50, 40) =

        1/40 * 1/50 = 1/2000
```

Then we apply this to the filtered cardinalities we got earlier on in this example to get the final result:

```
Join Cardinality =
        (1 / 2000) *
        10000 / 30 * 10000 / 40 =
        333 * 250 / 2000 =
        41.625
```

The result is different for 10*g* because of the multicolumn sanity check—instead of a join selectivity of 1/2,000, the optimizer would pick the individual selectivities from whichever table produced the smaller results (in this case `t2`, giving 1/40 * 1/40 = 1/1,600). So for 10*g* we get

```
Join Cardinality =
        (1 / 1600) *
        10000 / 30 * 10000 / 40 =
        333 * 250 / 2000 =
        52.031
```

Historically, people have found bugs with the optimizer that allowed them to *trick* it into using different execution plans by repeating simple predicates. In recent versions (9*i* and above) the optimizer has been enhanced to eliminate duplicated predicates, but it can still be caught by little quirks in the area of predicate generation by transitive closure. The preceding example is just one case where things go wrong. Another example is this:

```
select
        t1.v1, t2.v1
from
        t1,
        t2
where
        t1.join1 = 20
and     t2.join1 = t1.join1
and     t2.join1 = t1.join1                -- duplicated join predicate
and     t2.join2 = t1.join2
;
```

Notice the duplicated predicate. The optimizer ought to spot it and not consider it, and if the duplication had been on the `join2` columns, this is exactly what would have happened.

Unfortunately the query is transformed before the duplication is noticed. The optimizer eliminates one of the duplicates through transitive closure, leaving the other behind—and calculates a cardinality of 42 (9*i*) or 52 (10*g*) just like the previous example.

Moreover, as we saw in Chapter 6, the way in which transitive closure works is affected in 8*i* and 9*i* (but not 10*g*) by setting parameter `query_rewrite_enabled` to `true`. With this setting, the join predicates in my example will not be discarded as they generate the extra filter predicates. Rerun script `join_card_07.sql`, but set `query_rewrite_enabled` to true before doing so, and the join cardinality will change from the correct 1,667 we saw at the start of this section to the incorrect 42 we saw in the last example.

Three Tables

Although I said at the start of this chapter that Oracle only ever joins two tables at a time, it is worth walking through at least one example of a three-table join because it isn't intuitively obvious where the necessary selectivities will come from as the third table is joined to the previous pair.

My example (`join_card_08.sql` in the online code suite) demonstrates the awkward case where the third table is joined to columns from both the second and first tables. To keep the arithmetic easy, we start with 10,000 rows in each table. The query, and its execution plan, follows:

```
select
        t1.v1, t2.v1, t3.v1
from
        t1,
        t2,
        t3
where
        t2.join1 = t1.join1                -- 36 / 40 distinct values
and     t2.join2 = t1.join2                -- 38 / 40 distinct values
--
and     t3.join2 = t2.join2                -- 37 / 38 distinct values
and     t3.join3 = t2.join3                -- 39 / 42 distinct values
--
and     t3.join4 = t1.join4                -- 41 / 40 distinct values
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=109 Card=9551 Bytes=573060)
   1    0   HASH JOIN (Cost=109 Card=9551 Bytes=573060)
   2    1     TABLE ACCESS (FULL) OF 'T3' (Cost=28 Card=10000 Bytes=200000)
   3    1     HASH JOIN (Cost=62 Card=62500 Bytes=2500000)
   4    3       TABLE ACCESS (FULL) OF 'T1' (Cost=29 Card=10000 Bytes=200000)
   5    3       TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=10000 Bytes=200000)
```

Whatever it may look like at first sight of the execution plan, the join order for this query is t1 ➤ t2 ➤ t3. Oracle hashes t3 into memory, then hashes t1 into memory, then starts to read t2. For each row in t2 Oracle probes the t1 hash for a match, so the first join is t1 ➤ t2; and if the first probe is successful, Oracle probes the t3 hash for a match, so the second join is t2 ➤ t3—although, technically, you should say the second join is (t1 ➤ t2) ➤ t3.

Our task is to work out how the optimizer got an intermediate cardinality of 62,500 for the t1 ➤ t2 hash, and how it then got a cardinality of 9,551 by joining t3 to the intermediate result set. We just have to take it in steps.

First join t2 to t1—you will appreciate that I have avoided filter predicates and nulls just to keep the example simple—but it really is just a question of applying the formula repeatedly and using the right numbers. We have two columns in the join from t1 to t2, so we apply the selectivity for each in turn (remembering to check for the multicolumn sanity check if we are running 10*g*):

```
Join Selectivity =
        {join1 bit} *
        {join2 bit} =

        ((10000 - 0) / 10000) *
         (10000 - 0) / 10000)) /
        greater (36 , 40) *

        ((10000 - 0) / 10000) *
         (10000 - 0) / 10000)) /
        greater (38 , 40) =

        1/1600

Join Cardinality =
        1 / 1600 *
        10000  * 10000 =
        62,500
```

Now we have an intermediate table of 62,500 rows, and we have another table to join to it—this time with three join columns. Again we apply the formula:

```
Join Selectivity =

        {join2 bit} *
        {join3 bit} *
        {join4 bit} =

        ((10000 - 0) / 10000) *
         (10000 - 0) / 10000)) /
        greater( 37, 38) *                -- or should this be greater( 37, 40)
```

```
        ((10000 - 0) / 10000) *
         (10000 - 0) / 10000)) /
        greater( 39, 42) *

        ((10000 - 0) / 10000) *
         (10000 - 0) / 10000)) /
        greater( 41, 40) =
        1/65,436

Join Cardinality =
        1 / 65436 *
        62500 * 10000 =
        9,551             (as required)
```

The arithmetic raises an interesting point. In the calculation of join selectivity, I have used the `num_distinct`, `num_nulls`, and `num_rows` figures from the table that *owned* the `join3` and `join2` columns, where *ownership* is dictated by the table aliases visible in the join in the original SQL. (Hence my question in the middle of the calculation—should I be using `greater( 37, 40)` instead of `greater( 37, 38)`?)

Amongst other things, this raises the question of what you do about `null` columns (which we will deal with in the next section), and how you may inadvertently (or deliberately) change the computed cardinality by changing your choice of join predicates.

Look at the original query again; it includes the following predicates:

```
      t2.join1 = t1.join1
and   t2.join2 = t1.join2
--
and   t3.join2 = t2.join2               -- but t2.join2 = t1.join2
and   t3.join3 = t2.join3
--
and   t3.join4 = t1.join4
```

If we change the commented predicate, we see that the following would be an equally valid list of predicates. Remember, transitive closure only takes place if there is a literal constant somewhere in the chain, so we have to make this change manually; it can't happen automatically:

```
      t2.join1 = t1.join1
and   t2.join2 = t1.join2
--
and   t3.join3 = t2.join3
--
and   t3.join2 = t1.join2               -- was t3.join2 = t2.join2
and   t3.join4 = t1.join4
```

And under the 9*i*, the execution plan with this version of the join becomes the following:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=109 Card=9074 Bytes=544440)
   1    0   HASH JOIN (Cost=109 Card=9074 Bytes=544440)
   2    1     TABLE ACCESS (FULL) OF 'T3' (Cost=28 Card=10000 Bytes=200000)
   3    1     HASH JOIN (Cost=62 Card=62500 Bytes=2500000)
   4    3       TABLE ACCESS (FULL) OF 'T1' (Cost=29 Card=10000 Bytes=200000)
   5    3       TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=10000 Bytes=200000)
```

Note particularly how the computed cardinality of the final result has dropped from 9,551 to 9,074—simply because we made a small and completely valid rearrangement of the predicates (which is the sort of thing you might do for purely cosmetic reasons to conform to coding standards), and the formula necessarily changed `greater( 37, 38)` to `greater ( 37, 40)`.

Arguably this is a defect of the optimizer. We produced the intermediate cardinality of 62,500 in line 3 by using the selectivity of the `join2` column from table `t1`, and then used this same selectivity when joining the intermediate result to table `t3` to get a final cardinality of 9,074. But in the previous version of the text, we used the selectivity from table `t1` to generate the intermediate result set, and then used the selectivity from table `t2` to join the intermediate result set to table `t3`. In an ideal system, the optimizer should surely handle the selectivity of a column consistently all the way through a complex join.

There is a further impact to consider when you move to 10*g*. With the first arrangement of predicates, the final cardinality is unchanged at 9,551; but if you switch to the second arrangement of predicates, the final cardinality becomes 9,301 (rather than 9,074).

Remember the multicolumn sanity check? Take another look at the second arrangement of predicates. In the first arrangement, you see two possible sanity checks—`t2` to `t1`, and `t3` to `t2`. In the second arrangement, you also see two possible sanity checks—`t2` to `t1`, and `t3` to `t1`. It just so happened that the `t3` to `t1` sanity check kicked in on the second arrangement, so not only did we use the `t1` selectivity for column `join2`, we also used the `t1` selectivity for column `join4` when we had previously been using the `t3` selectivity for column `join4`.

If that's not confusing enough for you, you could always make matters worse. If you decide to include both `t3.join2 = t2.join2` and `t3.join2= t1.join2`, then the optimizer will use both of them in calculating the final cardinality, which then drops from several thousand down to 239. Be very careful how you write your multitable joins.

Nulls

After all the hassles with three table joins, predicate rearrangements, and sanity checks, what other complications could there possibly be to make joins difficult? If you haven't upgraded to 9*i* or 10*g* yet, then be prepared for a few surprises with `nulls`.

I wasn't entirely truthful in my earlier treatment of `nulls`—I chose to overlook a feature that is largely irrelevant when handling only two tables. However, the strategy for handling `nulls` changed between 8*i* and 9*i*. In fact, the change was so significant that the formula for join selectivity in MetaLink note 68992.1 isn't always true any more.

Based on the testing I have done, when the number of `nulls` in a join column exceeds 5% of the rows in the table, then the arithmetic changes.

Script join_card_09.sql in the online code suite builds three tables, and joins them *along* a column containing nulls. The computed cardinality for 8*i* is a long way off the computed cardinality for 9*i* and 10*g*.

```
create table t1 as
select
        mod(rownum-1,15)        n1,
        lpad(rownum,10)         v1,
        rpad('x',100)           padding
from
        all_objects
where
        rownum <= 150
;

update t1 set n1 = null where n1 = 0;
```

t1 is defined with 150 rows, and column n1 has ten copies of each of the values 1 to 14 and null. So num_rows = 150, num_distinct = 14, and num_nulls = 10.

Create tables t2 and t3 similarly but inserting 120 rows using mod(n1,12) in table t2, and 100 rows using mod(n1,10) in table t3. So table t2 ends up with the values 1 to 11 in column n1 and table t3 ends up with the values 1 to 9.

So all three tables have ten rows each for any value of column n1, and there are ten nulls in n1 in every single table—and that is enough nulls to exceed the critical 5% limit. We now run a query that joins all three tables along the n1 column and check the execution plan for 8*i* and 9*i*.

```
Select  /*+ ordered */
        t1.v1, t2.v1, t3.v1
from
        t1,
        t2,
        t3
where
        t2.n1 = t1.n1
and     t3.n1 = t2.n1
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=8 Card=9000 Bytes=378000)
   1    0   HASH JOIN (Cost=8 Card=9000 Bytes=378000)
   2    1     TABLE ACCESS (FULL) OF 'T3' (Cost=2 Card=140 Bytes=1960)
   3    1     HASH JOIN (Cost=5 Card=900 Bytes=25200)
   4    3       TABLE ACCESS (FULL) OF 'T1' (Cost=2 Card=90 Bytes=1260)
   5    3       TABLE ACCESS (FULL) OF 'T2' (Cost=2 Card=110 Bytes=1540)
```

```
Execution Plan (8.1.7.4 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=5 Card=8250 Bytes=346500)
   1    0   HASH JOIN (Cost=5 Card=8250 Bytes=346500)
   2    1     TABLE ACCESS (FULL) OF 'T3' (Cost=1 Card=150 Bytes=2100)
   3    1     HASH JOIN (Cost=3 Card=900 Bytes=25200)
   4    3       TABLE ACCESS (FULL) OF 'T1' (Cost=1 Card=100 Bytes=1400)
   5    3       TABLE ACCESS (FULL) OF 'T2' (Cost=1 Card=120 Bytes=1680)
```

9*i* has produced a computed cardinality of 9,000, while 8*i* has reached the value 8,250. You will, however, note that at line 3, both versions of Oracle have got the same cardinality (900) for the join between tables `t1` and `t2`, even though they have produced different cardinalities for each of the individual table scans on lines 4 and 5.

As a starting point, you might want to work out the *actual* number of rows returned:

- There are nine different (nonnull) values of `n1` in table `t1`, at 10 rows per value for a total of 90 rows.
- For each of those rows, there are 10 rows in table `t2`—so joining `t1` to `t2` will give 900 rows.
- For each of the joined rows, there are 10 rows in `t3`—so the final join will be 9,000 rows. And 9*i* has got the right answer.

The first clue about the difference comes from a more detailed `explain plan`. Use `explain plan` and `dbms_xplan` in 9*i* and you will find that three extra (filter) predicates have appeared from somewhere:

```
t1.n1 is not null
t2.n1 is not null
t3.n1 is not null
```

These predicates are there because they have to be true—after all if `t1.n1 = t1.n2`, then neither column can be `null`. This affects the calculations, because if they appear as filter predicates, you have to change the bit in the join selectivity formula that caters to `null` values: the `(num_rows - num_nulls) / num_rows` factor.

So the old-style 8*i* calculations look like this:

```
T1 to T2: the tables with 100 and 120 rows respectively:
Join Selectivity =
        (100 - 10) / 100) *
        (120 - 10) / 120) /
        greater(9, 11)                = 0.075

Join Cardinality =
        0.075 * 100 * 120 = 900
```

```
Intermediate to T3: the table with 150 rows:
Join Selectivity =
        (120 - 10) / 120) *       -- the CBO uses the t2 figures at one end
        (150 - 10) / 150) /       -- of the join, and the t3 at the other.
        greater(11, 14) =
        0.0611111

Join Cardinality =
        0.061111 * 900 * 150 =
        8,250
```

But the new-style 9*i* calculations look like this:

```
T1 to T2: the tables with 100 and 120 rows respectively:
Join Selectivity =
        1 / greater(9, 11) =
        0.09090909

Join Cardinality =
        0.09090909 * 90 * 110 =
        900

Intermediate to T3: the table with 150 rows:
Join Selectivity =
        1 / greater(11, 14) =
        0.0714285

Join Cardinality =
        0.0714285 * 900 * 140 =
        9,000                          (as required)
```

Note, with 9*i*, how we have factored the effects of the `is not null` predicates into the join cardinality line as the filtered cardinality of each of the individual tables. This gives us a more appropriate answer than the 8*i* strategy, which has factored the `null` count of `n1` into the calculations twice, once in the first join and then again in the second join, and so produced a final cardinality that is too low.

Of course, as you upgrade from 8*i* to 9*i*, queries involving multiple tables joining along the same nullable columns may suddenly change their execution plans because the computed cardinality goes up. If the computed cardinality has increased for one step of a query, the optimizer may decide that the next step should be a hash or merge join rather than a nested loop join.

I can't think of a rationale why this change only applies when the `null` count exceeds 5% of the row count, as it seems to be reasonable to invoke the rule across the board. But perhaps the limit has been imposed to reduce the number of queries that might change path on upgrade.

Implementation Issues

There are many ways to implement Oracle systems badly, and as a general rule, anything that hides useful information from the optimizer is a bad idea. One of the simple, and highly popular, strategies for doing this is to stick all your reference data into a single table with a `type` column. The results can be catastrophic as far as the optimizer is concerned. Unless you are very lucky, the optimizer will calculate ridiculously inappropriate cardinalities for most simple joins to this reference table. For example (see script `type_demo.sql` in the online code suite):

```
create table t1
as
with generator as (
        select  --+ materialize
                rownum          id
        from    all_objects
        where   rownum <= 3000
)
select
        trunc(dbms_random.value(0,20))        class1_code,
        trunc(dbms_random.value(0,25))        class2_code,
        rownum                                id,
        lpad(rownum,10,'0')                   small_vc
from
        generator        v1,
        generator        v2
where
        rownum <= 500000
;
```

We create a table of 500,000 rows that requires two lookups to expand meaningless code numbers into recognizable descriptions. I have restricted the test case to just a pair of reference sets with similar numbers of entries in each set to avoid some of the oddities that I mention in the next section, and to keep the arithmetic straightforward. Consider the following query:

```
select
       t1.small_vc,
       type1.description
from
        t1, type1
where
        t1.id              between 1000 and 1999
and     type1.id           = t1.class1_code
and     type1.type         = 'CURRENCY'
and     type1.description = 'GBP'
;
```

All this does is select some rows from my large table, and join to a reference table to translate a code into a description to eliminate data based on that description. This is typical of the way in which an end-user query would have to make use of the type table. (The SQL in the test case in the download does not use the literal values 'CURRENCY' and 'GBP', but I thought that a couple of meaningful code samples would help make the point of the example more clearly.)

So what does the execution plan look like? It depends on what you've done with your reference tables. Here's one option for the reference table—the currency data (or 'CLASS1' as it really was) is stored in a table all by itself:

```
create table type1 as
select
        rownum-1                id,
        'CLASS1'                type,
        lpad(rownum-1,10,'0')   description
from
        all_objects
where
        rownum <= 20
;
```

Based on this definition, an informal argument about the query would be that we are aiming for 1,000 rows from the base table, and the restriction to one description out of 20 (rownum <= 20) in the reference table will give us 50 rows in the final result set. Here's the execution plan:

```
Execution Plan (9.2.0.6 autorace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=287 Card=50 Bytes=1950)
1  0   HASH JOIN (Cost=287 Card=50 Bytes=1950)
2  1     TABLE ACCESS (FULL) OF 'TYPE1' (Cost=2 Card=1 Bytes=20)
3  1     TABLE ACCESS (FULL) OF 'T1' (Cost=284 Card=1001 Bytes=19019)
```

Sure enough, the calculated cardinality for the join is 50 rows. We can check the formulae—presented in their simplest form given we have no null values to worry about:

```
Join Selectivity = 1 / greater(20, 20) = 1/20

Join Cardinality = 1/20 * (20/20 * (500,000 * 1001/500000))  = 50
```

Now let's create a reference table that holds two sets of data and see what happens.

```
create table type2 as
select
        rownum-1                id,
        'CLASS1'                type,
        lpad(rownum-1,10,'0')   description
from
        all_objects
where
        rownum <= 20
union all
```

```
select
        rownum-1                id,
        'CLASS2'                type,
        lpad(rownum-1,10,'0')   description
from
        all_objects
where
        rownum <= 25
;

update type2 set
        description = lpad(rownum-1,10,'0')
;
```

We now have 45 rows in the reference table, and 45 distinct `descriptions`—but we now have 25 different `ID` values and two different `type` values. From a human perspective, we can identify exactly the 20 rows that belong to the join `type` and understand what is going on, but the optimizer simply does the arithmetic. Repeat the query (changing the name of the reference table) and the execution plan looks like this:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=287 Card=25 Bytes=975)
1  0   HASH JOIN (Cost=287 Card=25 Bytes=975)
2  1     TABLE ACCESS (FULL) OF 'TYPE2' (Cost=2 Card=1 Bytes=20)
3  1     TABLE ACCESS (FULL) OF 'T1' (Cost=284 Card=1001 Bytes=19019)
```

The cardinality is wrong—but not quite as wrong as I was expecting it to be. In principle, the join selectivity should have changed from 1/20 to 1/25—dictated by the unfortunate truth that the `type` we are *not interested* in has more distinct values for the `ID` than the `type` we are interested in. Moreover, the filtering on the `type2` table has changed from 1/20 to (1/2 * 1/45 = 1/90). So we hope to see the following appearing in the formulae:

```
Join Selectivity = 1 / greater(25, 20) = 1/25

Join Cardinality = 1/25 * (45/90 * (500,000 * 1001/500000)) = 20
```

It's not right—but it's easy to see where the wrong numbers have come from, the optimizer should have used 1/25 for the selectivity but in fact it seems to have used 1/20—and we can check this very precisely by examining the `10053 trace`: in this example the optimizer really has used the smaller `num_distinct` to calculate the join selectivity.

I don't yet know what the rules are that made it take that choice, but look at the exact value of the `sel` entry in the join cardinality lines that I've extracted from the `10053 trace` files from all three versions of Oracle. The optimizer has used 1/20, not 1/25, which produces a final cardinality (in 9*i* and 10*g*) of 25.

```
10g  Join Card:  25.03 = outer (0.50) * inner (1001.00) * sel (5.0000e-002)
9i   Join cardinality:  25 = outer (1) * inner (1001) * sel (5.0000e-002)  [flag=0]
8i   Join cardinality:  50 = outer (1) * inner (1002) * sel (5.0000e-002)  [flag=0]
```

It is very instructive to compare trace files across versions—sometimes you see very clearly how confusing it can be to guess what's going on. Notice how the 9*i* trace file has reported the same figures (allowing for a small rounding error) as the 8*i* trace in the *calculation*, but manages to come up with 25 as its final answer. Clearly 9*i* is tracking its intermediate results with high precision, but reporting them to the nearest integer (see the `outer(1)` in the 9*i* line in the preceding example). This move to high-precision calculations is controlled by the hidden parameter `_optimizer_new_join_card_computation`, which has the description "compute join cardinality using non-rounded input values" and defaults to `true` in 9*i*. Conveniently, the 10*g* trace file catches up with this feature and reports the intermediate results to two decimal places. Finally, 8*i* rounds and truncates as it goes—which means that in this *specific* case, it's actually going to come up with the right answer for the wrong reason.

SELECTIVITY RULES

Elsewhere in this book, I have said that there are cases where the optimizer uses 1/`num_rows` when the selectivity derived from multiple predicates falls below a critical limit. This may be true some of the time, but as you can see from the `outer(0.5)` in the 10*g* trace extract, sometimes the optimizer will use a selectivity that is smaller than 1/`num_rows`.

It is possible that my earlier assumption is actually wrong, and only *appears* to be true because of variations in the rounding strategy. It is possible that what I have observed is the optimizer rounding an intermediate result up to 1—giving the impression that there was a lower limit of 1/`num_rows` on a computed selectivity.

Of course, when things go wrong because of odd data distributions, like that `type` column, we can always fall back on generating histograms—we hope. Clearly the problem has appeared because we have two types in the same table, so let's create a cumulative frequency histogram on the type column and see what happens:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=287 Card=22 Bytes=858)
1  0   HASH JOIN (Cost=287 Card=22 Bytes=858)
2  1     TABLE ACCESS (FULL) OF 'TYPE2' (Cost=2 Card=1 Bytes=20)
3  1     TABLE ACCESS (FULL) OF 'T1' (Cost=284 Card=1001 Bytes=19019)
```

Would you believe it, the result gets worse. The optimizer has recognized that there is an uneven distribution on the `type` column, and knows that there are 20 rows with `type = 'CLASS1'` and 25 rows with `type = 'CLASS2'`, but it still has no information to tell it that the description we are supplying belongs in `'CLASS1'`, and therefore should be treated as one of 20 rather than one of 45. (In effect, because we have amalgamated several sets of data into a single set with a `type` column, we have caught ourselves in the trap of **dependent data** columns described in Chapter 6 in the section "Correlated Columns.")

In the test case I used for this chapter, the calculations are only out by a factor of two—but I engineered the test data by combining just two sets of reference data that were both about the same size. If you combine several sets of reference data, and the number of rows in the sets vary significantly, then the error can be huge.

Consider, for example, how bad the error could be when one reference data set has 3 entries, and another has 6,000—with an average number of rows per data set in the region of a couple of hundred. The cardinality of a simple join could easily be out by a factor of 100 or more.

If you find yourself in this trap, there is a simple (though possibly expensive) solution. Re-create the table as a **list partitioned** table, partitioning on the `type` column. If you do this, and if every query against the table references the `type` column as a literal value—watch out for `cursor_sharing`—then you have effectively turned the single table into one table per data set as the optimizer will use the partition level statistics to do its arithmetic. The expense, of course, comes from the license fee for the partitioning option.

Difficult Bits!

I am reluctant to call something a bug unless I can work out what Oracle is doing and can prove that it's doing something irrational. Too many people say, "It's a bug" when they really mean "I don't know why this happened."

In the case of the cost based optimizer and join cardinality, it is easy to make ridiculous numbers appear in very simple examples. Unfortunately, I can't figure out why these numbers might be appearing, so I can't decide whether I'm seeing something that might really be a bug, or might just be an unexpected side effect of a deliberately coded assumption. For example (see `join_card_10.sql` in the online code suite):

```
create table t1 as
select
        trunc(dbms_random.value(0, 100))        filter,
        trunc(dbms_random.value(0, 30 ))        join1,
        trunc(dbms_random.value(0, 20 ))        join2,
        lpad(rownum,10)                         v1,
        rpad('x',100)                           padding
from
        all_objects
where
        rownum <= 10000
;


create table t2 as
select
        trunc(dbms_random.value(0, 100))        filter,
        trunc(dbms_random.value(0, 4000 ))      join1,
        trunc(dbms_random.value(0, 50 ))        join2,
        lpad(rownum,10)                         v1,
        rpad('x',100)                           padding
from
        all_objects
where
        rownum <= 10000
;
```

We're going to do a two-column join with a filter on the two tables. Note that the filter columns have identical definitions (although the data content is randomized). More significantly, though, one of the join columns is dramatically different from its counterpart. Column `t1.join1` holds only 30 distinct values ranging from 0 to 29, column `t2.join1` holds a nominal 4,000 distinct values ranging from 0 to 3,999. (In fact, there were 3,668 distinct values in this column because of the effects of the random data generation.)

After creating the data, we run two queries—the following SQL encapsulates both of them; the second query can be constructed by switching the comment marker from one filter predicate to the other.

```
select
       t1.v1, t2.v1
from
       t1, t2
where
       t2.join1 = t1.join1
and    t2.join2 = t1.join2
and    t1.filter = 10
-- and     t2.filter = 10
;
```

Since the two filter columns are identical, to within a little random scattering, you might think that the optimizer would produce the same cardinality whichever filter column is used to identify 1% of the joined data. There is a bit of flaw in that argument in extreme cases, and Oracle seems to make some attempts to address it. These are the execution plans.

```
Execution Plan (9.2.0.6, autotrace. Filter on t1)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=57 Card=7 Bytes=266)
   1    0   HASH JOIN (Cost=57 Card=7 Bytes=266)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=100 Bytes=2000)
   3    1     TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=10000 Bytes=180000)

Execution Plan (9.2.0.6, autotrace. Filter on t2)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=57 Card=500 Bytes=19000)
   1    0   HASH JOIN (Cost=57 Card=500 Bytes=19000)
   2    1     TABLE ACCESS (FULL) OF 'T2' (Cost=28 Card=100 Bytes=2100)
   3    1     TABLE ACCESS (FULL) OF 'T1' (Cost=28 Card=10000 Bytes=170000)
```

Why are the computed cardinalities so amazingly different? According to the formulae:

```
Join Selectivity =
        {join1 bit} *
        {join2 bit} =

        1 / greater (30, 3,668) *
        1 / greater (20, 50)    =
        0.00000545256
```

```
Join Cardinality =
        0.00000545256 *
        10000  * 100 =
        5.4526
```

So why has the optimizer worked out a selectivity (which we can see in the `10053 trace`) of *exactly* 5.0000e-004 when we apply the filter to table `t2` (3,668 distinct values), and a selectivity of 7.1744e-006 when we apply the filter to table `t1` (30 distinct values)?

As you saw previously, one of the complications appears because both our test statements have a filter predicate on just one side of the join—so the standard formula is modified to use the `num_distinct` values from the other side of the join.

But then if you vary the test by generating column `t1.join1` as `dbms_random.value(0,30) + 1`, the computed cardinality of one test changes; and again, if you change the test to generate column `t1.join1` as `50 * dbms_random.value(0,30)`, the computed cardinality of one test changes quite dramatically. But remember the special calculation described in Chapter 6 where the optimizer produced some unexpected numbers when we had a small number of distinct values with a large range. In this join test, column join `t1.join1` has 30 distinct values joining to a column with a range of 4,000. Perhaps the odd numeric effects are simply the result of the same algorithm we saw before, applied in a different way.

Generally, complications set in as a result of three possible conditions:

- The number of distinct values for a column in one table is massively different from the number of distinct values in the corresponding column in the other table.
- The two ranges of values are massively different from each other.
- The product of the individual selectivities used in the join selectivity formula is much larger than the number of rows in the tables involved.

All three conditions certainly ought to have some impact on the join cardinality, but I'd prefer it if they did so in a way that seemed to have an intuitive explanation.

If you care to run this example in 10*g*, though, you will find that things have changed, and not necessarily for the better. Because of the multicolumn sanity check effect, you get the same computed cardinality whether the filter is on `t1` or `t2`. In both cases, the join selectivity switches to 0.0001 (`1/num_rows` for table `t2`), so the join cardinality is a constant 100. Unfortunately, this is much too high—and highlights another case where you may find dramatic swings in execution plans when you upgrade.

Features

Whatever else may occur, you can guarantee that it will be possible to find some optimizer bugs (or anomalies or limitations) in the leading-edge pieces of code. Here, for example, is a bug in the underlying treatment of **descending indexes** that results in an error in cardinality in 9*i* (the bug is fixed in 10*g*). The code is `descending_bug.sql` in the online code suite.

```
create table t1 as
select
        mod(rownum,200)         n1,
        mod(rownum,200)         n2,
        rpad(rownum,215)        v1
from
        all_objects
where
        rownum <= 3000
;

create table t2 as
select
        trunc((rownum-1)/15)     n1,
        trunc((rownum-1)/15)     n2,
        rpad(rownum,215)         v1
from
        all_objects
where
        rownum <= 3000
;

create index t2_i1 on t2(n1 /* desc */ );

--        Collect statistics using dbms_stats here

select
        t1.n1, t2.n2, t1.v1
from
        t1,
        t2
where
        t1.n2 = 45
and     t2.n1 = t1.n1
;
```

The cardinality reported for this query depends on whether you use a normal index, or take out the comment marks in the '`create index`' statement and use a descending index. Here are the two execution plans:

```
Execution Plan (9.2.0.6 autotrace - normal B-tree index)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=33 Card=225 Bytes=51750)
   1    0   HASH JOIN (Cost=33 Card=225 Bytes=51750)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=16 Card=15 Bytes=3330)
   3    1     TABLE ACCESS (FULL) OF 'T2' (Cost=16 Card=3000 Bytes=24000)

Execution Plan (9.2.0.6 autotrace - descending B-tree index)
```

```
------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=33 Card=1 Bytes=230)
   1    0   HASH JOIN (Cost=33 Card=1 Bytes=230)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=16 Card=15 Bytes=3330)
   3    1     TABLE ACCESS (FULL) OF 'T2' (Cost=16 Card=3000 Bytes=24000)
```

Note how the join cardinality has changed from 225 to just 1 because the index has been changed. Note, even more spectacularly, that the cardinality has been changed even though the execution plan doesn't even use the index! (I have an even more extraordinary example of this type of surprise in the script `delete_anomaly.sql` in the online code suite, where an execution plan changes because I create a bitmap index on a column that appears in the `select` list of a query—I'm not even going to try to think about why that one happens.)

The problem with the descending index is that the existence of the index has caused the optimizer to generate a surprise extra predicate, which we can see if we switch from `autotrace` to `dbms_xplan`:

```
-----------------------------------------------------------------------
| Id  | Operation            |  Name       | Rows  | Bytes | Cost  |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |     1 |   230 |    33 |
|*  1 |  HASH JOIN           |             |     1 |   230 |    33 |
|*  2 |   TABLE ACCESS FULL  | T1          |    15 |  3330 |    16 |
|   3 |   TABLE ACCESS FULL  | T2          |  3000 | 24000 |    16 |
-----------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("T2"."N1"="T1"."N1" AND
              SYS_OP_DESCEND("T2"."N1")=SYS_OP_DESCEND("T1"."N1"))
   2 - filter("T1"."N2"=45)
```

In 9*i*, the optimizer includes the `sys_op_descend()` predicate in the selectivity calculation. In 10*g*, the optimizer recognizes (I assume) that the `sys_op_descend()` predicate is the same as `t2.n1 = t1.n1`, and therefore doesn't double-count the effect.

An Alternative Viewpoint

If you feel a little uncomfortable with the way the previous pages have been describing and calculating join cardinalities, there is another way of looking at what goes on. It doesn't change what happens, or how the results drop out; it is just a different mental image that you could use to help you understand the way the arithmetic works.

Consider the query from `join_card_01.sql`:

```
select
        t1.v1, t2.v1
from
        t1,
        t2
```

```
where
        t1.filter = 1                   -- 25 values
and     t2.join1 = t1.join1             -- 50 / 30 values
and     t2.filter = 1                   -- 50 values
;
```

Apply just the simple filtering predicates and do a Cartesian join with the resulting rows. We have 400 rows extracted from table `t1`, and 200 rows extracted from `t2`, so the Cartesian join is 80,000 rows.

But there is one more predicate to apply to reduce that 80,000 rows down to the final result set, the predicate that represents the join condition. To apply this predicate, we look at it in two different ways, namely:

```
t2.join1 = :unknown_value
```

or

```
:unknown_value = t1.join1
```

Then we simply pick whichever one of these two conditions is the most selective. And, as you learned in Chapter 3, the selectivity of `column = :bind_variable` is either the `density` or `1/num_distinct`.

If there are multiple conditions, we simply apply the rules (which we also saw in Chapter 3) for combining predicates—allowing for the special sanity checks introduced by 10*g*—and the answer drops out.

I'd like to thank Benoit Dageville of Oracle Corporation for giving me this insight in a conversation we had at Oracle World 2004. I've found it a very helpful way to visualize some of the effects of different conditions.

Summary

If you are still working with 8*i*, the formulae supplied by MetaLink for join selectivity and join cardinality are correct, although the note could be enhanced to point out that

- The selectivity formula simply has to be repeated across all columns of an N-column join.
- A single join predicate involving an unbounded range-based test uses a fixed 5% selectivity.
- A single join predicate involving a bounded (between) range-based test uses a fixed 0.25% selectivity.

The join selectivity formula in a multitable join always uses the base table selectivity from the table that is explicitly named in the predicate (which is not necessarily the selectivity that was used in calculating the previous intermediate result). This means that you can make a legal textual change to rearrange the joins between tables, and find that the computed cardinality has changed.

When you move from 8*i* to 9*i*, the basic formulae supplied by MetaLink still work in many cases, but there is an alternative strategy for handling nulls, which caters to them in the join

cardinality formula rather than in the join selectivity formula. This means that some queries (multitable joins along the same nullable column, in particular) may produce a higher computed cardinality after the upgrade and therefore may change their execution plans.

When you upgrade to 10*g*, two new strategies are employed for multicolumn joins—the multicolumn sanity check, and the concatenated index sanity check. This means that the cardinality of multicolumn joins may again change (increase) on upgrade.

There are still cases where you can change the join cardinality by adding extra predicates that are technically redundant. In some cases, it will not always be intuitively obvious that the predicates are redundant. On the other hand, there are some cases of code fixes in 9*i* and 10*g* that identify some of the classic cases of redundant predicate and ignore them. This means that some queries may show a dramatic increase in join cardinality when you upgrade from 8*i* to 9*i* or 10*g*.

Joins involving *badly matched* tables, may produce better cardinality calculations, hence better plans, if you create histograms on one, or both, ends of the worst-matching columns.

Test Cases

The files in the download for this chapter are shown in Table 10-5.

Table 10-5. *Chapter 10 Test Cases*

Script	Comments
`join_card_01.sql`	Simple case demonstrating the join selectivity and cardinality formulae
`join_card_02.sql`	Demonstration that the formulae still work with nulls in the join columns
`join_card_03.sql`	Demonstration that the formulae still work with nulls in the join columns and filter columns
`join_card_01a.sql`	Example of the "wrong" `num_distinct` being used in the formula
`join_card_04.sql`	Two join predicates with an **AND** between them
`join_card_05.sql`	Two join predicates with an **OR** between them
`join_card_06.sql`	Effects of variation in low/high values of the joined column
`join_card_07.sql`	Effects of transitive closure (predicate generation) on joins
`join_card_08.sql`	Joining three tables
`join_card_09.sql`	Changes in `null` treatment—three-table join
`type_demo.sql`	Effects of storing all reference data in a single table
`join_card_10.sql`	Is this a bug, a side effect, or three simultaneous special cases?
`descending_bug.sql`	A bug in cardinality where descending indexes exist
`delete_anomaly.sql`	A strange example of cardinality changing
`setenv.sql`	Sets a standardized environment for SQL*Plus

CHAPTER 11

Nested Loops

After working out the cardinality of a join, the optimizer then has to consider the cost of the three ways of performing that join, nested loop, hash, or merge join. As I pointed out in Chapter 9, the star join and star transformation are not join mechanisms, they are join strategies, so they don't get a mention from this point onwards. The next three chapters examine each of the three join mechanisms in turn, first examining the operation and then the cost calculations.

As we go through the mechanisms, it is important to remember that several contradictory effects may appear. The run-time engine may not do exactly what the execution plan *appears* to say; the cost calculation may also not tally with what the plan *seems* to say, and the cost calculation may not tally with what the run-time engine *actually* does.

We start with the nested loop join, as that is the easiest to visualize, and the one whose cost calculation is best known.

Basic Mechanism

Consider the following query:

```
select
        t1.cola,
        t2.colb
from
        table_1         t1,
        table_2         t2
where
        t1.colx = {value}
and     t2.id1  = t1.id1
;
```

If we tried to write the code to emulate a nested loop join on these two tables, it might look something like the following (and, alas, people frequently do write PL/SQL or Java code that loops exactly like this):

```
for r1 in (select rows from table_1 where colx = {value}) loop
    for r2 in (select rows from table_2 that match current row from table_1) loop
        output values from current row of table_1 and current row of table_2
    end loop
end loop
```

Looking at this code, you see two loop constructs. The outer loop works through `table_1` and the inner loop works (possibly many times) through `table_2`. Because of the structure of this pseudo-code, the two tables in a nested loop are commonly referred to as the **outer table** and the **inner table**. The outer table is also commonly referred to as the **driving table** (although I don't think I've ever heard the inner table referred to as the driven table).

"OUTER" AND "INNER" TABLES

The terms *outer* and *inner* are really only appropriate to nested loop joins. When talking about hash joins, you ought to refer to the *build table* and *probe table*; and for merge joins, the terms *first table* and *second table* are sufficient. However, you will find that the `10053 trace` file always uses the terms *outer* and *inner* to identify the first and second tables respectively in a join operation. I will revisit this point in the relevant chapters.

The execution plan for a nested loop join with an index on the inner table can have two different forms from 9*i* onward: one when the optimizer uses the index on the inner table for a **unique scan**, and another when the optimizer uses the index for a **range scan**. The second form ceases to be an option, however, if the outer table is guaranteed to return a single row.

```
Execution Plan (9.2.0.6 autotrace - unique access on inner table (traditional))
----------------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=324 Card=320 Bytes=11840)
1  0   NESTED LOOPS (Cost=324 Card=320 Bytes=11840)
2  1     TABLE ACCESS (FULL) OF 'DRIVER' (Cost=3 Card=320 Bytes=2560)
3  1     TABLE ACCESS (BY INDEX ROWID) OF 'TARGET' (Cost=2 Card=1 Bytes=29)
4  3       INDEX (UNIQUE SCAN) OF 'T_PK' (UNIQUE) (Cost=1 Card=1)

Execution Plan (9.2.0.6 autotrace - range scan on inner table (new option))
----------------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=322 Card=319 Bytes=11803)
1  0   TABLE ACCESS (BY INDEX ROWID) OF 'TARGET' (Cost=2 Card=1 Bytes=29)
2  1     NESTED LOOPS (Cost=322 Card=319 Bytes=11803)
3  2       TABLE ACCESS (FULL) OF 'DRIVER' (Cost=3 Card=319 Bytes=2552)
4  2       INDEX (RANGE SCAN) OF 'T_PK' (UNIQUE) (Cost=1 Card=1)
```

The second form of the nested loop join appeared in 9*i*, and represents a cunning optimization—which I understand is known as **table prefetching**—that can reduce the logical I/O count (hence latching, and possibly physical I/O count) on larger nested loop joins.

You will note that the way the costs are displayed in the new plan doesn't reflect the modified form of the plan (or the notional saving in resources). The execution plan has simply moved the reference to the second table (line 3 in the traditional plan) to a point outside the nested loop (i.e., to line 1 in the new form of the plan). In principle, you might expect the costs and cardinality of the new plan to look more like the following:

```
Execution Plan (Notional, with full disclosure of prefetch costing)
----------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=322 Card=319 Bytes=11803)
1  0   TABLE ACCESS (BY INDEX ROWID) OF 'TARGET' (Cost=322 Card=319 Bytes=11803)
2  1     SORT (ROWID LIST)  (Cost=??? Card=319 Bytes=???)
3  2       NESTED LOOPS (Cost=??? Card=319 Bytes=???)
4  3         TABLE ACCESS (FULL) OF 'DRIVER' (Cost=3 Card=319 Bytes=2552)
5  3         INDEX (RANGE SCAN) OF 'T_PK' (UNIQUE) (Cost=1 Card=1)
```

In fact, whichever form you see in the execution plan, the run-time engine may still use the mechanism shown in the traditional plan—to date, I have only seen the execution engine use the new mechanism in one very special case.

The examples I have picked to show you the plan structure comes from a test case that also happens to show an interesting feature in later versions of 9*i*—both plans came from exactly the same query (see script `prefetch_test_02.sql` in the online code suite), where the theoretical access to the second table should have been a unique scan for a single row on the primary key index.

The reported execution plan actually switched mechanisms as the number of rows in the driving table changed—as you can see in the line with the `TABLE ACCESS (FULL) OF 'DRIVER'`, at 320 rows the traditional plan was reported, but at 319 rows the optimizer reported that it would switch to a range scan so that it could use the new mechanism. This test case happens to be an example of the one specific and unrealistic case where the new mechanism actually operates at run time, and if you care to check the logical I/O and latch activity while running the test case, you will find that there are some very clear changes in resource usage.

The script is in the online code suite, but you may find that you have to experiment to find the break point every single time you try the test. The exact test is not reproducible, and probably depends on recent activity on your system (script `prefetch_test.sql` runs the tests on autopilot, and `prefetch_test_01.sql` demonstrates that the effect depends on CPU costing being enabled). When I examined the `10053 trace` file for the two different plans, I was unable to spot any difference anywhere that could explain *why* the optimizer had changed its choice of plan. The test results were not reproducible on 10*g*.

There are two fairly standard pictures used to represent the nested loop join. Each has its strengths and weaknesses as a tool for explaining what is going on. The first picture, shown in Figure 11-1, simply connects rows from one table with rows from another, using arrows to indicate direction of activity. In a monochrome diagram, this makes it easy to see the connection between rows of one table and their partners in the other table.

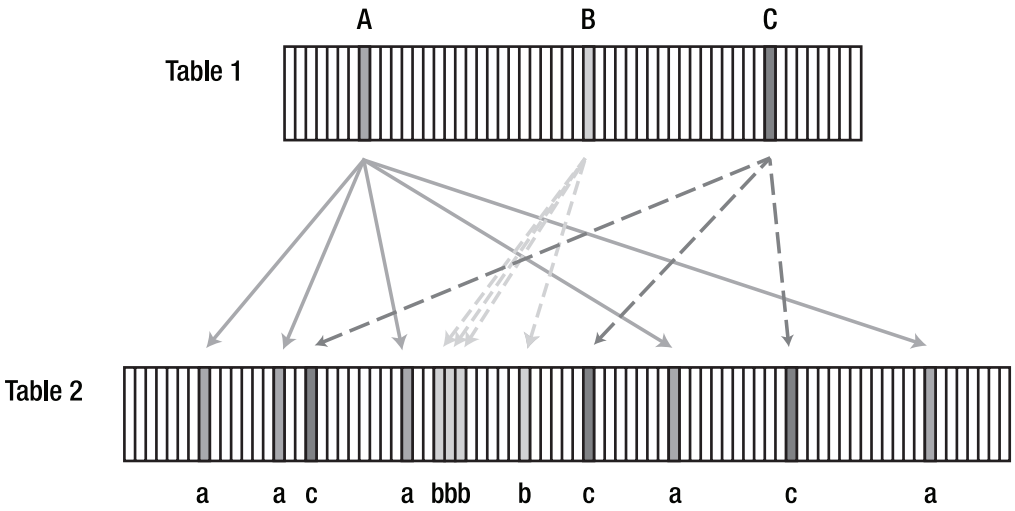

Figure 11-1. *Nested loop join*

The second picture, shown in Figure 11-2, includes a representation of working through an index on the second table, because an index is usually involved in this way when there is a nested loop around.

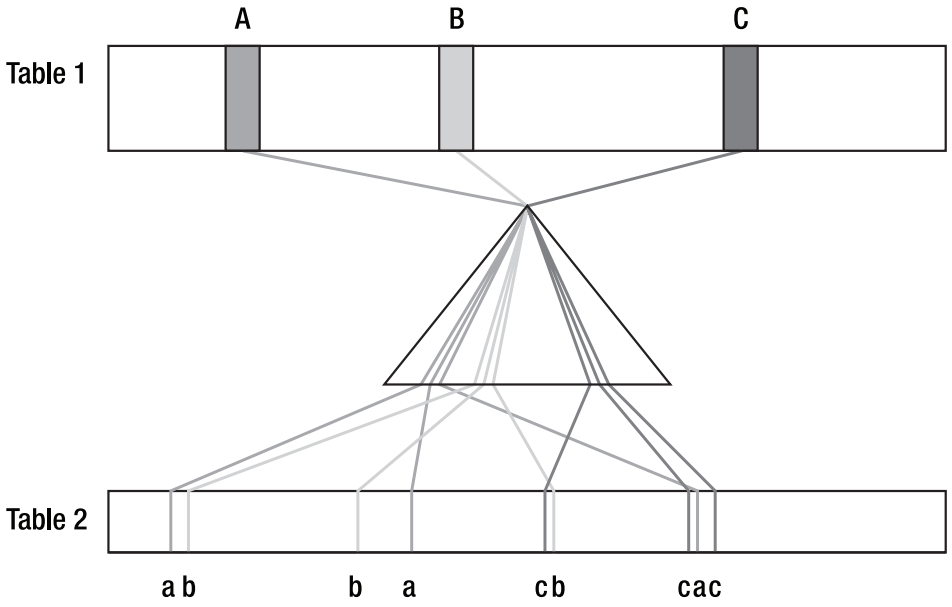

Figure 11-2. *Nested loop join with index visible*

Figure 11-2 gives us a visual hint of why the parameter `optimizer_index_caching` might be an appropriate parameter to set for a typical OLTP system. This parameter defaults to 0 (for

backward compatibility), and takes values from 0 to 100. It is used to indicate to the optimizer the percentage of index blocks from the inner (second) table's index that are likely to be cached during nested loop joins.

OPTIMIZER_INDEX_CACHING

`optimizer_index_caching` is used to adjust the cost calculation for index blocks of the inner table in nested loops and for the index blocks used during in-list iterators. It is not used in the calculation of costs for simple index unique scans or range scans into a single table.

Given that OLTP systems often have joins where "a few rows" from the first table are used to drive a join into the second table to pick up "a few rows" each, a value in the region of 75 is a reasonable starting guess (stress the word *guess*) for typical OLTP systems. The dense packing of the content in a typical index makes it seem reasonable that an indexed access path might do a physical read on the first pass through the loop, but that physical read would have populated a buffer that subsequent passes could reuse.

The second picture also has the advantage that it allows us to visualize both the old and new mechanisms for the nested loop.

The old mechanism finds the first row in the outer (driving) table, traverses the index, and visits each of the matching rows from the inner table in turn; then repeats for the second and third rows in the outer table. This results in the rows from table 2 being fetched in the order (a, a, a, b, b, b, c, c, c), which may allow a subsequent `order by` clause to achieve a `SORT (ORDER BY) NOSORT`.

The new mechanism finds the first row in the outer table, traverses the index, and stops in the leaf block, picking up just the relevant rowids for the inner table; then repeats for the second and third rows in the outer table. When all the target rowids have been found, the engine can sort them and then visit the inner table in a single pass, working along the length of the table just once, picking the rows in whatever order they happen to appear—in this case (a, b, b, a, a, c, b, c, a, c).

This may take fewer logical I/Os—the diagram suggests that the first two rows in the inner table may be in the same block, which means they could be acquired with one **consistent get**, rather than two. The mechanism may even result in fewer physical I/Os—that one block with the first two rows may have been flushed from the buffer between the two visits that would be needed using the traditional mechanism. On the down side, this strategy may mean that an `order by` clause could need a sort that would not otherwise be necessary. Clearly, the choice of plan needs to be cost-based; however, it may also be affected by a cache-monitoring routine carried out by the **CKPT** (**checkpoint**) process—as `event 10299` is described by the comment "Trace prefetch tracking decisions made by CKPT."

Looking at either picture, it is quite easy to see that a simple approximation to the cost of performing a nested loop could be described by this rationale:

- What is the cost of getting all the required rows from the first table?
- How many rows will appear in the first table?
- What is the typical cost of finding related rows once in the second table, given the newly available information I have from the current row in the first table?

■ **Note** An important detail that is often overlooked when people are thinking about nested loops is that we have picked some columns from the first table before we start our search in the second table, so we may be able to use more precise access paths into the second table. (There was a time when Oracle used to advise prefixing of local indexes on partitioned table for exactly this reason—but that ceased to be an automatic concomitant of partitioning a long time ago.)

With these three questions in mind, we can produce a very simple formula for the cost of a nested loop, which is as follows:

```
Cost of acquiring data from first table +
    Cardinality of result from first table * Cost of single visit to second table
```

For example, quoting a small section of a `10053 trace` from 9*i*, we see the following on a typical nested loop join calculation:

```
NL Join
  Outer table: cost: 14847  cdn: 4973  rcz: 35  resp:  14847
  Inner table: T2
    Access path: tsc  Resc: 51
    Join:  Resc:  268470  Resp:  268470
```

This translates as the serial cost (`resc`) of joining table `t2` to the previous table (which may be an intermediate result set from a prior join) is 268,470. This comes from

```
14,847 +    -- cost of getting data from outer table
4973 *      -- number of rows in outer table
51          -- cost of accessing inner table (t2) once
```

You will notice the two `resp` figures—these show the cost of using parallel execution, but since the trace file came from a query where the degree of parallelism was one, this matches the serial cost.

Worked Example

Looking back a few pages to the original example (`prefetch_test_02.sql` in the online code suite), you may start to question whether this formula is actually correct. Here's the (traditional) execution plan again:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=324 Card=320 Bytes=11840)
1  0   NESTED LOOPS (Cost=324 Card=320 Bytes=11840)
2  1     TABLE ACCESS (FULL) OF 'DRIVER' (Cost=3 Card=320 Bytes=2560)
3  1     TABLE ACCESS (BY INDEX ROWID) OF 'TARGET' (Cost=2 Card=1 Bytes=29)
4  3       INDEX (UNIQUE SCAN) OF 'T_PK' (UNIQUE) (Cost=1 Card=1)
```

Line 2 tells us that the cost of scanning the driver table was 3, with 320 rows returned.

Line 3 tells us that the cost of one trip to the target table was 2. Looking at line 4, we can infer that that cost was made up of 1 for hitting the index, plus 1 for hitting the table itself.

Applying the formula, then, we should see

```
Total cost = 3 + (320 * 2) = 643.
```

But the reported cost is 324—so what went wrong? The answer lies in a combination of rounding errors and the way that intermediate results are reported.

I had CPU costing enabled for the example, with the following settings:

```
begin
        dbms_stats.set_system_stats('MBRC',8);
        dbms_stats.set_system_stats('MREADTIM',20);
        dbms_stats.set_system_stats('SREADTIM',10);
        dbms_stats.set_system_stats('CPUSPEED',500);
end;
/
```

This means, in particular, that the CPU runs at 500 MHz (or 500 million Oracle operations per second) and that a single block read takes 10 milliseconds. Put another way, 5,000,000 operations equate to one single block read.

If you look at the output from a full report on the plan_table (which will include the I/O costs and CPU costs as separate items), you will find the following figures (I have excluded many details, including the cardinality from this output):

```
Execution Plan (9.2.0.6 - Queried from plan_table)
----------------------------------------------------------
0    SELECT STATEMENT (all_rows) IO Cost = 322, CPU = 7010826
1  0   NESTED LOOPS     IO Cost = 322, CPU Cost =7010826
2  1     TABLE ACCESS DRIVER (full)  IO Cost = 2 CPU Cost = 68643
3  1     TABLE ACCESS TARGET (by index rowid)  IO Cost = 1, CPU Cost = 21695
4  3       INDEX UNIQUE T_PK (unique scan)  IO Cost = 0, CPU Cost = 14443
```

Check the I/O costs, and you see that they match the following formula:

```
322 (line 1) = 2 (line 2)  + 320  * 1 (line 3)
```

Check the CPU costs, and you see that they match this formula:

```
7,010,826 (line 1) = 68,643 (line 2)  + 320  * 21,695 (line 3)
        (actually it comes to 7,011,043, which is an error of 217 out of 7M)
```

Given that the optimizer equates 5,000,000 operations with one (single block) I/O, you can now see what has happened in the autotrace report. Oracle has reported the line cost as IO cost + CPU cost / 5,000,000:

```
Cost = IO Cost + Adjusted CPU cost
 324 =     322 + ceiling(7,010,826 / 5,000,000)
   2 =       1 + ceiling(   68,643 / 5,000,000)
   1 =       0 + ceiling(   21,695 / 5,000,000)
```

So the simplified output from autotrace has introduced a point of confusion—it reports only integer values; and the rounding strategy in 9*i* has made things worse—it always rounds up. (The problem is much less visible in 10*g*, which rounds to the nearest integer and therefore manages to produce self-consistent figures much more frequently.)

Sanity Checks

In Chapter 6, I mentioned that one of the things we would see in this chapter was the special case where the standard formula for indexed access into a table is not used. The following extract generates a slightly unusual data set, and then queries it with a simple two-table query with a three-column join. (There are two related scripts in the online code suite—for comparative purposes script join_cost_03.sql in the online code suite shows the same test case before we fix the data to have the unusual distribution, and join_cost_03a.sql is the case with the modified data.)

```
create table t1 as
select
        rpad('x',40)                    ind_pad,
        trunc(dbms_random.value(0,25))  n1,
        trunc(dbms_random.value(0,25))  n2,
        lpad(rownum,10,'0')             small_vc,
        rpad('x',200)                   padding
from
        all_objects
where
        rownum  <= 10000
;

--
--        The critical data fix - done only in join_cost_03a.sql
--

update t1 set n2 = n1;
commit;

create index t1_i1 on t1(ind_pad,n1,n2)
pctfree 91
;

create table driver as
select
        rownum id,
        ind_pad, n1, n2
from        (
        select distinct ind_pad, n1, n2
        from t1
)
```

```
;

alter table driver add constraint d_pk primary key(id);

--        Collect statistics using dbms_stats here

select
        t1.small_vc
from
        t1
where
        t1.ind_pad = rpad('x',40)
and     t1.n1 = 0
and     t1.n2 = 4
;

select
        /*+ ordered use_nl(t1) index(t1 t1_i1) */
        t1.small_vc
from
        driver d,
        t1
where
        d.id        = 5
and     t1.ind_pad  = d.ind_pad
and     t1.n1       = d.n1
and     t1.n2       = d.n2
;
```

In the baseline test, there are 625 different combinations for the pair (`n1`, `n2`) in table `t1`, so the `driver` table has 625 rows, but in the special test case where we copy `n2` into `n1`, the `driver` table has 25 rows. Since there are 10,000 rows in `t1`, and the (`n1`, `n2`) combinations are randomly distributed, we can assume that there are 16 rows (10,000 / 625) per pair of values in the baseline test, and 400 rows (10,000 / 25) in the modified test.

Because of the primary key constraint on the `driver` table, the optimizer "knows" that the first predicate (`d.id = 5`) in the second query will result in precisely one row being identified in the `driver` table.

To emphasize the degree of variation from the standard formula, the first query we test is a single table indexed access into `t1`. According to the standard nested loop join formula, it is the cost from this operation that should be multiplied by the cardinality from the driving table when we switch to the two-table join. The plan we get from this single-table query is as follows:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=14 Card=16 Bytes=928)
1  0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=14 Card=16 Bytes=928)
2  1     INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=4 Card=16)
```

Although we know that this cardinality of 16 is actually wrong, the plan seems to have followed the standard cardinality formula for single table access quite correctly. To reprise the working for selectivity:

- Selectivity of ind_pad = 1
- Selectivity of n1 = 1 / 25
- Selectivity of n2 = 1 / 25
- Total selectivity = 1 / 625
- Hence cardinality = 10,000 / 625 = 16

Similarly, we can check view user_indexes for the values for the blevel (2), leaf_blocks (1,107), and clustering_factor (6,153), and put them into the standard cost formula:

```
Cost =   blevel +
           ceiling(selectivity * leaf_blocks) +
           ceiling(selectivity * clustering_factor)
   =    2 + ceiling(1,107 / 625)  + ceiling(6,153 / 625)
   =    2 + 2 + 10
   =    4 + 10          -- note the cost of the index (line 2 of the execution plan)
   =    14              -- the total cost of the table access (line 1 of the plan)
```

But look what happens when we generate the execution plan of the join. We know that we are going to pick up just one row from the driver table, so the cardinality of the final result should not be changed from the 16 that we got from the first query, and the cost ought to be

```
Cost of acquiring the one row from the driver +
   1 (cardinality from driver) * 14 (cost as for single-table query)
```

Instead we see

```
Execution Plan (8.1.7.4 or 9.2.0.6, or 10.1.0.4 autotrace)
----------------------------------------------------------
0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=292 Card=16 Bytes=1712)
1   0   NESTED LOOPS (Cost=292 Card=16 Bytes=1712)
2   1     TABLE ACCESS (BY INDEX ROWID) OF 'DRIVER' (Cost=1 Card=1 Bytes=49)
3   2       INDEX (UNIQUE SCAN) OF 'D_PK' (UNIQUE)
4   1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=291 Card=16 Bytes=928)
5   4       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=45 Card=16)
```

You will notice that this execution plan uses the traditional nested loop structure even in 9*i* and 10*g*—remember that one of the conditions I mentioned originally for the use of the new structure was that the driving table should return more than one row.

The cardinality for the execution plan is indeed unchanged at 16, but look what's happened to the cost. The cost of the index line has jumped from 4 to 45, and the incremental cost of visiting the table has jumped from 10 to (291 – 45 =) 246. Where did these numbers come from?

A quick trip back to the view user_indexes tells us the answer:

```
select
        blevel,
        avg_leaf_blocks_per_key,
        avg_data_blocks_per_key
from
        user_indexes
where
        table_name = 'T1'
and     index_name = 'T1_I1'
;

    BLEVEL AVG_LEAF_BLOCKS_PER_KEY AVG_DATA_BLOCKS_PER_KEY
---------- ---------------------- ----------------------
         2                     44                    246
```

- The cost of the index line is `blevel` + `avg_leaf_blocks_per_key` – 1.
- The incremental cost of the table line is `avg_data_blocks_per_key`.

In the special case of a join that uses equality on the entire index, the optimizer changes its cost calculation to use the precalculated values stored in the `user_indexes` view (although I cannot explain the appearance of that "minus 1"). And, as we saw in Chapter 6, these values are based on knowing the actual number of distinct keys, rather than multiplying together the individual selectivities of the column involved. So, in theory, this should give us a more realistic value for the cost of the join.

Unfortunately, the calculation of cardinality in this example did not use the same option (although, as we saw in Chapter 10, 10*g* does have a sanity check based on `concatenated index cardinality` that sometimes comes into play), so there is a serious discrepancy between the cost and the cardinality.

In a more complex case, this discrepancy could encourage the optimizer to put this pair of tables at a very early point in an execution plan—it seems to get a small number of rows, and they are expensive rows. Since the actual number of rows is much larger than the optimizer has estimated, the knock-on effects further down the plan could be resource-intensive at run time.

In principle, of course, although this particular special case ought to be beneficial, it does have an interesting, and sometimes painful, side effect. You may decide to add an extra column to an index to increase precision and reduce the number of redundant rows that some queries examine in a target table. Alternatively, you may choose to add a column to an index so that a popular query can become an index-only query and not visit the underlying table. (We know, of course, from Chapter 5 that this may have an undesirable effect on the cost calculations because of its impact on the `clustering_factor`.)

But if you have other queries that used to use the entire index with equality but don't use the new column, those queries will suddenly change their costing arithmetic from the *stored result* method to the *product of selectivities* method. And as we have just seen, there are some indexes where this change could result in a very large change to the cost calculation, which could result in a dramatically different execution plan for the join.

Summary

The cost of a nested loop is simple to calculate—but there are special cases, and the most dramatic special case produces yet another reason for checking carefully when you want to add columns to indexes.

Test Cases

The files in the download for this chapter are as shown in Table 11-1.

Table 11-1. *Chapter 11 Test Cases*

Script	Comment
`prefetch_test_02.sql`	Demonstration that there are two different nested loop plans for a unique access
`prefetch_test.sql`	Automated sweep through 999 configurations
`prefetch_test_01.sql`	Baseline test, without CPU costing to show that the plans don't vary
`join_cost_03.sql`	Demonstration of special case calculation of cost—baseline
`join_cost_03a.sql`	Demonstration of special case calculation of cost—adjusted to show special case
`setenv.sql`	Sets a standardized environment for SQL*Plus

CHAPTER 12

Hash Joins

Before saying anything about the method that the optimizer uses to estimate the cost of a hash join, I shall describe the mechanics. There are two good reasons for doing this. First, the mechanism is not commonly known; second, you have to know the mechanism before you can hope to guess how the cost calculation works.

In all previous chapters, I have disabled CPU costing (the 9*i* system statistics feature) and worked with a manual `workarea_size_policy` rather than using the automatic option introduced in 9*i*. This is often a reasonable strategy to adopt because it doesn't materially affect the work that the optimizer does—until you get to hashing and sorting.

In the next two chapters, therefore, I shall be covering the four different options: CPU costing on and off, with `workarea_size_policy` set to `automatic` or `manual`.

Since hash joins can operate at three levels of effectiveness (reported in `v$sysstat` under `optimal`, `onepass`, and `multipass` workarea executions), this introduces a nominal 12 possibilities to cover. I don't intend to wade through all 12 of them in detail, but will cover the key points of the different levels of effectiveness, and comment on critical differences that appear because of the four different environmental options

Perhaps the most significant issue we have to resolve is the problem that the formula from the 9.2 Performance Guide and Reference for the cost that I quoted in Chapter 1 does not contain any obvious I/O-related components that allow for a hash join spilling from memory to disk; in the absence of any solid information, we shall have to proceed by trial and error. Another unfortunate detail we will have to face is that the different versions of the optimizer can produce startlingly different costs for the same hash join under the same conditions.

Getting Started

When we do a hash join, we acquire one data set and convert it into the equivalent of an in-memory **single-table hash cluster** (assuming we have enough memory) using an internal hashing function on the join column(s) to generate the hash key.

We then start to acquire data from the second table, applying the same hashing function to the join column(s) as we read each row, and checking to see whether we can locate a matching row in the in-memory hash cluster.

Since we are using a hashing function on the join column(s) to randomize the distribution of data in the hash cluster, you will appreciate that a hash join can only work when the join condition is an equality. You could argue that `not exists` is another possible condition, but this is really using an equality with the hope of failing.

We refer to the first table as the **build table** (we "build" the in-memory hash cluster from it), and we refer to the second table as the **probe table** (we "probe" the in-memory hash cluster with it).

WHICH IS THE "OUTER" TABLE?

It is an unfortunate accident that there is no intuitive interpretation for the terms *outer table* and *inner table* for a hash join (as there is with the traditional nested loop join).

It is probably for this reason that the manuals have apparently described the mechanism of the hash join back to front for so long. At some stage, one of the manual writers must have assumed that the phrase *inner table* simply meant "the first one." Consequently, the manuals have stated for years that the inner table is used to build the hash cluster—even though every other reference you will find to the terms *inner* and *outer* (e.g., in the description of nested loops, in the `10053 trace` file, and in the `pq_distribute` hint) shows that the expression *inner table* is supposed to identify the second table in the join order.

Let's create an example so that we can see the different bits of this process in action. As usual, my demonstration environment starts with an 8KB block size, locally managed tablespaces, uniform extents of 1MB, manual segment space management, and (despite my opening comments) system statistics disabled and a manual setting for the `hash_area_size` (see script `hash_opt.sql` in the online code suite):

```
alter session set workarea_size_policy = manual;
alter session set hash_area_size = 1048576;

create table probe_tab as
select
        10000 + rownum                          id,
        trunc(dbms_random.value(0,5000))        n1,
        rpad(rownum,20)                         probe_vc,
        rpad('x',500)                           probe_padding
from
        all_objects
where
        rownum <= 5000
;

alter table probe_tab add constraint pb_pk primary key(id);

create table build_tab as
select
        rownum                                  id,
        10001 + trunc(dbms_random.value(0,5000))    id_probe,
        rpad(rownum,20)                         build_vc,
        rpad('x',500)                           build_padding
from
        all_objects
```

```
where
        rownum <= 5000
;

alter table build_tab add constraint bu_pk
        primary key(id);

alter table build_tab add constraint bu_fk_pb
        foreign key (id_probe) references probe_tab;

--   Collect statistics using dbms_stats here

select
        bu.build_vc,
        pb.probe_vc,
        pb.probe_padding
from
        build_tab        bu,
        probe_tab        pb
where
        bu.id between 1 and 500
and     pb.id = bu.id_probe
;

Execution Plan (9.2.0.6 Autotrace)
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=106 Card=500 Bytes=278500)
1  0  HASH JOIN (Cost=106 Card=500 Bytes=278500)
2  1    TABLE ACCESS (BY INDEX ROWID) OF 'BUILD_TAB' (Cost=42 Card=500 Bytes=15000)
3  2      INDEX (RANGE SCAN) OF 'BU_PK' (UNIQUE) (Cost=3 Card=500)
4  1    TABLE ACCESS (FULL) OF 'PROBE_TAB' (Cost=60 Card=5000 Bytes=2635000)
```

I have designed the sample data to demonstrate several points about hash joins. In particular, I constructed the tables in a way that allows me to demonstrate the error in one commonly (or perhaps carelessly) stated comment about hash joins. The hash join is almost invariably described as a join mechanism that does tablescans, but this is not a necessity. As you can see, my example acquires one of its data sets through an indexed access path.

Another of the comments that is made too casually is that a hash join is good for joining a small table to a large table (and the terms *small* and *large* are rarely qualified). If you check the statistics of these two tables, you will find that they are both the same size. The comment about small and large tables should really be stated in terms of the small and large data sets you have extracted from the tables—and even then you have to describe exactly what you mean if you don't want to be contradicted. The demonstration script includes a few variations on the query to expand on these points.

So what has the optimizer decided to do with the query? Obviously one option would have been to perform a nested loop join using the index on `probe_tab(id)` to collect the related row from `probe_tab` as we acquired each row from `build_tab`; but the cost of using a nested loop

was too high—the cost when hinted was 542. Similarly, we might have seen a merge join, but again this would have been too expensive—the cost when hinted was 445.

To get a picture of how Oracle performs the join, you can imagine it starts by splitting the SQL into two completely separate components:

```
select
        bu.id,
        bu.build_vc,
        bu.id_probe
from
        build_tab        bu
where
        bu.id between 1 and 500
;

select
        pb.probe_vc,
        pb.probe_padding,
        pb.id
from
       probe_tab        pb
;
```

The two components select the join columns and all the referenced columns in their `select` lists, using any available predicates from the main query to restrict the row selection. In this case, this mechanism allows the pseudo-query against `build_tab` to be reasonably selective, but the pseudo-query against `probe_tab` has no row restrictions at all. This lack of restrictions on the probe table explains why partition elimination often fails to occur when the second table in a hash join is a partitioned table. Unless the optimizer decides to execute a preliminary query (known as a **pruning subquery**) against the build table, or the decomposed query against the probe table includes a reference to the partitioning column(s), then there is nothing to tell Oracle how to eliminate partitions.

You will notice that I've included the `ID` column in the `select` list for `build_tab`. This seems to be unnecessary, but the optimizer allows for it in the calculations, so I've put it there.

After generating the two pseudo-queries, the optimizer estimates the number of rows and row size (hence total volume of data) of the two data sets. The cost and cardinality of the two queries comes from the standard calculations. The `bytes` figure is derived from the cardinality of the queries and (usually) the values from the column `avg_col_len` in `user_tab_columns`.

HOW BIG IS A "ROW"?

As a general rule, the figures for `bytes` in execution plans are derived from the `avg_col_len` columns of `user_tab_columns`. The deprecated `analyze` command excludes the length byte(s) for the column, but the call to `dbms_stats.gather_table_stats` includes the length byte(s). Since the choice of build table in a hash join is affected by the size of the data sets involved, a switch from `analyze` to `dbms_stats` could (in principle) change the order of a hash join, or even cause the optimizer to use a different join mechanism.

For the special case of `select * from table`, the optimizer seems to use the `avg_row_len` from `user_tables` as the row size if the statistics have been generated by `dbms_stats`, but `sum(avg_col_len)` if the statistics have been generated by `analyze`. (The optimizer can identify how the statistics were generated by checking the `global_stats` column of `user_tables` or `user_tab_columns`.)

If we check the column names and column lengths from the view `user_tab_columns`, we get the results shown in Table 12-1.

Table 12-1. *Calculating the Row Size*

Table	Column	avg_col_len (dbms_stats)	Total for Table (dbms_stats)	avg_col_len (analyze)	Total for Table (analyze)
Build_tab	Build_vc	21		20	
Build_tab	Id	4		3	
Build_tab	Id_probe	5	30	4	27
Probe_tab	Id	5		4	
Probe_tab	Probe_padding	501		500	
Probe_tab	Probe_vc	21	527	20	524

So for `build_tab`, we have 500 rows at 30 bytes for a total of 15,000 bytes; for `probe_tab`, we have 5,000 rows at 527 bytes for a total of 2,635,000 bytes—as we saw in the execution plan. If you look closely at the execution plan, you will also infer that the output row size that the optimizer is using is 557 (from 278,500 / 500)—which means it has simply added the lengths of the input rows, and has not allowed for double-counting of the join columns.

The figures for `bytes` would change to 13,500 bytes, 2,620,000 bytes, and 275,500 bytes if we switch from the `dbms_stats` package to the old `analyze` command.

After working out the data size, the optimizer identifies the smaller set as the build table, and starts working on building a hash table. Since we have a `hash_area_size` of 1MB, and only 15,000 bytes of data, we can be confident that the hash table will fit in memory, and we won't have to spill to disk. This is what gives us a definition of a "small data set"—it is one that fits entirely within the `hash_area_size`, even after allowing for some overheads.

The Optimal Hash Join

Hash joins fall into the category of **workarea executions** and are categorized from 9*i* onwards as `optimal`, `onepass`, or `multipass` in `v$sysstat` where you can find the statistics `'workarea executions - optimal'`, `'workarea executions - onepass'`, and `'workarea executions - multipass'`.

According to our estimates, the data we want from the build table is so small that it will easily fit into the `hash_area_size`. This is effectively the definition of an optimal hash join.

As a simple overview of the optimal hash join, Figure 12-1 indicates how the join takes place.

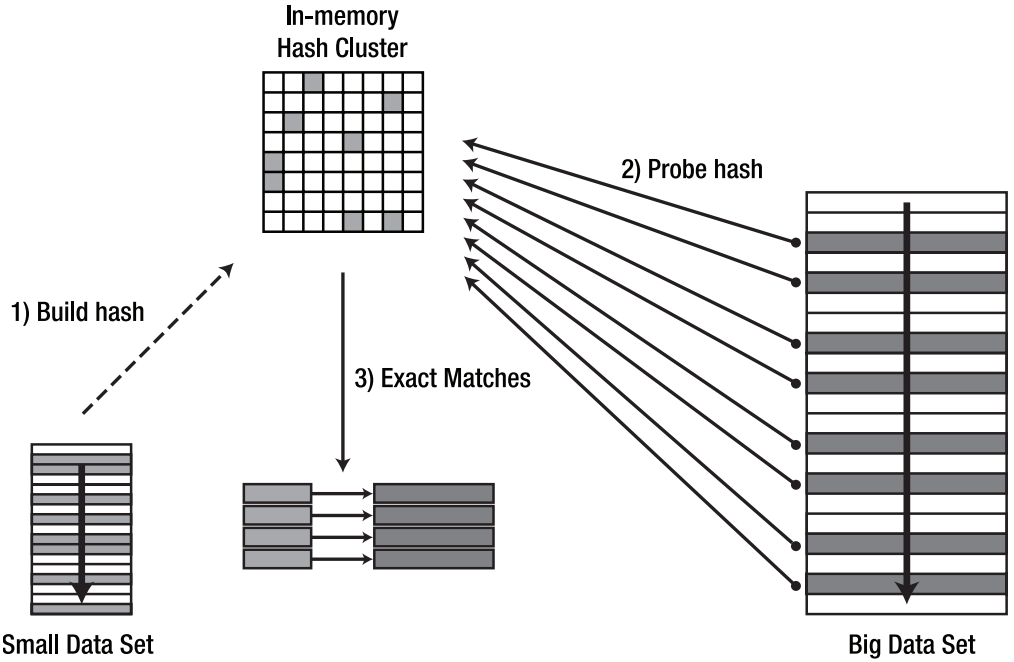

Figure 12-1. *Optimal hash join*

The steps are as follows:

1. Oracle reads the first data set and builds an array of "hash buckets" in memory. A hash bucket is little more than a location that acts as the starting point for a linked list of rows from the build table. A row belongs to a hash bucket if the bucket number matches the result that Oracle gets by applying an internal hashing function to the join column(s) of the row. The number of buckets in the hash table always seems to be an even power of two (common values for small hash joins are 1,024 or 4,096 buckets). Although the entire structure is really a complex collection of fixed arrays and linked lists, it is convenient to think of the hash table as a square array of cells with the rows from the first (build) table scattered randomly around the square.

2. Oracle starts to read the second table, using whatever access mechanism is most appropriate for acquiring the rows, and uses the same hash function on the join column(s) to calculate the number of the relevant hash bucket. Oracle then checks to see if there are any rows in that bucket—this is known as *probing the hash table*.

3. If there are no rows in the relevant bucket, Oracle can immediately discard the row from the probe table. If there are some rows in the relevant bucket, Oracle does an exact check on the join column(s) to see if there is a proper match. You will recall from Chapter 9 that it is always possible for two different input values to a hashing function to collide on the same output value. Since it is possible for rows with different values in the join column(s) to be in the same hash bucket, we have to do the exact check. Any rows that survive the exact check can immediately be reported (or passed on to the next step in the execution plan).

There is an important difference between Oracle's approach to acquiring memory for hashing compared to its approach for sorting. As you might guess from the comments about hash collisions in step 3, the hash join works most efficiently (as far as CPU is concerned) if there is at most one row in any hash bucket, so Oracle demands a huge fraction of the `hash_area_size` as soon as the hash join starts so that it can create a large number of buckets, as this helps to minimize hash collisions.

THE BACKWARDS NESTED LOOP

If you are familiar with single-table hash clusters, you will realize that an optimal hash join is really just a back-to-front nested loop into a dynamically created single-table hash cluster. We build a single-table hash cluster in local memory from the rows selected from build table, and then, for each row selected from the probe table, check that hash cluster by hash key.

The main benefit of an optimal hash join is that the build table is transferred into local memory, rather than being a real single-table hash cluster in the buffer cache. This means that the latch, buffer, and read consistency costs that normally occur on a table access simply don't appear as we probe the hash table.

Given that we have managed to produce an optimal (i.e., in-memory) hash join in this case, it seems reasonable that the final cost of the query (106) that we see in the execution plan would be as follows:

- The cost of acquiring data from the build table (42), plus
- The cost of acquiring data from the probe table (60), plus
- A little extra cost to represent the CPU cost of performing the hashing and matching.

Nevertheless, an increment of four in the cost for such a small amount of CPU work might be considered a little suspect (especially since we have not enabled CPU costing anyway). And in fact this example is a little deceptive, because the `hash_area_size` (1MB) is very much larger than the size of the build table data set (15KB).

The Onepass Hash Join

Of course, things can be a little more complex, as we can see if we change our tables to increase the size of columns and select longer columns so that both data sets are too large to fit into the `hash_area_size` (see script `hash_one.sql` in the online code suite). The significant changes to the code and the resulting plan are shown here:

```
create table probe_tab as
select
        10000 + rownum                          id,
        trunc(dbms_random.value(0,5000))        n1,
        rpad(rownum,20)                         probe_vc,
        rpad('x',1000)                          probe_padding
from
        all_objects
```

```
where
        rownum <= 10000
;

create table build_tab as
select
        rownum                                          id,
        10001 + trunc(dbms_random.value(0,5000))        id_probe,
        rpad(rownum,20)                                 build_vc,
        rpad('x',1000)                                  build_padding
from
        all_objects
where
        rownum <= 10000
;

--    Collect statistics using dbms_stats here

select
        bu.build_vc,
        bu.build_padding,                   -- extra length in build row
        pb.probe_vc,
        pb.probe_padding
from
        build_tab        bu,
        probe_tab        pb
where
        bu.id between 1 and 2000            -- more rows in build set
and     pb.id = bu.id_probe
;

Execution Plan (9.2.0.6 Autotrace)
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1127 Card=2000 Bytes=4114000)
1  0  HASH JOIN (Cost=1127 Card=2000 Bytes=4114000)
2  1    TABLE ACCESS (FULL) OF 'BUILD_TAB' (Cost=255 Card=2000 Bytes=2060000)
3  1    TABLE ACCESS (FULL) OF 'PROBE_TAB' (Cost=255 Card=10000 Bytes=10270000)
```

We can do the same arithmetic with the values from `user_tab_columns` to discover that the `bytes` figures are indeed *number of rows* * sum of column lengths*—giving us 2MB of data from the smaller data set, and 10MB from the larger data set.

Because the "small" data set is now larger than the available memory, Oracle is clearly going to have to adopt a more complex strategy for handling the hash table—and this problem is echoed in the cost of the join (1,127), which is now far more than the cost of the two tablescans (255) that are needed to acquire the data in the first place.

The difference in cost is largely the effect of the optimizer's estimate of the increased I/O that will occur at run time. Because the hash table is too big to fit into memory, some of it has to be dumped to disk and subsequently reread. Moreover, a similar fraction of the probe table

will also have to be dumped to disk and reread. Figure 12-2 indicates how the mechanism might work if the required hash table was about four times the size of the available memory.

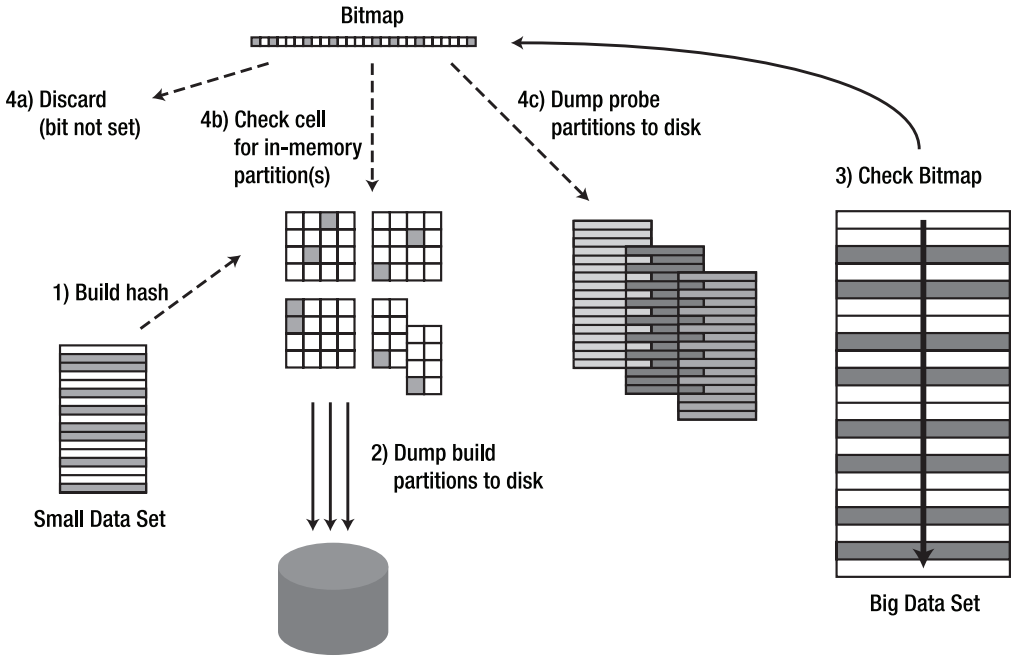

Figure 12-2. *Onepass hash join*

In this diagram, we see more of the components that are actually involved when Oracle performs a hash join. These are also present in the optimal hash join, but I chose to ignore them in my original diagram so that I could keep the picture as simple as possible.

The most important omission from Figure 12-1 was the (small amount of) memory reserved for a bitmap representing the hash table at the rate of 1 bit per bucket in the hash table. When a build row hashes to a particular bucket, the corresponding bit is set.

The rest of the `hash_area_size` is broken up into chunks (known as **clusters** or **slots**) of a size dictated by the parameter `_hash_multiblock_io_count` (obsolete as of 10*g*). This parameter always seemed to take the value 9 in earlier versions of Oracle, but is now much more dynamic, and the value is set individually for each query as it is optimized.

■ **Note** The hidden parameters `_smm_min_auto_io_size` and `_smm_max_auto_io_size` probably ought to affect the hash I/O size from 9*i* onwards, but don't seem to.

In the diagram, you can see that some of these memory clusters have been used to create the hash table, which has been sliced into four separate **partitions** (a word that is used to mean far too many different things in Oracle). The number of partitions always seems to be a power of two, and the number of partitions is chosen to leave a few memory clusters spare for juggling the anticipated I/O that will be needed to dump the hash table to disk. (The bottom right hash partition is split to show that it is made up of two chunks, one of which seems to be on its way down to disk.)

Bear in mind that the hash table is really just a collection of linked lists—the image of partitions as nice tidy squares with the same number of chunks is a visual convenience I am using to try to show the "logical identity" of hash buckets in the hash table. At any one moment, each partition could be made up of a different number of memory chunks.

You can get a lot of detail about how memory has been used by setting `event 10104`, the hash join trace, before executing a test query. The trace for the query from script `hash_one.sql`, for example, includes the following information that describes the components I have been describing:

```
Number of partitions: 8
Number of slots: 13                     -- 13 clusters at 9 blocks each
Multiblock IO: 9                        -- number of blocks to a cluster
Block size(KB): 8                       -- size of block
Cluster (slot) size(KB): 72             -- size of cluster in KB: 9 * 8KB = 72KB
Bit vector memory allocation(KB): 32    -- total memory for bitmap: 8 * 4KB = 32KB
Per partition bit vector length(KB): 4  -- memory per partition for bitmap
```

My `hash_area_size` was set to 1MB and, as you can see, Oracle decided to run with eight partitions for the hash table, and has enough usable memory to juggle with 13 clusters of 72KB—for a total of 936KB. The rest is taken up by 32KB for the bitmap (at 4KB per partition) and another 40KB, which is reported as a management overhead.

The order of activity is as follows:

1. The first data set is acquired and scattered into the hash table. As a bucket is used, the corresponding bit in the bitmap is set.

2. As the memory fills up, clusters are dumped to disk. The dumping is done using a cautious strategy that tries to keep as many complete partitions in memory for as long as possible. When the build table is exhausted, it is possible that some partitions will still be held completely in memory, while the rest have only a few (but at least one) clusters left in memory. It may be that one partition has just a few clusters, in memory and the rest have only one each. Whatever the outcome, Oracle will have a detailed map of where the data from each partition can be found. Moreover, whenever a hash bucket has been used (whether the relevant data items are in memory or on disk), the corresponding bit is set in the bitmap—which is always held completely in memory. At this point, Oracle tidies up the hash table, trying to get as many complete partitions into memory as possible and dumping any excess from other partitions to disk. As part of the rebuild, Oracle will reserve some clusters (a minimum of one per partition) for processing the probe table.

3. Once the hash table has been tidied up, Oracle starts to acquire rows from the second data set, applying the same hash function to the join column(s) of each row. The result of the hash function is used to check the relevant bit in the bitmap (a detail I chose to ignore in my description of the first example).

4. Oracle takes one of several possible actions, depending on the result of the test.

   4a) **Event**: The bit is clear (0).

   Action: There is no match, and the row is discarded.

   4b) **Event**: The bit is set (1), and the relevant hash bucket is in a partition that is in memory.

   Action: Check the hash bucket—if the probe row matches the build row, report it; otherwise discard it.

   4c) **Event**: The bit is set (1), but the relevant hash bucket is in a partition that is on disk.

   Action: Put the probe row to one side. It may match a build row that is on disk, but it would be too expensive to reread the relevant build partition at this point to check it.

The probe rows that are put to one side are collected in sets that match the partitions of the hash table previously dumped to disk. Just as we used some spare clusters for accumulating and dumping the hash table to disk, we use some spare memory clusters for accumulating the probe rows that are possible matches, and dump them to disk as each cluster becomes full.

As we get to the end of the probe table, we are left with matched pairs of partitions from the build table and probe table on disk.

At this point, Oracle has a complete map showing where all the data is and how many rows there are in each partition, so it picks a matched pair of dumped partitions (one build, one probe), and performs a hash join between them. As an extra optimization detail, Oracle can choose to swap the roles of the two partitions at this point because it knows exactly how much data there is in each partition, and there may be some benefit in using whichever is the smaller one to build the new in-memory hash.

So, in the case of a high-volume hash join, the hash table can spill to disk, with the probe table following it. The cost of the join ought to allow for the I/O performed in dumping the excess to disk and reloading it for the second phase of the join. This type of hash join is recorded as a `onepass` workarea execution because the probe dataset is reread from disk just once.

To generate a reasonable calculation of the cost of the join, we need to know the extra volume of I/O, the form it takes (i.e., typical I/O size), and how the optimizer decides to cost each I/O.

For example, in the worst possible case, Oracle might have to read the data sets from both tables, dump most of the data to disk, and then reread it. So the cost ought, perhaps, to go up by a factor of about three (corresponding to the fact that we have read, written, and reread virtually the entire volume of data).

Of course, we have to allow for the fact that when we are dumping and rereading the partitions, we may be able to use more efficient I/O than we did when we first collected the data. We also have to allow for the fact that we won't dump every row of our data, as some rows will be joined and reported on the initial acquisition from the probe table.

If we check the execution plan for the sample query, we can see the following data sizes:

```
Size (bytes) of build data:        2,060,000
Size (bytes) of probe data:        10,270,000
```

Since the hash_area_size in the test case is 1MB, and the volume of data from the build table is nearly 2MB, we will have to dump about half the build table (1MB) to disk and reread it later (another 1MB). Since we are going to dump about half the build table, it seems likely that we will dump about half of the probe table (5MB) and reread it (another 5MB).

Given these figures, we might hope to see an extra cost representing that 12MB (which equates to 1,536 blocks) of extra I/O. Of course, we saw from the 10104 trace file that the multiblock I/O size was 9 blocks—so Oracle should have been dumping and rereading 9 blocks at a time, for a total of about 170 I/O requests.

Unfortunately, the total cost of the join is 1,127. Since the two tablescans that went into the first pass of acquiring the data had a cost of 255 each, this means we have to explain an extra 1,127 – 2 * 255 = 617 units of cost. This seems a little high compared to our "guesstimate."

Clearly, the arithmetic done by the optimizer is not completely in sync with my description of how the hash join is working, and since my description is based to a fair degree on observing actual I/O patterns, you can infer that the cost calculation doesn't necessarily represent exactly what happens at run time.

Just to confuse the issue—if you run the query against 8*i* and 10*g*, you get costs of 1,079 and 1,081 respectively (the difference between these two results comes from the two table scans, not the join itself—remember that you add one to tablescan costs from 9*i* onwards).

What has actually happened in this case is that the run-time engine really has done the onepass join that we expected, but the optimizer has calculated that the join will require two passes. Moreover, the run-time engine in 9*i* has used a cluster size of nine even though the optimizer has based its calculations on a predicted cluster size of eight blocks (although in 8*i* and 10*g* the optimizer based its calculations on a predicted size of nine blocks, leading to a lower cost).

Then the cost model has assumed that the run-time engine will dump the whole probe table, and reread the whole probe table—because it hasn't worked out that we are doing a onepass join, which has some inherent I/O benefits.

A final complication appears when it comes to deriving a cost for the multiblock writes and reads used to dump and reload the partitions; the optimizer adopts the same strategy for adjusting multiblock read figures that we saw in Chapter 2, and we can use the results we saw for tablescans to calculate the costs of hash joins. Because the optimizer is predicting the cluster size as eight blocks in the 9*i* calculations, it is using the value 6.588 to do the arithmetic for the cost.

This issue highlights one of the difficulties of working out how the optimizer has derived the cost of a hash join. We don't know what cluster size the optimizer thinks it will be using—it's not one of the figures that's printed in the 10053 trace—so we don't know how many blocks the optimizer factors into a single I/O.

Moreover, we cannot guarantee that the code in the run-time engine follows the assumptions built into the optimizer—so even if we try to fine-tune memory to get the right cost, we can't guarantee that the run-time engine will behave for us anyway.

Untangling the oddities and conflicting bits of information, we can finally work out that the value of 617 can be derived as follows:

```
Size of large data set = 1,269 blocks        -- from column sizes and row counts
Size of small data set = 255 blocks          -- from column sizes and row counts
Cluster size (predicted) = 8                 -- inferred, after the event
I/O size used for calculation = 6.588        -- adjustment as described in Chapter 2
Probe passes = 2                             -- as reported in 10053 trace file
```

Then

```
cost =
        (probe passes + 1) * round(1,269/6.588) + round(255/6.588) =
        3 * 193 + 39 =
        579 + 39 =
        618                  --  as required (with a little rounding error)
```

You'll notice that the optimizer seems to have allowed for dumping and rereading the probe partitions—it used probe passes + 1 in the calculation—but only caters to the cost of the hash table once. I don't know why. (You'll also notice that I could have made a better choice about where to do my rounding to make the numbers come out exactly right—but there's always a little room for error in the rounding and print formatting routines that go into the `10053 trace`.)

Just to confirm that this hypothesis is correct, and not just a one-off lucky chance, we can run the calculations for 8*i* and 10*g*. With a `hash_multiblock_io_count` of 9, the optimizer will use the value 7.12 in the calculations. Bear in mind that we are looking for a total cost of 1,081 (in 10*g*)—which means that when we factor out the cost of the tablescans, the hash join component will be 1,081 – 510 = 570. (The result is the same for 8*i* once we allow for the slightly reduced tablescan cost: 570 = 1.079 – 508.)

Sure enough:

```
        (probe passes + 1) * round(1,269/7.12) + round(255 / 7.12) =
        3 * 178 + 36 =
        570                  --  as required
```

If you want to test the arithmetic further, you can rerun `hash_one.sql` with different values for the `hash_multiblock_io_count` and watch the hash join cost change. Note that the parameter became hidden in 9*i*, so the syntax for setting the parameter will change:

```
alter session set "_hash_multiblock_io_count" = 4;        -- 9i syntax
alter session set hash_multiblock_io_count = 4;           -- 8i syntax
```

Type very carefully when testing this feature with 9*i*; the 8*i* syntax does not produce an error in 9*i*, but the value is silently ignored.

The Multipass Hash Join

Let's look a little more closely at the most degenerate case where the available `hash_area_size` is much too small (or the statistics were sufficiently inaccurate that the optimizer made a very bad choice of the number of partitions needed for the hash table).

For the purposes of explanation, imagine you have 4MB of build data that needs to be scattered into a hash table, but the `hash_area_size` has been set so that there are only nine blocks (72KB) available for building the hash table.

Oracle will (probably) pick a cluster size of one, and build a hash table of four partitions, nominally two clusters per partition, with one spare block left for I/O. As each block in the hash table fills, it will be dumped to disk. At the end of the build phase, there will be four build partitions on disk, each about 1MB in size, and Oracle will have tidied up its memory so that the only data left in memory will be a few blocks from the first build partition.

At this point, the run-time engine knows that it is going to have to dump four partitions from the probe table to disk. So it has to reserve (at least) four blocks in memory, one for each probe partition, plus (at least) one spare block to read the probe table. So the in-memory hash table will be restricted to a maximum four blocks—to handle a partition of 1MB!

The probe pass begins, and rows are read from the probe table. As with the onepass hash join, Oracle can discard a row because the bitmap shows no matches, check the row against the in-memory data (either reporting or discarding it) when the bitmap shows a possible match and the relevant hash bucket is in memory, or put the row aside if the bitmap shows a possible match, but the relevant bucket is on disk.

When the first probe pass is complete, some rows will have been reported (the ones that found a match in the four blocks of the build table we had in memory), and there will be four probe partitions dumped to disk, one for each of the build partitions previously dumped to disk.

This is where things get nasty. Oracle is going to do a hash join between each pair of build and probe partitions. But each build partition is about 1MB, and there is only enough memory to reload a few blocks at a time. Since we have nine blocks to play with, and since we now know that we are joining partitions that are known to match, we have some room to maneuver—we could use one block to hash the build partition, and eight blocks to read the probe partition; or two blocks for the build partition and seven blocks for the probe partition; and so on. The more blocks we use for the hash, the smaller the read size on the probe partition; the fewer blocks we use for the hash, the larger the reads on the probe partition, but the more times we have to repeat the process. Assume we use four blocks to hash the build partition.

Oracle reads four blocks from the first build partition, and scans the entire first probe partition, joining where possible; then Oracle reads the next four blocks from the first build partition and scans the entire first probe partition, joining where possible. If it takes 32 passes to get through the whole of the first build partition, Oracle will have to scan the entire first probe partition 32 times.

And then Oracle goes on to the second pair of partitions and repeats the process. (This hash operation is known as a *multipass operation* because after the data has been dumped to disk, it is reread multiple times.)

If your `hash_area_size` is much too small, then the amount of I/O that takes place during a large hash join can be extreme. In my hypothetical case, we might expect the I/O cost to be about 34 times the cost of an optimal join, as we read the whole probe table, write (almost) all of the probe table to disk, and then reread every partition of the probe table 32 times.

MULTIPASS HASH JOINS

The current implementation of the hash join has a weakness that can appear only when you cannot perform a onepass join and it takes several passes to read each build partition. Every time you read a few blocks from the build partition, you scan the *entire* matching probe partition.

Ideally, you would hope that the run-time engine would detect this problem as it came to the end of the first pass. After all, it has collected exact details of the volume of data in each pair of partitions (one build, one probe), and knows how many "subpartitions" each partition would have to be split into to allow each pair of subpartitions to complete in a single pass.

In principle, the run-time engine could work out a mechanism of "recursive" hashing, applied to each partition in turn. The code would have to read a single build partition, hashing into a suitable number of subpartitions (such as the next power of two above the known number of passes required for the partition). The same split would have to be done to the probe partition—but this could be done while attempting to join to the first subpartition. With a strategy like this, the total I/O would be limited to a fixed amount.

Ignoring some of the boundary details: we read the build table, dump it to disk once in partitions, reread it and dump it to disk in subpartitions, and then reread each subpartition to do the join. We have also read the probe table, dumped it in partitions, reread it and dump it in subpartitions, and then reread each subpartition once. This looks like a better bet than having to reread the probe partitions multiple times because the build partitions are too big to be held in memory. (There is clearly a break point at about two where there is no point in trying to subpartition the hash table.)

Of course, on the downside, you have to worry about the extra complexity in the code, and the possibility that this "recursive hash join" would have to be applied to the subpartitions, and their subpartitions, and so on. And each time you descend into recursion, you have to take out more memory for mapping and other overheads—and since the multipass is only invoked when you are short of memory anyway, a recursive hash join may not really be a sensible thing to do.

If you see multipass hash joins occurring—and this is only likely to occur in a few special-case big jobs—you need to consider strategies for increasing the available memory, changing the join mechanism, or redesigning the SQL.

Although I haven't built a test case that matches the description exactly, script `hash_multi.sql` in the online code suite gets quite close. It repeats the table definitions of script `hash_one.sql`, but then forces a hash join to take place with a `hash_area_size` of 128KB—when the size of the build table is about 2MB.

This is what happens to the execution plan:

```
Execution Plan (9.2.0.6 Autotrace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=13531 Card=2000 Bytes=4114000)
1  0   HASH JOIN (Cost=13531 Card=2000 Bytes=4114000)
2  1     TABLE ACCESS (FULL) OF 'BUILD_TAB' (Cost=255 Card=2000 Bytes=2060000)
3  1     TABLE ACCESS (FULL) OF 'PROBE_TAB' (Cost=255 Card=10000 Bytes=10270000)
```

Notice how the cost of the tablescans has (of course) not changed, but the cost of the hash join has jumped from the onepass cost of 1,127 to a multipass cost of 13,531.

The optimizer can always work out (in principle) how many partitions would be needed to achieve a onepass hash join. But if the `hash_area_size` is too small to allow that many partitions to be created—remember that each partition needs at least one block of memory at all times—then the optimizer will recognize that a multipass join is required, and be able to work out how many passes will be required per probe partition. If you call the number of passes N, then the excess cost of the multipass join will be the cost of writing out the possible match rows once, and rereading them N times.

The detail from the `10104 trace` that shows us the size of our problem is the section labelled as `Phase 2`. Picking out the relevant bits from the trace file produced by the script `hash_multi.sql`, we find the following:

```
*** HASH JOIN GET FLUSHED PARTITIONS (PHASE 2) ***
Getting a pair of flushed partions.
 BUILD PARTION: nrows:256 size=(33 slots, 264K)
 PROBE PARTION: nrows:260 size=(34 slots, 272K)
*** HASH JOIN BUILD HASH TABLE (PHASE 2) ***
Number of blocks that may be used to build the hash hable 10
```

Critically we can see from the figures that we need 33 slots (which in this case also means 33 blocks as the starting section of the trace reported that the slot [cluster] size was 1 block) and we can compare this with the number of blocks (just 10) that may be used to build the hash table. (Yes, it did say "hash hable" and "PARTION" in the trace file—trace files are not for end users to see, so no one in Oracle development is going to rush to correct spelling mistakes. There are more important things to do.) Consequently we are going to have to split each build partition into four sections, reloading them one at a time, and scan each probe partition four times. This tells us that we need, at the very minimum, to increase our `hash_area_size` by a factor of about four—or at least 3.3—to avoid a multipass hash join.

This observation is confirmed all the way down the trace file as Oracle reports the amount of work it has done and the amount of work left to do. Extracting the critical details for the first pair of partitions (i.e., one build partition, one probe partition), we see groups of lines like the following:

```
Number of rows left to be iterated over (start of function): 256
Number of rows iterated over this function call: 78
Number of rows left to be iterated over (end of function): 178
...
Number of rows left to be iterated over (start of function): 178
Number of rows iterated over this function call: 78
Number of rows left to be iterated over (end of function): 100
...
```

```
Number of rows left to be iterated over (start of function): 100
Number of rows iterated over this function call: 78
Number of rows left to be iterated over (end of function): 22
...
Number of rows left to be iterated over (start of function): 22
Number of rows iterated over this function call: 22
Number of rows left to be iterated over (end of function): 0
```

Each one of these blocks of text tells us that we have read another little section of the first build partition, and scanned the entire corresponding probe partition.

As before, we can say that the cost of the hash join ought to represent the fact that we have done so much extra I/O, and the optimizer ought to be able to work out roughly how much I/O that is.

Again, it's a little difficult finding the right numbers when we work backwards from the activity.

In this case, the final cost was 13,531. The initial tablescan costs totaled 510. So the incremental cost of the join was 13,021. We know that we had to dump about 2MB for the build table (i.e., most of it), and 10MB of the probe table (again, most of it). We then had to reread the build table once (even though we read it in four separate pieces, we read each piece just once), but had to reread the probe table four times. So the total extra I/O was 2MB (written) + 2MB (read) + 10MB (written) + 4 * 10MB (read) = 54MB = 6,912 blocks.

Even though the "multiblock" direct read size in this case was just one block, the incremental cost has to be too high by a factor of about two—how can you manage to get a cost of 13,021 from a total of 6,912 I/Os?

In fact, as we saw with the onepass hash, the optimizer has got the numbers wrong. We saw four passes against each probe partition in the `10104 trace` file, but the calculations in the `10053 trace` file reported an estimated 16 passes. (In this case, we can assume quite happily that the actual cluster size of 1 was also the predicted cluster size.)

```
Size of large data set = 1,269 blocks      -- from column sizes and row counts
Size of small data set = 255 blocks        -- from column sizes and row counts
Cluster size (predicted) = 1               -- inferred, after the event
I/O size used for calculation = 1.676      -- adjustment as described in Chapter 2
Probe passes = 16                          -- as reported in 10053 trace files
```

Then:

```
cost =
        (probe passes + 1) * round(1,269/1.676) + round(255/1.676) =
        17 * 757 + 152 =
        12,869 + 152 =
        13,021                --  as required
```

Trace Files

Two trace files are critical to investigating the hash join. One is the standard CBO trace—`event 10053`; the other is the hash join trace—`event 10104`.

It can also be quite instructive to `enable 10046` at level 8 to watch for I/O wait states as the join progresses—and from 9*i* onwards, you can synchronize the `10046 trace` with the known I/O that the hash join performs by setting the 10104 trace to level 12. This may be particularly helpful, as the direct writes and reads produced by hash joins when they are dumping and rereading partitions can use a form of asynchronous I/O that results in very few I/O waits appearing in the `10046 trace` file.

Event 10104

The particularly nice thing about the `10104 trace` file is that the detail supplied is extremely informative, and can give you a very good idea of how big your `hash_area_size` should be to get from a multipass to a onepass join, or from a onepass to an optimal join.

We have already seen various bits of the `10104 trace`, but here are four of the lines that appear very early on in the 9*i* trace file (similar content with a slightly different presentation appears in the 8*i* and 10*g* files):

```
Original memory: 131072
Memory after all overhead: 129554
Memory for slots: 122880
...
Estimated build size (KB): 2050
```

The `Memory for slots` tells you how much memory is available at maximum for creating the in-memory hash table. The estimated build size tells you how much memory the optimizer thinks you need for slots. In this case, if you want to do an optimal hash, it looks like you are going to have to increase your `hash_area_size` from 128KB (of which about 120KB is available) to something a little over 2MB.

If we know we don't have enough memory to allow a `hash_area_size` large enough for the optimal join, then we may be able to allocate enough for the onepass join—and as we saw previously, we need only check the section of the trace file that describes how Oracle is reacquiring data from disk:

```
Getting a pair of flushed partions.
 BUILD PARTION: nrows:256 size=(33 slots, 264K)
 PROBE PARTION: nrows:260 size=(34 slots, 272K)
*** HASH JOIN BUILD HASH TABLE (PHASE 2) ***
Number of blocks that may be used to build the hash hable 10
Number of rows left to be iterated over (start of function): 256
Number of rows iterated over this function call: 78
Number of rows left to be iterated over (end of function): 178
```

The number of times we see (`end of function`) before the number of rows left gets down to zero is a good indicator of the scale factor we will have to use to multiple up the `hash_area_size` to get from the multiplass to the onepass join. (This sequence happens once per pair of dumped partition—so make sure you check for the worst case.)

You may find another important clue about the general performance of a hash join in a section of the trace file that gives a partition-level summary of how successfully the rows have been distributed:

```
### Partition Distribution ###
Partition:0    rows:247        clusters:32     slots:1      kept=0
Partition:1    rows:244        clusters:32     slots:1      kept=0
Partition:2    rows:208        clusters:27     slots:1      kept=0
Partition:3    rows:260        clusters:34     slots:1      kept=0
Partition:4    rows:260        clusters:34     slots:1      kept=0
Partition:5    rows:243        clusters:32     slots:1      kept=0
Partition:6    rows:282        clusters:37     slots:1      kept=0
Partition:7    rows:256        clusters:33     slots:7      kept=0
```

This is from the 9*i* trace file for `hash_multi.sql`. As you can see, the optimizer chose to split the hash table into eight partitions. This section of the trace file then shows how many rows were distributed to each partition and, as you can see, there is a little imbalance in the distribution—one partition has only 208 rows, another has 282 rows. This is actually quite reasonable, and nothing to be alarmed about.

However, if you can see massive imbalance between the partitions, this may be because your data has an odd distribution with a few highly repetitive values in the join columns. This may lead to excessive CPU consumption as the join takes place. For a multitable join, this may give you a clue that you should try to rewrite the query in some way to rearrange the join order.

An imbalance is unlikely to be the result of multiple hash collisions with Oracle's hashing function. However, I believe there may be a second hashing function that the code can use to rehash the data in memory if there are indications that the primary hashing function has produced excessive collisions between nonmatching rows.

The other columns in this part of the trace are as follows:

- **Clusters**: The number of clusters (slots) needed to hold all the rows in that partition. As I pointed out earlier, each cluster in this example is made up of exactly one block.
- **Slots**: The number of slots (clusters) from this partition that are currently in-memory. This part of the trace file is reported as the build completes, but before Oracle has tried to tidy up the hash table. As you can see, under pressure Oracle has been dumping partitions zero to six ferociously, although it has to keep at least one slot per partition in memory, and has only managed to keep seven slots of the 33 needed for partition seven.
- **Kept**: A flag set to zero or one to show whether or not the entire partition is still in memory. If all partitions are marked as kept, then this is an optimal hash join. This flag isn't included in the 8*i* trace—you have to infer it by checking whether the `clusters` value matches the `slots` value (which is labelled as `in-memory slots` in 8*i*).

This bit of the trace file gives another clue about how much you have to increase your `hash_area_size` to get from a multipass hash join to a onepass hash join. We can see that the best retention Oracle has managed is 7 slots compared to a worst case of 37 (the best partition is partition 7, which needs only 33 slots and got 7 slots, but the worst case partition is partition 6 which needs 37 slots). We probably need to multiply the `hash_area_size` by roughly 37/7 = 5.3 to get to a onepass hash.

Event 10053

The 10053 trace file is quite sparse in its information. Here, for example, is the 9*i* hash join section for the accepted join order in hash_opt.sql—the traces for 8*i* and 10*g* are very similar:

```
HA Join
  Outer table:
    resc: 42  cdn: 500  rcz: 30  deg: 1  resp: 42
  Inner table: PROBE_TAB
    resc: 60  cdn: 5000  rcz: 527  deg:  1  resp: 60
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 4   Deg: 1
      hash_area:  128 (max=128)  buildfrag:  129  probefrag:   329 ppasses:    2
  Hash join   Resc: 106   Resp: 106
```

Key points in this section of the trace are described in Table 12-2.

Table 12-2. *Explaining the Hash Join 10053 Entries*

Item	Description
Outer table	A set of figures about the data acquired so far in the join order. In our case, this is just the cost of getting 500 rows from build_tab.
Inner table	A set of figures about the data needed from the latest table (or instantiated view) in the join order. In our case, this is just the cost of getting 5,000 rows from probe_tab.
resc	Cost for serial execution of a step.
deg	Degree of parallelism of a step.
resp	Cost for full parallel execution of a step.
hash_area	Nominal value for hash_area_size in blocks. In 9*i* this is the minimum that will be available to the join, dictated by hidden parameter _smm_min_size (which is given in KB).
hash_area (max=)	Nominal maximum value for hash_area_size in blocks. In 9*i* this is the maximum that will be available to the join when running with the automatic workarea_size_policy. In principle this is dictated by the hidden parameter _smm_max_size (in KB). However, although the manuals state that the maximum value for a single workarea is 5% of the pga_aggregate_target, the value reported here will be 10% of the target (i.e., 2 * _smm_max_size). This seems to be copying the original behavior of hash_area_size, which would default to twice the sort_area_size. One part of the optimizer calculation seems to follow this limit, but includes a boundary condition that switches the calculation to use the minimum size. At run time, the 5% limit is obeyed—but there are indications that it is a slightly flexible limit. To further confuse the issue, there is another hidden parameter, _pga_max_size, that defaults to 200MB, and this is the accounting limit for the total PGA of a single process. The _smm_max_size is not supposed to exceed half the _pga_max_size, but you won't notice this unless your pga_aggregate_target exceeds 2GB—at which point _smm_max_size hits 100MB and stops growing unless you start tweaking the _pga_max_size or _smm_max_size directly.

Table 12-2. *Explaining the Hash Join 10053 Entries*

Item	Description
buildfrag	The size of the build (first) data set in Oracle blocks. This is clearly wrong for our optimal hash join in 8*i* (and 9*i* and 10*g* when they emulate 8*i*)—it seems to be reported as hash_area + 1 for optimal hash joins. In our example, the correct value is 3.
probefrag	The size of the probe data set in Oracle blocks.
ppasses	The number of probe passes needed to complete the join. Clearly incorrect in our optimal example, as we know we never dump to disk and reread. But ppasses is never reported as zero. This value seems to be for information only, as it has already been catered to in the value for the hash join one ptn (see the next entry)—and in most of my test cases, the value did not agree with what actually happened at run time anyway. Although in the case where we use the hash_area_size and do not enable CPU costing (i.e., the traditional costing approach), the value for ppasses seems to be derived from buildfrag / hash_area; in all other circumstances it never moves from the value 1.
Hash join one ptn	Probably the most important number. This is notionally the unit cost of dealing with one build partition. Despite the cost being labeled resc: (which is normally interpreted as "serial cost"), the value is also used in parallel queries, where the optimizer seems to multiply resc: by deg before adding it into the total cost of the join. For multipass joins, this number has a component that has been premultiplied by the value of ppasses before being printed here.
Hash join	For all (serial) joins, the total cost of the join seems to be Hash join one ptn plus the costs of acquiring data from the inner and out table costs. In the example, we have 106 = 4 + 60 + 42.

Headaches

It is quite easy to find anomalies in the hash join calculations, and before making any comments about the differences between the various costing mechanisms, I'd like to show you a couple of these anomalies so that you can appreciate the difficulty of micro-tuning the hash_area_size (or pga_aggregate_target, for that matter).

Traditional Costing

Go back to hash_one.sql, and run the baseline query with a manual workarea_size_policy and hash_area_size set to 1,100KB; then repeat the exercise with the hash_area_size set to 2,200KB (see script hash_one_bad.sql in the online code suite):

```
Execution Plan (9.2.0.6 autotrace - Hash area size = 1,100 KB)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1081 Card=2000 Bytes=4114000)
1    0   HASH JOIN (Cost=1081 Card=2000 Bytes=4114000)
2    1     TABLE ACCESS (FULL) OF 'BUILD_TAB' (Cost=255 Card=2000 Bytes=2060000)
3    1     TABLE ACCESS (FULL) OF 'PROBE_TAB' (Cost=255 Card=10000 Bytes=10270000)
```

```
Execution Plan (9.2.0.6 autotrace - Hash area size = 2,200 KB)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2769 Card=2000 Bytes=4114000)
1    0   HASH JOIN (Cost=2769 Card=2000 Bytes=4114000)
2    1     TABLE ACCESS (FULL) OF 'BUILD_TAB' (Cost=255 Card=2000 Bytes=2060000)
3    1     TABLE ACCESS (FULL) OF 'PROBE_TAB' (Cost=255 Card=10000 Bytes=10270000)
```

We've increased the memory available for the hash join, and the cost of doing the hash join has gone up by a factor of 2.5. This isn't what you would expect, but it's something that happens when your memory allocation is hovering on the boundary between the optimal and onepass hash joins.

The most dramatic thing about this example was that the run-time trace also showed Oracle switching, for no obvious reason, from multiblock I/O (cluster size is nine blocks) to single block I/O (cluster size is one block) when the available memory was larger. Normally, when the optimizer arithmetic does something strange, you find that the run-time mechanism does something completely different anyway. In this case, the run-time engine and the optimizer showed the same strange behavior.

Modern Costing

With the move to CPU costing and the use of dynamic workarea sizing in 9*i*, you may think that such anomalies are a thing of the past. Script `hash_pat_bad.sql` in the online code suite shows otherwise. It uses the same query as we had in `hash_one.sql`, enables system statistics, and uses two different values of `pga_aggregate_target`. Again, it is possible to produce counterintuitive results:

```
Execution Plan (9.2.0.6 autotrace - pga_aggregate_target = 20,000 KB)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=719 Card=2000 Bytes=4114000)
1    0   HASH JOIN (Cost=951 Card=2000 Bytes=4114000)
2    1     TABLE ACCESS (FULL) OF 'BUILD_TAB' (Cost=257 Card=2000 Bytes=2060000)
3    1     TABLE ACCESS (FULL) OF 'PROBE_TAB' (Cost=257 Card=10000 Bytes=10270000)

Execution Plan (9.2.0.6 autotrace - pga_aggregate_target = 22,000 KB)
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=951 Card=2000 Bytes=4114000)
1    0   HASH JOIN (Cost=719 Card=2000 Bytes=4114000)
2    1     TABLE ACCESS (FULL) OF 'BUILD_TAB' (Cost=257 Card=2000 Bytes=2060000)
3    1     TABLE ACCESS (FULL) OF 'PROBE_TAB' (Cost=257 Card=10000 Bytes=10270000)
```

The setting I have used for the `pga_aggregate_target` in this example is rather small—even the default is larger—but the test data set was also quite small. The anomaly can be re-created with larger values for the parameter by using larger data sets.

The extra oddity here is that the only differences you can find in the trace files are in the `10053 trace`, where the cost of doing the partition join changes. The `10104 trace` files are identical. This is another indication that the optimizer and the run-time engine don't always use the same model.

Comparisons

Oddities exist in the costing—particularly in the boundary areas where the memory is in the right ballpark for the switch between one level of join and the next. There are contradictions between the model used by the optimizer code and the activity carried out by the run-time engine. However, it is possible to get a rough idea of what's going on, and make some guesses about how data sizes and parameter settings make a difference. And with the aid of the `10053` and `10104 traces`, it is possible to work out whether you can do some tweaking to make an important query operate more efficiently.

At present, though, I don't think that this micromanagement viewpoint is terribly important. The real issue that most people have to face is the impact of migrating between versions of Oracle, and deciding which features to enable. This is where problems are likely to occur. So in this section, we will run a standardized test so that we can compare the behavior of the optimizer in four different working environments.

The online code suite contains four scripts to help us compare the different behaviors. Between them, they cover four separate combinations of options:

CPU costing disabled, manual `hash_area_size` allocation	`has_nocpu_harness.sql`
CPU costing enabled, manual `hash_area_size` allocation	`has_cpu_harness.sql`
CPU costing disabled, automatic `hash_area_size` allocation	`pat_nocpu_harness.sql`
CPU costing enabled, automatic `hash_area_size` allocation	`pat_cpu_harness.sql`

Of these, the first is the traditional 8*i* approach, the last is where I think your strategic choice ought to be with 9*i*. Unfortunately, the results you get from hash join calculations vary dramatically with your choice.

HASH_AREA_SIZE AND SHARED SERVERS

If you are mixing automatic workarea sizing with shared servers (once known as multithreaded servers—MTS), then many of the details about the `pga_aggregate_target` do not apply until you get to 10*g*. Historically, the memory for hash joins had to be in the UGA, which was kept in the SGA when you were running shared server. In 9*i*, this is still true, and the `hash_area_size` still dictates the memory allocated by shared servers, even though the allocation is then monitored by the newer workarea code. Strangely, the optimizer does not know whether a piece of SQL is coming from a shared server or a dedicated server, so the calculations are always based on the `pga_aggregate_target` approach—even when the available memory is limited by the `hash_area_size`.

In 10*g*, the shared server hash area has managed to migrate out of the UGA into the PGA, and so its size is dictated by the `pga_aggregate_target`. Note: in earlier versions of Oracle, you were advised to avoid using shared servers for sessions doing long-running hash or sort operations—perhaps this warning is no long relevant in 10*g*.

The four scripts basically do the same thing. Where I have enabled CPU costing, I have used the following set of figures to emulate the arithmetic that the traditional I/O costing would do:

```
begin
        dbms_stats.set_system_stats('MBRC',6.59);
        dbms_stats.set_system_stats('MREADTIM',10.001);
        dbms_stats.set_system_stats('SREADTIM',10.000);
        dbms_stats.set_system_stats('CPUSPEED',1000);
end;
/
```

I have set the MBRC to 6.59 because that corresponds to the adjusted value that the optimizer uses to cost tablescans (and index fast full scans) when the db_file_multiblock_read_count is set to my usual test value of 8. I have then set the mreadtim and sreadtim to be (virtually) identical—which is one of the basic assumptions of the older I/O costing algorithms used by the optimizer.

The scripts then generate an SQL script that calls a third SQL script (has_dump.sql or pat_dump.sql depending on the driving script) many times.

■ **Caution** The mechanism of scripts writing scripts then calling them is one that I consider to be extremely dangerous on production systems—but it is occasionally convenient when testing.

The scripts that use manual hash_area_size allocation contain a line like this one:

```
alter session set hash_area_size = &2;
```

The scripts that use automatic hash_area_size allocation—i.e., the ones that have workarea_size_policy set to automatic and rely on the pga_aggregate_target to control the hash_area_size—contain a line like the following:

```
alter system set pga_aggregate_target = &2 scope = memory;
```

All the called scripts enable the 10053 trace and the 10104 trace, and use the tracefile_identifier to make sure that each test generates its own suitably named trace file—for example:

```
alter session set events '10053 trace name context forever, level 2';
alter session set events '10104 trace name context forever';
alter session set tracefile_identifier = 'pat_&1._&2';
```

■ **Note** For a less resource intensive, though less informative, approach, I have also written four scripts that simply cycle through values of hash_area_size—or pga_aggregate_target—generating execution plans for a fixed statement; these scripts are in the online code suite as hash_stream_(a,b,c,d).sql.

I had a bit of a problem deciding how to line up the figures from the hash_area_size tests with those from the pga_aggregate_target tests.

According to the manuals, any individual workarea is limited to 5% of the pga_aggregate_target, so obviously a pga_aggregate_target of 100MB should be compared with a hash_area_size of 5MB.

On the other hand, when I set the pga_aggregate_target to 100MB, the 10053 trace reported the hash area (max=) at 10MB, not 5MB. So in the end I decided to line the two sets of figures up on the formula hash_area_size = 0.1 * pga_aggregate_target on the basis that this was probably the value that the optimizer was using for most of its calculations.

On top of this, there was another little problem: the minimum legal value for pga_aggregate_target is 10MB—which corresponds to 1MB for the hash_area_size calculations by my formula; however, the legal minimum for the hash_area_size is 32KB (although this is ignored and the effective minimum seems to be 64KB). So my sets of figures don't start at the same place.

Finally, rather than reporting the actual cost of the query, I have subtracted the cost of the two tablescans from the total cost, to give you the cost of the join itself.

Table 12-3 lists the costs of the baseline query for different feature sets at equivalent levels of memory for the hash table—and yes, they are the right headings; the figures with CPU costing enabled really do come out lower than the figures without.

Table 12-3. *Costs Changing with Memory Allocation and Feature Usage*

Target Size for Hash Memory	Cost : Manual No CPU Costing	Cost : Manual CPU Costing	Cost : Automatic No CPU Costing	Cost : Automatic CPU Costing
128KB	13,021	658		
256KB	6,965	408		
384KB	3,455	1,242 ****		
512KB	1,911	765		
768KB	1,103	384		
960KB	884	234		
1,024KB	617	194	506	441
1,280KB	533	117	506	441
1,408KB	501	90	506	441
1,536KB	448	62	506	441
1,792KB	408	28	306	209
2,048KB	2,416 ****	2	506 ****	441 ****
2,560KB	789	2	506	441
3,072KB	392	2	506	441
3,584KB	255	2	306	209
4,096KB	186	2	306	209

Table 12-3. *Costs Changing with Memory Allocation and Feature Usage (Continued)*

Target Size for Hash Memory	Cost : Manual No CPU Costing	Cost : Manual CPU Costing	Cost : Automatic No CPU Costing	Cost : Automatic CPU Costing
4,608KB			306	209
5,120KB			306	209
5,632KB			190	104
6,144KB			190	104

The first thing you notice is that the only two columns that bear any resemblance to each other are the ones driven by the `pga_aggregate_target`. And even then it's the trends that match, not the actual values. Whatever change you make to your systems, whether it's an upgrade or enabling a feature, you are likely to run into some problems with dramatic swings in hash join costs.

The second thing you notice is that, whichever option you choose for your database, there are bugs in the code, or possibly flaws in the model. In every single column, you can find a case (marked with the ****) where increasing the available memory for the hash join results in the join itself being given a higher cost.

This seems to occur close to the boundary where the `hash_area_size` (or `hash area (max=)` value) is about the same as the first data set—and the knock-on effect is that optimal hash joins are probably given a cost that is too high. The cost of the join tapers off as the `hash_area_size` grows, but only becomes negligible when the `hash_area_size` is very large.

PROBLEMS WITH PARTITION SIZING

We have already seen that the numbers used by the optimizer to do calculations (i.e., `ppasses` when it is populated) and the numbers used by the run-time engine do not have to be the same. The strange large increases in the costs shown in the table may be the result of the optimizer switching to a larger partition size in its model as the memory grows—but finding that this results in more probe passes.

I have some cases where this surprising error also appears in the run-time trace. For example, with a `hash_area_size` of 256KB, I have a join that runs with 16 partitions, and completes as a onepass join. When I increase the `hash_area_size` to 320KB, the partition count drops to 8 (bigger) partitions—which makes for more efficient I/Os—but the join switches to a multipass join.

If you enable CPU costing, but don't enable the features of automatic workarea policy, then the costing for onepass and optimal joins seems to be quite reasonable—even to the extent that pure in-memory joins become effectively free of charge at just about the right point. If your most important queries stick within this scale of operation, then this seems to be the best possible option for your system (or at least the sessions that you expect to be doing hash joins). However, there is clearly a problem when the first data set is large compared to the available `hash_area_size`—it looks as if the arithmetic to cater to multiple probe passes has fallen out of the code path—and the cost of multipass joins seems to be much smaller than you might expect. It is possible that this will result in hash joins being selected when there are better

options. Watch out for extremely inefficient hash joins if you have SQL where both data sets in the join are very large.

When you switch to using the automatic workarea policy, the results look a little surprising. On the downside, the code is clearly putting an unrealistically high cost on optimal hash joins. In fact, the version using CPU costing did eventually drop the cost of the hash join to 0, but only after I increased the `pga_aggregate_target` to nearly 1000MB, and I still haven't been able to work out why.

There is, however, an important difference between the models used for manual and automatic policy that makes the newer model a much more appropriate one—although it may still need some adjustment. Notice how the cost for the older model changes gradually (in most cases) as the available memory changes, while the cost for the newer model stays constant for a while, and then jumps.

This is quite reasonable—there are actually two good reasons why there should be sharp steps in the cost of a hash join. Recall that the hash table is split into partitions, and partitions are made up of clusters. The cluster is the unit size for I/O. The most important decisions that the optimizer has to make about a hash join are how many partitions, and how big should the clusters be.

Assume you have enough memory for N clusters—if you increase the memory by N blocks, every cluster can become one block bigger, which means that the (multiblock) I/O operations for dumping and rereading partitions will be a little more efficient. So that's one reason for a step.

The second reason for the step could be justified by thinking about the partition dumping and reloading that takes place. Assume at run time that it is possible to hold two build partitions in memory as the build phase ends. In this case, only six out of eight probe partitions will need to be dumped to disk and reloaded. Make the memory about 25% larger and perhaps three partitions of the eight build partitions will be still be in memory at the end of the build—so only five probe partitions will need to be dumped and reloaded.

Of course, as the memory size increases, the optimizer may decide to change the size of clusters, or change the number of partitions in a balancing act designed to trade the size of I/Os against the possibility of doing an optimal join, but in principle, these two considerations ought to show up in the cost calculations.

In fact, if you look closely at the detailed figures for the *traditional* costing, and then cross-reference with the `10104 trace` files, you can see the effect of the cluster size changing. The following is an extract from the `hash_stream_a.sql` set of figures:

```
Hash area KB Total Cost  Hash Cost
------------ ---------- ----------
         512       1277        765
         520       1273        761
         528       1269        757
         536       1265        753
         544       1261        749
         552       1072        560        delta = -189 (see below)
         560       1069        557
         568       1066        554
         576       1063        551
         584       1060        548
         592       1057        545
         600       1054        542
```

```
608        1051        539
616        1048        536
624        1045        533
632        1042        530
640        1039        527
648        1036        524
656        1033        521
664         927        415        delta = -106 (see below)
672         925        413
680         923        411
688         920        408
696         918        406
704         915        403
```

As the `hash_area_size` increases, the cost of the hash drops steadily by a few units, but every now and again it experiences a sharp drop (as highlighted by the *delta* lines in the preceding listing). These step changes result from the optimizer allowing for a larger cluster size as the `hash_area_size` increases. If you compare the `10104 trace` with the `10053 trace`, you will usually find that each step change does actually correspond (closely, but not perfectly—the run-time engine doesn't follow the exact model used in the calculations) to the cluster size increasing by a single block.

So the old model does allow for different sized I/Os—and a similar effect is visible in the numbers when you enable CPU costing, even if you stick with the manual `hash_area_size`. Since that's the case, you might start to wonder how things change when you start to adjust the system statistics—after all, the most significant effect of system statistics is to tell the optimizer the size and relative time for multiblock reads. So which system statistics affect the cost of the join?

Unsurprisingly, if you change the CPU speed, then the cost of the join changes—though, as you might expect, it won't normally change very much.

The cost also changes if you change the relative values of `mreadtim` and `sreadtim`. Hash joins do a lot of multiblock reads and writes (albeit direct path ones), so the I/O response time should make a difference to the cost, and there are some indications that the optimizer simply uses the `mreadtim` as the time for each of its cluster-sized I/Os, irrespective of the actual current size of the cluster.

The value of the `MBRC` statistic also has an impact—but apparently only some of the time. I haven't been able to work this out yet, but I believe it is introducing yet another complication into the decision about number of partitions and the cluster size, and since you can only see the run-time values for this settings, not the optimizer predictions, it is very hard to work out why the costs have changed with the value of `MBRC` when they do change, and why they haven't when they don't.

So CPU costing has some relevance, and there is good sense in the stepped values we see when using automatic workarea sizes. Nevertheless, the long intervals of constant cost you see with the automatic workarea sizing are still a bit of a puzzle. The answer only becomes clear when you examine the `10104 trace`, and realize that the optimizer is making memory-based decisions in a radically different way.

As we saw earlier on, the first values reported in the `10104 trace` as follows:

```
Original memory: 581632                          -- 568 KB
Memory after all overhead: 710649                -- 694 KB
Memory for slots: 688128                         -- 672 KB
```

I've picked this set of three values from a trace file that had the `pga_aggregate_target` set to 11,440KB to highlight a couple of points. First, that the hash join did, indeed, start with 5% of the `pga_aggregate_target`, but almost immediately decided that more was necessary. So it seems the 5% is not an absolutely hard limit for hash joins.

The second point emerges when I extract the `Memory for slots` from several other trace files with `pga_aggregate_target` varying from 10MB to 40MB—every single one up 33,200KB (32.4MB) has the same 672KB, after which Oracle jumps to using 1,440KB for its slots. This strategy is a reflection of the costing we saw with automatic workarea sizing enabled—the optimizer gets the same cost time after time because it is planning to use the same amount of memory time after time, irrespective of what you tell it is available.

When you switch to the modern technology, it really doesn't matter what you have set the `pga_aggregate_target` to, the optimizer is going to work out a sensible amount of memory for doing the join in an efficient fashion, and is only going to vary the memory demands in large steps, because basically it's only large changes that make a worthwhile difference. In this example, the big switch occurred because Oracle moved from using 7 blocks per partition to 15 blocks per partition—it's the sort of change that might make the extra use of memory give a reasonable return in terms of I/O performance.

Multitable Joins

If the description of the change in strategy isn't enough to convince you that using the automatic workarea sizing is a good idea, then let's consider some more realistic examples of SQL.

Very few sites are content to stop at queries against just two tables, and when you start executing multitable hash joins, your memory demand can go through the roof. Consider this simple query (extracted from script `treble_hash_auto.sql` in the online code suite):

```
select
        /*+
                ordered
                full(t1) full(t2) full(t3) full(t4)
                use_hash(t2) use_hash(t3) use_hash(t4)
                swap_join_inputs(t2) swap_join_inputs(t3)
        */
        count(t1.small_vc),
        count(t2.small_vc),
        count(t3.small_vc),
        count(t4.small_vc)
from
        t1,
        t4,
        t2,
        t3
```

```
where
        t4.id1 = t1.id
and     t4.id2 = t2.id
and     t4.id3 = t3.id
;

Execution Plan (9.2.0.6 Autotrace).
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=927 Card=1 Bytes=38)
1    0   SORT (AGGREGATE)
2    1     HASH JOIN (Cost=927 Card=343000 Bytes=13034000)
3    2       TABLE ACCESS (FULL) OF 'T3' (Cost=2 Card=70 Bytes=420)
4    2       HASH JOIN (Cost=923 Card=343000 Bytes=10976000)
5    4         TABLE ACCESS (FULL) OF 'T2' (Cost=2 Card=70 Bytes=420)
6    4         HASH JOIN (Cost=919 Card=343000 Bytes=8918000)
7    6           TABLE ACCESS (FULL) OF 'T1' (Cost=2 Card=70 Bytes=420)
8    6           TABLE ACCESS (FULL) OF 'T4' (Cost=915 Card=343000 Bytes=6860000)
```

The hints aren't really necessary, but I've put in the complete set to demonstrate the `swap_join_inputs()` hint that you can use with hash joins. The hints also highlight the fact that sometimes the order that tables appear in the execution plan is a little counterintuitive.

In this example, Oracle will hash table `t3` (line 3) into memory, then hash table `t2` (line 5) into memory, then hash table `t1` (line 7) into memory. Then the scan of table `t4` will begin.

Oracle will pick a row from `t4` and probe `t1`; if the row survives, Oracle will probe `t2`; if the row survives, Oracle will probe `t3`; if the row survives, then it will be passed to the sort (aggregate) operation. A multi-table hash join can produce results extremely quickly—it is not (always) necessary for the first hash join to complete before the second one can start.

Despite my description of Oracle hashing `t3`, then `t2`, then `t1`, and then probing with `t4`, Oracle has obeyed my `ordered` hint. Look carefully at the order of the tables in the `from` clause (`t1`, `t4`, `t2`, `t3`)—Oracle really has obeyed that hint. It goes like this:

- Join order [1]: `t1`, `t4`, `t2`, `t3`
- Now joining `t4`: (`t1`, `t4`)
- Now joining `t2`: ((`t1`, `t4`) , `t2`)
- But we use `swap_join-inputs(t2)`, which means when we see `t2` as the second table of a hash join, we reverse the join order: (`t2`, (`t1`, `t4`))
- Now joining `t3`: ((`t2`, (`t1`, `t4`)), `t3`)
- But we `swap_join_inputs(t3)`, which means when we see `t3` as the second table of a hash join, we reverse the join order: (`t3`, (`t2`, (`t1`, `t4`))), as we got in the execution plan.

The purpose of this section, though, was to describe the benefits of using the automatic `workarea_size_policy`. So let's run the query twice: once with a `hash_area_size` of 10MB, and once with a `pga_aggregate_target` of 200MB (which is the equivalent of 10MB `hash_area_size` according to the manuals, though the optimizer may use 20MB as the baseline figure in the calculations according to my observations).

Before we run the test, remember that this query needs three work areas—one for each hash join that takes place—and they will all be allocated simultaneously. If you had an 11-table join, you would need ten workareas if all the joins were hash joins (and one or two more if you did some grouping and ordering following the join).

If I start a new session and check the session statistics after executing the query, these are the results I get from my 9.2.0.6 instance with the manually controlled `hash_area_size`:

```
Name                                     Value
----                                     -----
session logical reads                    6,037
session uga memory max                12,948,124
session pga memory max                14,791,008
consistent gets                          6,037
physical reads                           6,015
```

whereas this is what I get with the automatic `workarea_size_policy`:

```
Name                                     Value
----                                     -----
session logical reads                    6,037
session uga memory max                 3,058,432
session pga memory max                 3,256,672
consistent gets                          6,037
physical reads                           6,018
```

The figures about logical and physical I/Os are there just to indicate that both queries did the same thing. The critical numbers are the `memory max` figures. Notice how much more memory the query used when we set a manual `hash_area_size`. Considering the three tables that were hashed contained only 70 short rows each, that's a lot of memory for hashing. (10*g* did even better on the automatic sizing, restricting itself to no more than 1.5MB for the four-way join.)

When running with the manual `hash_area_size`, there are indications that Oracle immediately allocates at least half the `hash_area_size` at the start of a single hash join—and never releases any of that memory even though each table that is hashed turns out to need only a small amount of memory.

HOW GREEDY ARE HASH JOINS?

Various details of the hash startup mechanism may have been tweaked across versions, but the amount of memory demanded for a multitable hash join can get very large as your `hash_area_size` grows—in early versions of 8*i* there may even have been cases where 100% allocation took place on every join.

When running with a large `pga_aggregate_target`, it seems that Oracle can start with a fairly large amount of memory allocated for hash table slots, and then release it when it is seen to be unnecessary. As the hash join runs, you may see references to resizing operations scattered through the `10104 trace` file—resizing downwards in the following example:

```
Slot table resized: old=23 wanted=12 got=12 unload=0
```

In other cases, of course, you will see Oracle starting small and growing the memory—which is why in the previous section we saw the `10104 trace` file starting with 672KB of memory when there was a notional limit of at least 1.5MB or even 3MB—if Oracle had needed more to avoid an expensive join, it would have resized the hash table upwards as it came to the end of the first build phase.

Summary

The algorithms for working out the cost of a hash join are highly dependent on whether you enable CPU costing and/or the automatic workarea policy. This means that an upgrade from 8*i* to 9*i*, or any change in the features enabled, may cause dramatic changes in costs, which can result in dramatic changes in execution plans.

Whatever set of features you have enabled for costing, there are "catastrophe" points in the calculations, i.e., points where a change in the `hash_area_size` (or `pga_aggregate_target`) that should apparently produce a decrease in cost may cause an increase in cost—the point at which this occurs is dependent on which features are enabled. Effectively, this means you cannot safely fine-tune hash joins by tweaking either of these parameters and expect the end result to be stable—you can always be unlucky.

The most nicely behaved option at present seems to be manual `hash_area_sizes` with CPU costing enabled. This fails to produce an appropriate cost for multipass joins, but does produce good costs for onepass and optimal joins.

Despite the apparent niceness of the manual `hash_area_size` with CPU costing, the strategic option is to set the `workarea_size_policy` to `auto`, and use the `pga_aggregate_target` with CPU costing. For problem queries, you may want to override the memory allocation at the session level, and fix some access paths by very deliberate use of hints at the statement level.

Bear in mind that `workarea_size_policy` budgets for multitable joins much more wisely than the manual policy. This, in itself, is a very good reason for switching to the new technology.

Test Cases

The files in the downloads for this chapter are shown in Table 12-4.

Table 12-4. *Chapter 12 Test Cases*

Script	Comments
`Hash_opt.sql`	Simple example of in-memory (optimal) hash join
`Hash_one.sql`	Onepass hash join
`Hash_multi.sql`	Multipass hash join
`Hash_one_bad.sql`	An anomaly in costing a onepass join using traditional methods
`Hash_pat_bad.sql`	An anomaly in costing a onepass join using the latest features
`has_nocpu_harness.sql`	Driver for generating traces with manual `hash_area_size`, without CPU costing

Table 12-4. *Chapter 12 Test Cases*

Script	Comments
has_cpu_harness.sql	Driver for generating traces with manual hash_area_size, with CPU costing
pat_nocpu_harness.sql	Driver for generating traces with automatic hash_area_size, without CPU costing
pat_cpu_harness.sql	Driver for generating traces with manual hash_area_size, with CPU costing
has_dump.sql	Unit script used by has_nocpu_harness.sql and has_cpu_harness.sql
pat_dump.sql	Unit script used by pat_nocpu_harness.sql and pat_cpu_harness.sql
hash_stream_a.sql	Hash join costs with manual hash_area_size and no CPU costing
hash_stream_b.sql	Hash join costs with manual hash_area_size and CPU costing enabled
hash_stream_c.sql	Hash join costs with automatic hash_area_size and no CPU costing
hash_stream_d.sql	Hash join costs with automatic hash_area_size and CPU costing enabled
treble_hash.sql	Generates tables for three-way hash join
treble_hash_auto.sql	Executes three-way hash join using workarea_size_policy = auto
treble_hash_manual.sql	Executes three-way hash join using workarea_size_policy = manual
snap_myst.sql	Reports changes in current session's statistics
c_mystats.sql	Creates view used by script snap_myst.sql—has to be run as SYS
setenv.sql	Sets a standardized environment for SQL*Plus

CHAPTER 13

Sorting and Merge Joins

A merge join requires both its inputs to appear in sorted order, and quite often it is the sorting that uses more resources than the rest of the operation. Consequently, the first section of this chapter is all about sorting, with the mechanisms and costs of the merge operation appearing only in the latter part of the chapter.

Sorting is a rather special operation that is worth focusing on because it is such a common operation, and yet the actual mechanisms and resource requirements are still not commonly known. Oracle may need to use sorting for `order by` clauses, `group by` clauses, the `distinct` operator, merge joins, `connect by` queries, B-tree to bitmap conversions, analytic functions, set operations, index creation, and probably a couple of other cases that I happen to have overlooked.

Although users of 9*i* onwards should be taking advantage of the automatic `workarea_size_policy` for memory management and using the `pga_aggregate_target` to set the accounting target for the bulk-memory operations such as sorting, this chapter starts with these features disabled. Remember that the automatic `workarea_size_policy` affects the quantity of memory a process is allowed to acquire, and how it frees it; but since I will initially be trying to demonstrate how Oracle uses that memory, it doesn't really matter whether the allocation strategy was something that Oracle handled dynamically, or something that I fixed manually.

As we did in Chapter 12, though, we will eventually move on to review the four possible combinations of **CPU costing** and `workarea_size_policy` settings. And, as with hash joins, we will find that the changes as we enable new features are more important than the absolute figures that come out of any calculations.

One of the key problems in understanding the cost calculation for sorting is that the cost formula from the 9.2 Performance Guide and Reference that I quoted in Chapter 1 does not include a component to represent the writes and reads that might be relevant to sort operations that spill to disk.

A further problem appears when you examine the `10053 trace` files, and find that there are several discrepancies between the model implied by the numbers and the actual activity. The dramatic variation between versions doesn't make the trace files any easier to comprehend.

Getting Started

As with hash joins, there are three levels of efficiency with sorting—**optimal**, **onepass**, and **multipass**.

- The optimal sort takes place completely in memory. We simply read and sort a stream of data, keeping everything in memory as we go. We run out of data before we run out of memory, so we don't have to use any disk space as a scratch working area. Contrary to popular belief, we only allocate memory gradually as we read data; we don't allocate the entire `sort_area_size` (or limit implied by the `pga_aggregate_target`) as we start the sort.

- The onepass sort appears when the data gets too big to fit in memory. We read as much data as we can deal with into the available memory, sorting it as we go, and when we reach our memory limit, we dump the sorted set to disk. We then read more data, sorting it as we go, and when memory is full again, we dump the next sorted set to disk, and so on until we have used up the entire input data stream. At this point, we have several sorted runs of data on disk, and have to merge them into a single set. We have a onepass sort if we have enough memory to read a chunk from every single sorted run at once. If you want an idea of what the merge step looks like, record a video of someone dealing a deck of cards, and then play the video backwards.

- The multipass sort starts like a onepass sort, but it comes into play when all the data has been used up and written to disk, and your session discovers that it does not have enough memory to hold a chunk from every single sorted run at once. In this case, you have to merge a few streams, writing a single larger stream back to disk, and then merge a few more streams, writing another larger stream back to disk. Eventually, you would have processed all the original streams, and can start reading back and merging a few of the larger streams—possibly writing even larger streams back to disk. Ultimately, you would be able to merge all the (possibly very large) streams that were left on disk. The number of times you have to read the data is referred to as the *number of merge passes*.

Let's build an example so that we can get some idea of what is going on with a simple sort operation. We start with a table of about one million rows (1,048,576—the closest power of two—to be exact) of pseudo-random, fixed-length data and run a simple query that sorts it.

As usual, my demonstration environment starts with an 8KB block size, locally managed tablespaces, 1MB uniform extents, manual segment space management, and system statistics (CPU costing) disabled and, as I pointed out in the chapter introduction, a manual `workarea_size_policy`. Moreover, for reasons that will become clear later, I have to point out that I am using a 32-bit operating system for this series of tests (`sort_demo_01.sql` in the online code suite).

My strategy for investigating sorting is going to be completely different from the approach I have used in the rest of the book. Instead of looking at `autotrace` or any other execution plan details, I am going to employ various trace events and dynamic performance views to show what's going on.

```
execute dbms_random.seed(0)

create table t1 as
with generator as (
        select  --+ materialize
                substr(dbms_random.string('U',4),1,4)        sortcode
        from
                all_objects
```

```
        where
                rownum <= 5000
)
select
       /*+ ordered use_nl(v2) */
       substr(v2.sortcode,1,4) || substr(v1.sortcode,1,2)    sortcode,
       substr(v1.sortcode,2,2)                               v2,
       substr(v2.sortcode,2,3)                               v3
from
        generator        v1,
        generator        v2
where
        rownum <= 1048576
;

--        Collect statistics using dbms_stats here

alter session set workarea_size_policy = manual;
alter session set sort_area_size = 1048576;

alter session set events '10032 trace name context forever';
alter session set events '10033 trace name context forever';
alter session set events '10046 trace name context forever, level 8';
alter session set events '10053 trace name context forever';

select  sortcode
from    t1
order by
        sortcode
;
```

Apart from the trace options that I have listed in the preceding code fragment, my sample program also captures statistics relating to file I/O, session activity, and session waits for the duration of the query, and some of the statistics about table `t1`. With all these collections going on, we can start cross-referencing various pieces of information to produce some interesting observations.

After using the `dbms_stats.gather_table_stats` procedure, I can check view `user_tables` to confirm that the table fills 2,753 blocks, and holds 1,048,576 rows with an `avg_row_len` of 14 bytes, and that the `avg_col_len` for column `sortcode` is 7 bytes (the `v2`, `v3` columns are there simply to avoid the possibility of some funny boundary conditions appearing). These 7 bytes include the 1 byte that holds the column length itself, so we can work out that the total volume of data sorted must be 6MB, and that my memory allocation of 1MB is not going to be sufficient for an in-memory (optimal) sort.

Sure enough, here are some corroborative statistics extracted from reports based on `v$mystat` and `v$tempstat`, respectively, when running the test case against 9.2.0.6 (scripts `snap_myst.sql` and `snap_ts.sql` will create packages to acquire and format this data):

```
Name                                     Value
----                                     -----
consistent gets                          2,758
physical reads direct                    1,565
physical writes direct                   1,565
table scans (long tables)                    1
table scan rows gotten                1,048,576
table scan blocks gotten                 2,753
sorts (disk)                                 1
sorts (rows)                          1,048,576

T/S      Reads   Blocks  Avg Csecs     Writes   Blocks  Avg Csecs
----     -----   ------  ---------     ------   ------  ---------
TEMP       993    1,565      2.420        269    1,565       .710
```

In the session statistics, we can see that we have done one long tablescan and one sort to disk. The number of rows scanned in the tablescan is the 1,048,576 that we expect. The figure for `consistent gets` (2,758) agrees reasonably well with the size reported for the table (2,753 blocks), as does the number of `table scan blocks gotten`. Finally, the number of `physical reads direct` and `physical writes direct` agrees with the number of block reads and writes reported from `v$tempstat`.

■ **Note** File-based statistics report read requests and blocks read as two separate figures, similarly write requests and blocks written, while the session statistics report the number of blocks written and read with names that makes it look as if they were the number of I/O requests.

SESSION STATISTICS

Many people use the view `v$sesstat` to check the statistics for a specific session. However, if the session you wish to monitor is your own, there is a more precisely targeted view called `v$mystat` that reports only the statistics for the current session.

For convenience, I usually create a view called `v$my_stats` in the sys schema to join `v$mystat` to `v$statname` (see scripts `c_mystats.sql` and `snap_myst.sql` in the online code suite).

`Event 10032` reports the statistics about activity that took place for a sort, `event 10033` lists details of the I/O that took place, `event 10046` is the extended trace that I have enabled at level 8 to record wait states ("eight" for "wait," if a rhyming mnemonic helps), and `event 10053` is the CBO trace. So let's see how some of their outputs tally with the statistics we have seen so far.

`Event 10033` shows the following:

```
*** 2005-01-20 09:35:02.666
Recording run at 406189 for 62 blocks
Recording run at 4061c8 for 62 blocks
Recording run at 406206 for 62 blocks
```

```
        19 similar lines deleted

Recording run at 4066de for 62 blocks
Recording run at 40671c for 62 blocks
Recording run at 40675a for 62 blocks
Recording run at 406798 for 15 blocks
Merging run at 406798 for 15 blocks
Merging run at 406189 for 62 blocks
Merging run at 4061c8 for 62 blocks

        19 similar lines deleted

Merging run at 4066a0 for 62 blocks
Merging run at 4066de for 62 blocks
Merging run at 40671c for 62 blocks
Merging run at 40675a for 62 blocks
Total number of blocks to read: 1565 blocks
```

As Oracle dumps sorted data to disk, producing **sort runs**, it records the volume of data dumped and the database block address in the **temporary tablespace** where each sort run starts. We could add up the number of blocks written and read, but conveniently the total for this onepass sort appears at the bottom of the output—and it agrees with the direct reads and writes, and the blocks read and written figures in `v$mystat` and `v$tempstat`, respectively.

The (hex) numbers, such as `4061c8`, are the block addresses where each sort run starts. So if we want, we can start dumping raw blocks to see what's in them. Convert the hex to decimal (`0x4061c8` is 4,219,336) and then use the `data_block_address_file` and `data_block_address_block` functions from the `dbms_utility` package to extract file and block numbers. (Bear in mind that for systems with lots of tablespaces and files, the file number is the absolute file number, not the tablespace-relative file number.) In this case, block 4,219,336 corresponds to block 25,032 of (temporary) file 1. So we can issue a `dump` command and check the content of the trace file as follows:

```
alter system dump tempfile 1 block min 25032 block max 25032;

Start dump data blocks tsn: 2 file#: 1 minblk 25032 maxblk 25032
buffer tsn: 2 rdba: 0x004061c8 (1/25032)
scn: 0x0000.031652d5 seq: 0x01 flg: 0x0c tail: 0x52d50801
frmt: 0x02 chkval: 0x65b6 type: 0x08=unknown
Dump of memory from 0x10593014 to 0x10594FFC
10593010          004061C8 0000003F 004061C9      [.a@.?....a@.]
10593020 00000000 00000041 00001FE8 00000000  [....A..........]
10593030 000002A6 00000008 41410006 45425242  [.........AABRBE]
10593040 00000008 41410006 584A5242 00000008  [......AABRJX....]
10593050 41410006 484F5242 00000008 41410006  [..AABROH......AA]
10593060 45535242 00000008 41410006 42555242  [BRSE......AABRUB]
```

The trace file isn't really exciting, but you will notice one detail—in the section to the right of the data dump, fifth line down, you can see that six dots appear between `AABROH` and `AABRSE`. In sorting our 6-byte strings, there is an overhead of 6 bytes per entry in the dumped data.

In this example, we can explain the overhead by looking at the entry for AABRBE a couple of lines higher up. After sorting out the byte-swapping for the platform, we can see that this matches the numeric dump 00000008 41410006 45425242 on the same line. We have

6 bytes for the length of the entry	00000008	--
2 bytes for the length of the column	0006	--
6 bytes of data	414142524245	-- after byte reordering

Although it's not visible in the example, entries are also zero-padded to a multiple of 4 bytes.

The formatting of data that gets dumped to disk during a sort means that the total volume dumped may be somewhat larger than you expect—we dumped a total of 1,565 blocks, which is over 12MB, when sorting only 6MB of raw data. The overhead looks particularly expensive in this case because we are sorting a very short row.

But there's another puzzle about the sort runs written to disk. Why are they 62 blocks long when I have a block size of 8KB and a sort_area_size of 1MB (or 128 blocks)—surely Oracle should be able to sort more than 62 blocks worth before running out of memory and dumping it to disk?

CHOOSING TEMPORARY EXTENT SIZES

You may recall that for many years, the directive on setting the extent size in the temporary tablespace was to use an exact multiple of the sort_area_size (sometimes the advice suggested an extra block or two) as Oracle would dump data equivalent to the sort_area_size when it had to dump at all. It always struck me as an attempt to be too precise—it was only when I discovered event 10033 that I realized how totally artificial this advice was.

In passing, if you are using proper temporary tablespaces and **global temporary tables** (GTTs), it is probably the characteristic use of GTTs that should dictate the uniform extent size of the temporary tablespace. Remember that every concurrent user of a GTT acquires one extent for their private copy of the table and one extent for each of its indexes. You might want small extents if you have a large number of concurrent users of GTTs.

Before solving the puzzle of the "missing" space, let's use a little trial and error to find out how large my sort_area_size has to be before I get an optimal (in-memory) sort (see script sort_demo_01a.sql in the online code suite). Table 13-1 shows memory settings, sizes of sort runs on disk, and (for reasons that will become apparent) the CPU time spent for different settings of sort_area_size.

Table 13-1. *Comparing sort_area_size with Resource Usage*

sort_area_size	Sort Run Sizes—Excluding Last	Number of Sort Runs	Blocks Written	CPU Seconds
1MB / 128 blocks	62 blocks	26	1,565	4.30
2MB / 256 blocks	122–123 blocks	13	1,552	4.44
4MB / 512 blocks	241–245 blocks	7	1,550	5.09

Table 13-1. *Comparing sort_area_size with Resource Usage*

sort_area_size	Sort Run Sizes—Excluding Last	Number of Sort Runs	Blocks Written	CPU Seconds
8MB / 1,024 blocks	484–492 blocks	4	1,549	5.71
12MB / 1,540 blocks	725–748 blocks	3	1,549	6.00
16MB / 2,048 blocks	970 and 577 blocks	2	1,548	6.02
25.5MB / 3,264 blocks		0		6.21

So, to sort a data set of 6MB, we need to allocate 25.5MB of memory before we can complete the sort in memory. Not only that, the amount of CPU that we use tends to increase as we use more memory. Does this tell us anything about how Oracle is doing its sorting?

Memory Usage

Oracle's sorting mechanism seems to be based on a **binary insertion tree**. What this means is that Oracle is effectively loading your data into a simple in-memory list, and maintaining an in-memory index on that data at the same time.

The significance of the word *binary* is that I believe the index is a binary tree, which means that each node in the tree has at most two children. As each new row of data is read, Oracle traverses the tree, using a simple binary chop mechanism to determine where that row should belong.

At the bottom of the tree, there are only two options. Either there is an available spot in the node where the data belongs or the node is full. If the node is full, Oracle has to add a new node, possibly by splitting the current node and propagating the split back up the tree as far as necessary. If this hypothesis is correct, this would result in a tree that was always height balanced—which means no long dangly bits hanging down from the main body of the tree—with a height that was always approximately log_2(*number of rows to be sorted*). For example, with eight rows loaded in an order that made a perfect tree, we would have a binary tree with a height of three ($Log_2(8) = 3$).

ORACLE'S SORTING ALGORITHM

I would particularly like to mention Richmond Shee (coauthor of *Oracle Wait Interface: A Practical Guide to Oracle Performance Diagnostics and Tuning*) for giving me some very important clues (see his paper to the IOUG-A conference 2004, "If your memory serves you right," about how Oracle manages its sorting).

Until I saw his paper, I had casually assumed that Oracle would be loading data into memory and then using one of the standard algorithms for sorting in place. It had not occurred to me that Oracle would be using an insertion tree mechanism for sorting, and reading his paper gave me a truly "Aha!" moment.

Figure 13-1 shows a simple schematic of how Oracle uses the sort memory. You will notice that I have drawn a tree and a separate data stack. I don't think Oracle keeps the actual row

values embedded in the tree, as the code and memory structures used would be much cleaner if the data were listed separately and the leaf nodes on the tree held pointers to the relevant row.

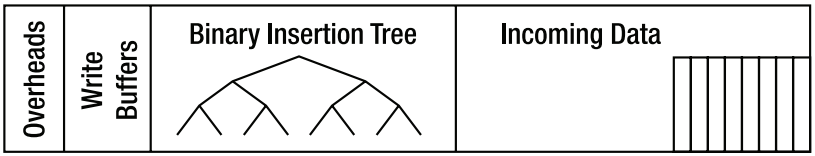

Figure 13-1. *Memory usage during a sort*

I've also simplified the diagram by showing the whole tree in just one chunk of memory, and the entire data stack in another chunk. In fact, Oracle allocates memory for sorting a little bit at a time as the data is read, so the tree and the data will be broken up into lots of little pieces, interleaved through the available memory.

When Oracle has completed the sort, or when it has to dump the data to disk in sorted order, it simply walks the tree, jumping around the data heap to supply the data in the correct order.

This sorting strategy would explain the unexpected memory requirement as follows:

- My guess is that each node on the tree is just a set of three pointers (parent, left child, right child), and for a 32-bit operating system, a pointer will be 32 bits, 4 bytes, so each node will be 12 bytes.
- Although my data items (in their raw form) were only 6 bytes each, by the time they had been put into the format we saw in the block dump, they had grown to 12 bytes each. Now we have to allow for 12 bytes per node, and the total number of nodes in a (fully populated) binary tree is one less than the number of items—note how the diagram shows eight data items and seven nodes on the tree.
- In effect, our single raw data item has grown from 6 bytes to 24 bytes—it's not surprising that we couldn't do a full in memory sort with less than the 25.5MB of memory we used, especially allowing memory for I/O buffers.

There is an important side effect, of course. I was running on a 32-bit system. What's going to change if I switch to a 64-bit system? Pointers are going to double in size. The extra 12 bytes per row due to the nodes is going to turn into an extra 24 bytes per row. If you are running a 64-bit operating system, the sort in the sample code won't complete in memory unless you set the `sort_area_size` to something like 38MB.

CPU Usage

Historically, the mantra has always been that you tune sorts by increasing the `sort_area_size` so that they complete in memory and don't spill to disk. But I've just demonstrated that increasing the `sort_area_size` increases the CPU usage—which seems to contradict the oral tradition. This, too, could be explained by the binary insertion tree algorithm.

Before addressing the contradiction, let's go back to another of the traces that I enabled—`event 10032`—and look at some of the information it provides. Running the same test with the initial 1MB for `sort_area_size`, the full `10032 trace` file looks like this:

```
---- Sort Parameters ------------------------------

sort_area_size                  1048576
sort_area_retained_size         1048576
sort_multiblock_read_count      2
max intermediate merge width    29

---- Sort Statistics ------------------------------

Initial runs                              26
Number of merges                          1
Input records                             1048576
Output records                            1048576
Disk blocks 1st pass                      1565
Total disk blocks used                    1567
Total number of comparisons performed     19863631
  Comparisons performed by in-memory sort 14869919
  Comparisons performed during merge      4993712
Temp segments allocated                   1
Extents allocated                         13
    ---- Run Directory Statistics ----
Run directory block reads (buffer cache)  27
Block pins (for run directory)            1
Block repins (for run directory)          26
    ---- Direct Write Statistics -----
Write slot size                           49152
Write slots used during in-memory sort    2
Number of direct writes                   268
Num blocks written (with direct write)    1565
Block pins (for sort records)             1565
Cached block repins (for sort records)    25
    ---- Direct Read Statistics ------
Size of read slots for output             16384
Number of read slots for output           64
Number of direct sync reads               435
Number of blocks read synchronously       461
Number of direct async reads              558
Number of blocks read asynchronously      1104
```

You'll see that the trace file starts by listing the critical sort-related parameters (`sort_area_size`, `sort_area_retained_size`, and `sort_multiblock_read_count`) and the derived figure `max intermediate merge width`. The `max intermediate merge width` is the number of sorted runs that Oracle will be able to read and merge simultaneously (and dump back to disk as one larger run) if the volume of data to be sorted pushes Oracle into a multipass sort.

The second part of the printout, the `Sort Statistics` section, is actually dumped twice—possibly once as Oracle finishes building the result set, then again as the cursor closes. I don't want to describe the entire report in detail, but you'll notice that the `Initial runs` figure (26) matches the number of lines in the `1033 trace` that say `Recording run at mmmmm for nn blocks`,

and the `Disk blocks 1st pass` figure (1565) also agrees with the summary line `Total number of blocks to read: 1565 blocks` in the `10033 trace`. The number of merges is one—this is a good thing and the ideal target for very large sorts that can't complete in memory. The number of merges will inevitably be one if the `Initial runs` value does not exceed the `max intermediate merge width`, and the session statistics will record this as a onepass workarea execution. (In fact, it is possible for the number of merges to be one even when the initial runs exceeds the `max intermediate merge width`, as we shall see in the upcoming section about the `pga_aggregate_target`.)

We can see the number of input records (1,048,576), but the interesting numbers are the number of comparisons: roughly 20,000,000 in total, 15,000,000 during the in-memory sorting, and 5,000,000 during the merge from disk. These numbers will vary with the randomness of the incoming data and the memory size, but for a fixed set of data, it is worth running a few examples to see if there are any patterns to the numbers (script `sort_demo_01a.sql` in the online code suite may help).

Table 13-2 lists a few results—the last two columns report some results derived from the preceding columns.

Table 13-2. *How sort_area_size Affects Number of Comparisons Needed*

sort_area_size	In-Memory Comparisons	Initial Runs	Merge Comparisons	Number of Rows * Log_2(Initial Runs)	Total Comparisons
1MB	14,869,919	26	4,993,712	4,929,000	19,863,631
2MB	15,969,150	13	3,946,598	3,880,000	19,915,748
4MB	16,984,029	7	2,981,434	2,944,000	19,965,463
8MB	17,967,788	4	2,097,141	2,097,152	20,064,929
12MB	18,564,704	3	1,604,693	1,662,000	20,169,397
16MB	18,884,122	2	1,048,573	1,048,576	19,932,695
24MB	19,622,960	2	1,048,555	1,048,576	20,671,515
25.5MB	19,906,600	0	0	n/a	19,906,600

From the table, you can see that the number of comparisons that take place during the merge from disk is remarkably close to *number of rows* * log_2(*number of sort runs*). This tells you that Oracle is probably keeping a little tree-like structure that points to the start of each run that has been reloaded, and effectively keeps walking down the tree to find which run has the next smallest item. (Alternatively, you could think of this tree as a list of the top item from each run, with Oracle doing a binary chop to see which top of list is currently the lowest value.)

If you recall that our test case has 1,048576 rows, and log_2(1,048,576) = 20, you will also notice that the number of comparisons for the memory sort is quite close to: *number of rows* * log_2(*number of rows*). The figure is not extremely accurate (it would have to be 20,971,520 to be perfect), and for some special cases the result can be quite a long way off—the code seems to have some optimizations that cater to runs of presorted values.

It was the appearance of this factor of $\log_2()$, combined with the three pointers per data-value, that led me to believe that the insertion tree was a binary tree, but the degree of error that sometimes appears may mean my assumption is wrong.

HEAPSORTS

It is possible that Oracle is using a form of the **heapsort**, which would start by loading data into a partly ordered tree structure known as a **heap**. (The word *heap* sometimes means something far more structured than the way teenagers store their belongings.) A heapsort then uses a shuffle-down mechanism to restructure the heap into a fully sorted structure.

I have discounted the heapsort as Oracle's mechanism for sorting because even though the number of comparisons used is appropriate, the optimum implementation of the heapsort algorithm needs only one pointer per entry rather than the three which, given the amount of memory involved, Oracle seems to use.

Interestingly, I have heard comments that the 10*g*R2 release is very much faster at sorting than previous releases—so perhaps the Oracle developers have finally changed the algorithm and the code will have switched to a heapsort by the time you read this note.

The final column in the table reports the total number of comparisons performed. As you can see, this doesn't really change significantly, as the balance shifts from in-memory sorting to merge sorting. But the previous table showed that we used increasing amounts of CPU as we shifted from the disk based to the memory-based sort. How could this happen?

It seems a little unlikely that the CPU time of an actual comparison between data items is different in the two types of operation—but it could take more CPU to walk up and down the tree to find something to compare with! When we did the full in-memory sort, we had a pretty tall tree to traverse (the height of the tree is $log_2(number\ of\ rows)$). But when we sorted a small fraction of the data (4% in our smallest test case earlier) before dumping it, each individual in-memory sort run produced a much shorter tree—so the total CPU time spent traversing trees was a lot less. And the final merge is only traversing a very short tree as well. It's not the data comparisons that cost us CPU, it's traversing the pointers that make up the trees.

It is important to realize that if your bottleneck is CPU and your I/O subsystem is not under pressure, then you may be able to improve your performance on large sorts by switching from in-memory sorts to using the minimal amount of memory that can produce a onepass sort.

It is also important to note that if you can't do an in-memory sort, you will be dumping the entire data set to disk anyway—you do not get a "little extra" benefit by having a "little extra" memory, it's all or nothing—so you want to set your `sort_area_size` to the smallest value you can get away with that doesn't push the sort into a multipass operation. (When we get to the automatic `workarea_size_policy`, we will see that this strategy seems to be built into the run-time code.)

■ **Note** Contrary to the traditional advice, there are cases where you can make sort operations complete more quickly by reducing the memory, and sorting to disk. The most dramatic example I have seen of this counterintuitive result was a series of large index builds where a typical index build time dropped from 150 seconds to 85 seconds (140 CPU seconds down to 75 CPU seconds) because we dropped the `sort_area_size` from 500MB to 5MB. However, that was a system with very fast disks and an enormous cache between Oracle and the disks—the I/O subsystem was under absolutely no pressure whatsoever.

sort_area_retained_size

The sort_area_retained_size can also be set to modify the behavior of sorting. The default value is 0, which means the sort_area_retained_size should dynamically adjust itself to match the current setting of the sort_area_size.

Setting the sort_area_retained_size to something other than 0 has two effects. First, it changes the way Oracle acquires and releases memory, and, second, it affects the way that Oracle uses the temporary tablespace.

SHARED SERVERS AND P_A_T

If you are using Shared Servers (or Multithreaded Server —MTS—as it used to be called), then in 9*i* the memory for bulk-memory operations such as sorting, hash joins, and so on, *still* uses the parameters relevant to a manual workarea_size_policy rather than the limits dictated by the pga_aggregate_target, and any memory allocated in the user global area (UGA) for these operations is based in the shared or system global area (SGA), not in the process global area (PGA). This changes in 10*g*.

Go back to script sort_demo_01.sql and run two tests, restarting your SQL*Plus session for each test. For the first test, set the sort_area_size to 30MB and do not set the sort_area_retained_size; for the second test, leave the value for the sort_area_size at 30MB and set the sort_area_retained_size to 8MB (this test is implemented as script sort_demo_01a.sql in the online code suite) After each test, check the tempfile I/O figures and the changes in session statistics relating to sorts and memory usage. Table 13-3 shows what I got under 9.2.0.6.

Table 13-3. *Impact of sort_area_retained_size on I/O*

Statistic	sort_area_retained_size = 0	sort_area_retained_size = 8MB
session UGA memory max	25.5MB	8.3MB
Session PGA memory max	25.8MB	28.5MB
Physical writes direct	0	1,547
sorts (disk)	0	0

The most significant thing about these statistics is the physical writes direct—by specifying a non-zero sort_area_retained_size we have made Oracle dump the entire data set to disk before returning results to the client (which may also explain the extra PGA memory—we need it for the I/O buffers, which we wouldn't otherwise be using).

I have included the number of sorts to disk reported by v$sesstat, just to point out that Oracle does not consider this to be a sort to disk—the sorting was all over before the data was dumped to disk as a holding location.

This is the way things happen when the sort_area_retained_size is set. The first allocation of memory is in the UGA, but when the allocation reaches the limit set by sort_area_retained_size, memory is then allocated in the PGA. Even if a sort completes in memory, the entire sorted data set will be dumped to disk if that memory has been split between the UGA and the PGA.

SO WHEN IS MEMORY ALLOCATED?

It is important to remember that the sort_area_size and sort_area_retained_size represent limits on memory allocation. You may still hear the suggestion from time to time that a session acquires the amount of memory defined by sort_area_size as it starts. This is not true. The memory is only allocated as needed for sorting, and even then it is only allocated a bit at a time, not all at once.

The same error has also been made about the pga_aggregate_target. There have been a couple of articles stating that the whole of the pga_aggregate_target is actually acquired by the instance at instance startup. This is completely incorrect. First, the parameter is simply a number that is used for accounting purposes (and you can go over budget); secondly, the memory is never acquired "by the instance," it is acquired and released by individual processes only as and when needed.

As I pointed out earlier in this chapter, if you are using Shared Servers (MTS), then the UGA is in the SGA, so memory up to the sort_area_retained_size will be allocated from the SGA—typically the **large pool**, although if you have forgotten to specify a value for the parameter large_pool_size, it will come from the **shared pool**. The excess demand, up to (sort_area_size – sort_area_retained_size) will be allocated in the PGA, which means it will appear in the memory of the Shared Server process (i.e., the processes that are called things like ora_{SID}_sNNN—at which point it will be accounted by the code that handles the pga_aggregate_target calculations).

If the sort_area_retained_size is less than the sort_area_size, then the sorted data will be dumped to disk as the sort completes, the excess memory allocated in the PGA will be made available for other activity, and the sort_area_retained_size (possibly limited to 2MB according to some recent experiments) will be used to reload the dumped data for further processing.

pga_aggregate_target

At the start of this chapter, I made the comment that the mechanics of sorting don't change if you use a manual workarea_size_policy and sort_area_size rather than the newer automatic workarea_size_policy and pga_aggregate_target. The time has come to demonstrate this fact.

Using the automatic workarea_size_policy, the DBA sets an accounting target for Oracle that defines a limit on the total amount of memory that should be in use across all processes for the bulk-memory operations such as sorting, hash joins, creating indexes, and bitmap index operations. Various parts of the Oracle kernel code then cooperate to check current memory allocation, avoid allocating excessive memory, and release memory that is not needed.

Of course, memory related to PL/SQL operations (such as array processing) falls outside the *control* of this mechanism, although the allocation is still *tracked* for accounting purposes and has an impact on the amount of memory allowed to other work areas. (You may have problems with systems where sessions connect and disconnect extremely rapidly, as checking of the limit only occurs every three seconds.)

There are many parameters (mostly hidden) affecting the details of operation—but there are two critical details that matter most: by default, any one work area for a serial operation is allowed a maximum of 5% of the pga_aggregate target, and a work area for a parallel operation is limited so that the total across all slaves involved in that operation is no more than 30% of the pga_aggregate_target—with an upper limit of 5% for each slave, which means that you only see the effect of the 30% limit if you have queries with a degree of parallelism greater than six (since 5 * 6 = 30).

The hidden parameter _smm_max_size shows the limit for serial execution, and _smm_px_max_size shows the limit for parallel execution. A further parameter is the _smm_min_size, which shows the smallest effective memory allocation that a session will get for a work area. In 9.2.0.6, this seems to default to 0.1% of the pga_aggregate_target, with a minimum value of 128KB, and a maximum of 1MB. Given that a single session may have several bulk-memory operations running concurrently, there is a further parameter that is the limiting target for a single session, the _pga_max_size that defaults to 200MB; and in fact, a secondary limit on the default value for the _smm_max_size is that it should be no more than half the _pga_max_size (although we saw in Chapter 12 that hash joins seem to base their arithmetic on double the _smm_max_size and may be allowed some leeway in overshooting the _smm_max_size at run time).

We can get some idea of how Oracle makes best use of memory by running the same sample code as before (see script sort_demo_01b.sql in the online code suite) but setting workarea_size_policy to its default value of auto with a pga_aggregate_target of 200MB. The effect is that our process has a limit of 10MB (5% of 200MB) imposed on the sort operation, and we know that our sample code needs about 25.5MB for an in-memory sort.

After running the query, we see in v$mystat that the UGA memory max has gone up by 9.7MB, and the PGA memory max has gone up by 10.7MB. So it looks as if the session has been allowed to acquire the full 10MB limit to do this sort. However, if we look at the 10032 and 10033 trace files, we observe an oddity (the following is an extract of the critical details):

```
Recording run at 404489 for 598 blocks
Recording run at 4042e0 for 127 blocks
Recording run at 40425f for 128 blocks
Recording run at 4041df for 162 blocks
Recording run at 404181 for 198 blocks
Recording run at 403947 for 226 blocks
Recording run at 403a29 for 111 blocks
```

```
---- Sort Parameters ------------------------------
sort_area_size                    3555328
sort_area_retained_size           3555328
sort_multiblock_read_count        31
max intermediate merge width      5

---- Sort Statistics ------------------------------
Initial runs                              7
Number of merges                          1
```

The most obvious anomaly is that Oracle has reported the `sort_area_size` as 3.5MB, when we know that the memory demand must have peaked at around 10MB.

Then we notice the sizes of the sort runs: the first one was 598 blocks (about 4.5MB), followed by several more modestly sized runs in the region of 1MB to 1.7MB. Given our knowledge of how much "excess" memory Oracle needs for sorting, we know that the first sort run is consistent with a 10MB `sort_area_size`, and the rest are consistent with a `sort_area_size` of around 2MB to 3.5MB.

You might also notice that the `sort_multiblock_read_count` of 31 is much higher than the value of 2 that we saw in our earlier `10032 trace`, with a corresponding reduction in the `max intermediate merge width`.

There is even a hint of an anomaly in the `max intermediate merge width`—although it has the value 5, the next few lines from the `10032 trace` tell us that we produced seven sort runs, but merged them all in one pass. The key is in the word *intermediate*. The mechanism (or possibly just the reporting) varies depending on whether the `workarea_size_policy` is set to `manual` or `auto`, but the run-time engine has some room to be flexible on the final merge pass of a sort operation, as it can determine that it will be possible to read and merge all the streams in one go if it avoids reserving any memory for writing streams back to disk. Hence my comment earlier on that the number of merge passes could still be one even though the number of initial runs exceeds the `max intermediate merge width`.

THE SORT_MULTIBLOCK_READ_COUNT PARAMETER

The `sort_multiblock_read_count` dictates the number of blocks that a process can read from a single sort run in one read. A larger read size may allow your hardware to offer better performance on the merge, but since there is a strict limit on the total memory available for the merge, a larger read size reduces the number sort runs that can be read and merged simultaneously. If you cannot merge all the runs in one go, you have to merge a few at a time, rewriting the results as you go, and then do extra merge passes with the larger runs you have just produced. This is reported from 9*i* onwards in the session statistics as `workarea executions - multipass` and is something that you normally want to avoid.

Historically, Oracle Corp. advised against modifying the `sort_multiblock_read_count`, and the parameter usually seemed to set itself to two (or one) blocks—even when the available memory was very large and the upper limit for the `max intermediate merge width` (678) had been reached. If you are still running 8*i* (or 9*i* with manual workarea sizes), you may find that you can get a little performance edge for very large sorts by checking the `10032 trace` and adjusting the `sort_multiblock_read_count` for a specific session or specific statement.

The strange contradictions in the trace reports can be explained by the optimistic, yet thrifty, mechanisms used by the run-time engine. Because the pga_aggregate_target was set to 200MB for the test, my process had a limit of 10MB for the sort. Because nothing else was going on during my test, the kernel code allowed my session to grow to that limit—effectively checking view v$pgastat (and possibly summing v$sql_workarea_active) to compare the global target with the total amount of PGA memory currently allocated.

So the sort operation starts with an effective sort_area_size of 10MB; but on the first pass—when the 10MB is completely filled for the first time by the data and its insertion tree—it becomes apparent that the sort cannot complete in memory. The session dumps the first run to disk, and releases memory back to the accounting pool. I suspect that there is some calculation at this point that estimates a small but safe size of memory that will allow the sort to complete as a onepass sort. The ideal is to hold on to enough memory to keep the operation down to a onepass sort, but not to have so much that (a) the CPU cost is unnecessarily increased, or, more importantly, (b) other processes in need of memory are starved as a result of our greed.

Given the steady increase that we can see in the sizes of the sort runs, it is possible that the session checks the state of the PGA global allocation each time it dumps to disk. It is also possible that part of the calculation is keeping a check on the number of runs generated and anticipating the amount of memory needed to perform a single merge with the minimum I/O overheads.

There is one more significant detail about the 10032 trace that doesn't stand out in the extract I've printed. In 8*i*, the Sort Parameters section of the trace file is printed before the sort runs start—in 9*i* the figures are printed after the first pass to disk has completed. The figures in the trace file reflect the state of the PGA just after the final sort run is dumped; and this just happens to catch the fact that the new code has the capability to take advantage of larger memory allocations by electing to use a much larger I/O size during merging. (The minimum and maximum for this I/O size is now set by hidden parameters _smm_auto_min_io_size and _smm_auto_max_io_size.) Whatever else the optimizer may be doing with its arithmetic in 9*i*, it is not using the run-time value of sort_mulitblock_read_count reported in the 10032 trace.

Real I/O

One final point to consider before moving on to the way the optimizer generates a cost estimate for a sort is the real impact of the I/O that we've seen.

Looking back at the 10032 trace for the original test with the 1MB sort_area_size, we wrote 1,565 blocks to disk in sort runs that were typically 62 blocks each and then had to reread them to merge them. On the other hand, although the results from v$tempstat reported the same number of blocks written and read, the number of write requests was reported as 269 and the number of read requests as 993. It would be interesting to check what type of writes and reads actually took place, how large they were, and whether they can be adjusted.

In the code extract from the original demonstration script, one of the events that I enabled was event 10046—the "extended trace" event—enabled at level 8 so that I could track the wait states that showed up. Running tkprof against this trace file, I got the following results for the main query:

```
Event waited on                             Times   Max. Wait  Total Waited
----------------------------------------   Waited  ----------  ------------
direct path write                               2        0.00          0.00
direct path read                              435        0.59         15.00
```

This isn't a good match for the reads and writes recorded against the temporary files. So I took a close look at the trace file itself. The following lines are eight consecutive lines from a trace file for a typical test run. I have put in a couple of blank lines to clarify the change of function:

```
WAIT #1: nam='db file scattered read' ela= 33985 p1=5 p2=1847 p3=6
WAIT #1: nam='db file sequential read' ela= 246 p1=5 p2=1875 p3=1

WAIT #1: nam='direct path write' ela= 6 p1=201 p2=16799 p3=6
WAIT #1: nam='direct path write' ela= 22 p1=201 p2=16805 p3=2

WAIT #1: nam='direct path read' ela= 293 p1=201 p2=16792 p3=2
WAIT #1: nam='direct path read' ela= 20087 p1=201 p2=19206 p3=2
WAIT #1: nam='direct path read' ela= 6655 p1=201 p2=19081 p3=2
WAIT #1: nam='direct path read' ela= 3685 p1=201 p2=19144 p3=2
```

The first two lines are the tail end of reading the table—typically multiblock reads against data file 5 (`p1 = 5`) as we scan the table, with one single-block read to catch the last block of the table.

The next two lines are the only reported waits for writes to the temporary tablespace (file 201 on my test system). On several tests of the script, the number of writes reported varied only very slightly—with a maximum of four writes appearing in just one test run. The last reported write in every test was two blocks, the rest were always six blocks.

TEMPORARY FILE NUMBERING

Numbering for temporary files may seem a little odd. If you subtract the value of parameter `db_files` from the number recorded in the trace file, you get the file number recorded in `v$tempfile`. Alternatively, you can query `x$kcctf` for column `tfafn` to list the **Temporary File Absolute File Numbers**.

Oracle can use a form of asynchronous write mechanism for direct path writes (if the operating system supports this feature), sending a few batches of blocks to the I/O subsystem, and then checking back a little later to see if the batches have been written. Some of the details of this activity are visible in the `10032 trace` for this test, where we see the statistics about direct writes and direct reads (the following is an extract). I assume the figures for `Write slots used` and `Number of read slots` tell us how many asynchronous calls Oracle can make before cycling back to see whether a batch has been written or read. I've added a couple of lines to this sample that didn't actually appear in my first test, and are relevant only during multipass operations:

```
---- Direct Write Statistics -----
Write slot size                          49152
Write slots used during in-memory sort   2
Write slots used during merge                                  -- only in multipass
Number of direct writes                  268
Num blocks written (with direct write)   1565
Waits for async writes                                         -- did not appear in 9i
```

```
---- Direct Read Statistics ------
Size of read slots for merge phase                          -- only in multipass
Number of read slots for merge phase                        -- only in multipass
Size of read slots for output              16384
Number of read slots for output            64
Number of direct sync reads                435
Number of blocks read synchronously        461
Number of direct async reads               558
Number of blocks read asynchronously       1104
```

Notice how the number of direct sync reads (435) in the `10032` statistics matches the number of direct read waits reported in the `10046 trace` file. Moreover, the total number of reads (435 + 558) reported in the `10032 trace` matches the number of reads reported in the `v$tempstat` file statistics (993).

It's a pity that the `number of direct writes` in the `10032 trace` (268) doesn't quite match the number of writes reported in `v$tempstat` (269), but the 9*i* trace was missing a line about async writes that appeared in the 8*i* and 10*g* traces, so perhaps the discrepancy is a simple accounting error somewhere in the async code.

Going back to the `10046 trace`, the last four lines from the trace show Oracle rereading blocks from the sort segment to merge them. The `p2` value listed in these four lines is the starting block number in the file for the read, and shows that Oracle is jumping all over the place in the file; but there were 26 consecutive read waits at the start of the list, corresponding to the 26 lines saying `Merging run at mmmm for nn blocks` in the `10033 trace`.

The `p3` value is the number of blocks read. After starting with two blocks for each read (the limit set by the `sort_multiblock_read_count`), all subsequent reads in the trace file were just one block each—and they left gaps all over the place in the sort runs that had previously been written, which is probably the effect of having lots of `read slots for output` available for async read requests.

Given the reports of asynchronous writes and reads in the `10032 trace`, it is no surprise that we can't get the numbers to match up. In fact, we even find that the number of read waits in the `10046 trace` shows quite clearly that sometimes we end up waiting for async reads, as well as sync reads.

Cost of Sorts

Once you know how sorting works, it is possible to look at the numbers that the optimizer produces to estimate the cost of a sort, and try to invent a rationale that would explain how those numbers have been generated. Of course, there are four different scenarios to deal with if you want complete coverage:

`workarea_size_policy = manual`	CPU costing disabled	(8*i* style)
`workarea_size_policy = auto`	CPU costing disabled	(default 9*i* style)
`workarea_size_policy = manual`	CPU costing enabled	(common 9*i* style)
`workarea_size_policy = auto`	CPU costing enabled	(strategic 9*i* and default 10*g* style)

Then you have to worry about the problem of generalization: Is there anything different for `sort (aggregate)`, what about sorting to create an index; and is the cost of sorting in a sort/merge join in any way different from a simple sort? Finally, of course, there are the O/S issues—does the costing mechanism make any allowance for the differences between 32-bit and 64-bit platforms or the availability of asynchronous I/O?

And before I say anything about how the optimizer calculates the cost of sorts, I would like to draw your attention to a hidden parameter that appeared in 9*i*, `_new_sort_cost_estimate`. The default value for this parameter is `true`, and the description reads `enables the use of new cost estimate for sort`.

Inferring the actual algorithms of the optimizer by checking the results from controlled experiments is virtually impossible—especially when there are so many changes between releases, and so many little tweaks that might be due to version, might be bugs, or might be special cases. I'm not claiming a complete understanding of the costs for sorts and merge joins—but the following notes may help you deal with problems.

10053 trace

We'll start with the trace event I set in the very first test—the CBO or `10053 trace`. When I had a manual `workarea_size_policy`, a `sort_area_size` of 1MB, and CPU costing disabled, the critical sections of the trace file look like this:

```
SINGLE TABLE ACCESS PATH
  TABLE: T1     ORIG CDN: 1048576  ROUNDED CDN: 1048576  CMPTD CDN: 1048576
  Access path: tsc  Resc:  266  Resp:  266
  BEST_CST: 266.00  PATH: 2  Degree:  1

Join order[1]:  T1[T1]#0
ORDER BY sort
    SORT resource      Sort statistics
      Sort width:   29 Area size:   712704  Max Area size:   712704    Degree: 1
      Blocks to Sort:     2311 Row size:           18 Rows:    1048576
      Initial runs:         27 Merge passes:        1 IO Cost / pass:       2711
      Total IO sort cost: 2511
      Total CPU sort cost: 0
      Total Temp space used: 33711000
Best so far: TABLE#: 0  CST:       2777  CDN:    1048576  BYTES:    7340032
```

I have included the single-table access path in the extract to show you the cost of the full tablescan (266) that Oracle used to acquire the data that was to be sorted.

The last line printed shows the best cost so far as 2,777, which is the sum of 266 (the tablescan cost) and 2,511 (the `Total IO sort cost`). We need to understand how the optimizer derives the value 2,511 for the `Total IO sort cost`.

Before getting into the difficult bits, I'd like to explain the other numbers that appear in the second section of the output. In order:

- `Sort width`: This is the same as the `max intermediate sort width` we saw in the `10032 trace`.
- `Area size`: This is the amount of memory available for processing the data. Our `sort_area_size` was 1MB, so the size reported here seems a little small. In the 8*i*-style calculations, the optimizer allows 10% (rounded to an exact multiple of the block size) of the `sort_area_size` for write buffers. This still leaves a lot missing—which according to my theory is the allowance built into the model for the binary insertion tree. Subtract 10% (rounded to the next complete block) from the initial value, and then subtract 25% of the rest for the sort tree: 1,048,576 bytes = 128 blocks (at 8KB); subtract 13 blocks (10%) and you get 115 blocks; subtract 25%, and you get 86.25 blocks—rounding up to the nearest block this is 87 blocks—712,704 bytes, which is what we see here. Note, however, that this 25% allowance does not change as the row size changes (which it probably ought to if my theory were correct); nor does it change as you move from 32-bit Oracle to 64-bit Oracle.
- `Max Area size`: Since this is a 9*i* trace file, the `Area size` reports the minimum allocation size available based on the setting of the `pga_aggregate_target`, and the `Max area size` reports the maximum area that the session will be allowed. Since this test case is running 8*i*-style, using the manual workarea size policy, the `Area size` and `Max area size` are identical. When the two values differ, the calculation of cost seems to be based on the `Max area size`, rather than the `Area size`.
- `Degree`: If you are allowing parallel execution, you will get two completely separate, identically structured sections of trace file showing the cost calculations for the sort. One will show degree 1, the other will show the degree of parallelism set for the query (which may be derived from a hint, a specific degree of parallelism on a table, or the automatic degree dictated by Oracle if you have enabled parallel automatic tuning).
- `Blocks to Sort`: A measure of the data to be sorted. Derived as `Row size` * `Rows` / block size. If you have enabled parallelism, the value for `Blocks to Sort` in the second (parallel) set of figures will be the value from the first (serial) set of figures divided by the degree.
- `Row size`: The optimizer's estimate of the average size of the rows to be sorted. Allowing for a couple of little adjustments and variations, this is derived from column `avg_col_length` of view `user_tab_columns`, and is usually (12 + `sum`(`avg_col_length`) + (`n` – 1)) (where `n` is the number of columns to be sorted). The fixed value 12 covers the cost of the tree node quite nicely, but the rest of the formula seems inappropriate as we have already seen that our 6 bytes of column data had a 2-byte column overhead and a 4- byte row overhead added. It's just another spot where the model for calculation doesn't quite agree with the run-time activity.

DIFFERENCES BETWEEN DBMS_STATS.GATHER_TABLE_STATS() AND ANALYZE

One difference between these two methods of gathering statistics is that the `avg_col_len` generated by the `dbms_stats` package includes the length byte, but the `analyze` command does not.

Since the `Row size` used by the sort calculation uses the `sum(avg_col_len)` from view `user_tab_columns`, this means that the cost of a sort will be slightly higher following a call to `dbms_stats` than it would be if you carried on using the `analyze` command. (The difference is also relevant to hash joins.)

There are lots of ways that the row size can be a bad estimate. For example, if you use a simple function of two columns (e.g., `substr(col1 || col2,5,2)`) the optimizer will add the column lengths of both columns in the calculation when it "obviously" ought to use the constant 2.

- `Rows`: This is the 1,048,576 that is the computed (filtered) cardinality for the table. Again, if you have enabled parallelism, the value in the second (parallel) set of figures will be the value from the first (serial) set of figures divided by the degree. Because the number of rows is scaled down while the `Area size` may stay unchanged, the cost of a parallel sort can easily be much lower than the cost of a serial sort—even to the point where it is lower than the serial cost divided by the degree.

- `Initial runs`: The optimizer's estimate of the number of sort runs that will be dumped to disk. You can derive this as `Blocks to Sort` * block size / `Max Area size`. (2,311 * 8,192 / 712,704 = 26.56). In this example, the optimizer has managed to get the right (observed) result by using a strange number for the area size, and a wrong number for the row size. Both values (`Blocks to Sort` and usable `Area size`) are larger than what actually occurs at run time.

- `Merge passes`: Luckily, the `Initial runs` is less than the `Sort width`, so the optimizer has decided that all the runs can be merged in a single pass. This is the significance of `Merge passes`. Unfortunately, it always seems to be at least one—even for an in-memory sort. This figure counts the number of times the entire data set will be written to and read back from disk if you go into a multipass sort. This is not the same as the number of merges recorded in the `10032 trace`—a single merge pass (from `10053`) could require many merges (from `10032`). Imagine you had 15 sort runs on disk, but could only read three at a time, then the first merge pass would require five (15 / 3) merges and produce five larger sort runs on disk.

- `IO Cost / pass`: The cost of doing a single merge pass. (In future I will refer to this as `IO cost per pass` to avoid confusion when I want to use the division symbol.) This ought to be a number that reflects the need for the writing and rereading that has to be done if the sort cannot be completed in memory. If we have a onepass workarea execution, then we will have dumped the entire data set once, and read it once to merge it. If we couldn't merge all the data in a single pass, we will have dumped it into larger runs on disk by the end of the first merge attempt, and then reread again to merge the larger dumps. Although there is some scope for optimization with multiple merges, we have to keep dumping the entire data set time and time again—so we only need to know how much extra I/O

we do when we dump it once, and multiply by the number of passes it takes to get to the final merge. This figure is the one that we most need to understand, and one subject to a lot of change with versions and features. The `sort_multiblock_read_count` will affect it in 8*i*, but not in 9*i*, where the parameter value seems to be ignored.

- `Total IO sort cost`: Presumably this is supposed to combine the cost per pass with the number of passes—but before the introduction of CPU costing, this always seemed to be smaller than the cost per pass on a onepass merge. A typical result would be (`Blocks to Sort` + `IO cost per pass` * `Merge passes`) / 2 as it is in this example (and the fixed value of 2 in this formula is not there as a consequence of the deprecated parameter `_sort_multiblock_read_count`). When CPU costing is enabled, the cost usually seems to be (`Blocks to Sort` + `IO cost per pass` * `Merge passes`)—although there are special boundary conditions where the division by 2 reappears.

- `Total CPU sort cost`: The CPU component of the cost, measured in CPU operations—which will be converted to a (relatively) small I/O equivalent cost later on in the trace file.

- `Total Temp space used`: In principle, the amount of temporary space we need for our sort operation. We actually used 1,565 blocks (12.5MB) rather than the 32MB reported here. Bear in mind, though, that the optimizer has estimated 2,311 blocks (18MB) to be sorted, which reduces the error somewhat.

The trace extract shown previously comes from 9*i* emulating the model of 8*i*, but if we switch to the strategic 9*i* configuration (CPU costing enabled, and `workarea_size_policy` set to `auto`), we see a number of significant changes to the trace file.

In the following extract, I have a `pga_aggregate_target` of 200MB, which results in the `_smm_min_size` being 204KB and the `_smm_max_size` being 10MB. (0.1% and 5% respectively of the target). I have also set the system statistics (using the `dbms_stats.set_system_stats()` procedure) in a way that causes the calculation of tablescan costs to mimic the arithmetic used for tablescans when CPU costing is disabled and the `db_file_multiblock_read_count` is set to 8 (see script `sort_demo_01b.sql` in the online code suite).

```
SINGLE TABLE ACCESS PATH
  TABLE: T1     ORIG CDN: 1048576  ROUNDED CDN: 1048576  CMPTD CDN: 1048576
  Access path: tsc  Resc:  283  Resp:  283
  BEST_CST: 283.00  PATH: 2  Degree:  1

Join order[1]:  T1[T1]#0
ORDER BY sort
    SORT resource      Sort statistics
      Sort width:   58 Area size:  208896 Max Area size: 10485760   Degree: 1
      Blocks to Sort:     2311 Row size:           18 Rows:    1048576
      Initial runs:          2 Merge passes:        1 IO Cost / pass:     580
      Total IO sort cost: 2891
      Total CPU sort cost: 1011773433
      Total Temp space used: 33711000
Best so far: TABLE#: 0  CST:       3276  CDN:    1048576  BYTES:    7340032
```

```
Final - All Rows Plan:
  JOIN ORDER: 1
  CST: 3276  CDN: 1048576  RSC: 3275  RSP: 3275  BYTES: 7340032
  IO-RSC: 3157  IO-RSP: 3157  CPU-RSC: 1181444018  CPU-RSP: 1181444018
```

The cost of the tablescan in the single-table access path has gone up from 266 to 283. Although it is not explicitly displayed in this trace file, we can determine (by doing a separate `10053 trace` of a simple select against the table) that the difference of 17 is the calculated CPU cost of scanning the table.

Now look at the second section of the output: first, we can see that the `Area size` is down to 204KB, and the `Max area size` is 10MB—exactly as expected. But pause for thought—when we set a `sort_area_size`, we used to see about 67.5% (75% of 90%) of that value appearing in these columns, but here we are seeing the entire 100%; does this mean that the write buffers and insertion tree are catered to elsewhere, or is it simply that the report is being used differently?

In fact, if we adjust the number of rows used in the test, we find that the number of initial runs drops to one—and the number of merge passes to zero—only when `Rows * Row size` gets down to the value of `Area size` (not the `Max area size`). This tells us that there is a special case formula for predicting in-memory sorts, but it also hints that the optimizer is behaving as if the various overheads and I/O buffers fall outside the memory allocation for sorting.

Note also that the sort width is 58, which is what you would get with a `sort_area_size` of around 2MB using the manual `workarea_size_policy`. With a `sort_area_size` of 204KB (or rather 320KB, which would report an area size of about 204KB), the sort width would be 9 and the initial run count would be 86—leading to three merge passes. With a `sort_area_size` of 10MB (or rather 15.1MB, which would report an area size of about 10MB), the sort width would be 426 with a run count of two.

Despite our attempts to pick a `pga_aggregate_target` that could emulate a `sort_area_size` of 10MB, we cannot see figures that allow us to align the two parameters in an obvious way.

To cap it all, whatever we see in the `10053 trace`, remember that the original `10032 trace` for this example reported a `sort_multiblock_read_count` of 31 and a `max intermediate merge width` of 5. The optimizer assumptions do not form an accurate model of the actual run-time activity.

Clearly it is not going to be easy to reverse engineer the algorithm that the cost based optimizer uses to derive a cost for a sort. So before trying to untangle the knotty problem of how the optimizer works out the `IO cost per pass`, I'd like to run a few more test cases and comment on the results.

Comparisons

The scripts I have used to compare the costs for sorting in the four different environments are very similar to the scripts I used in Chapter 12. I have a pair of scripts in the online code suite called `pat_dump.sql` and `sas_dump.sql` that run the following query after setting the `pga_aggregate_target` or `sort_area_size` respectively:

```
select
        sortcode
from
        (select * from t1 where rownum <= 300000)
order by
        sortcode
;
```

These scripts are driven by four other scripts—two for `pat_dump.sql`, and two for `sas_dump.sql`. The calling scripts are `pat_cpu_harness.sql`, `pat_nocpu_harness.sql`, `sas_cpu_harness.sql`, `sas_nocpu_harness.sql`. As you can infer from the names, two of the scripts enable CPU costing, two of them don't.

The scripts that enable CPU costing use the following PL/SQL to mimic the normal arithmetic of a system that is not using CPU costing when the `db_file_multilblock_read_count` is set to eight blocks:

```
begin
        dbms_stats.set_system_stats('MBRC',6.59);
        dbms_stats.set_system_stats('MREADTIM',10.001);
        dbms_stats.set_system_stats('SREADTIM',10.000);
        dbms_stats.set_system_stats('CPUSPEED',1000);
end;
/
```

To draw some valid comparisons between traditional costing based on the `sort_area_size` and costings based on the `pga_aggregate_target`, I have assumed that the 5% rule is appropriate—for example, aligning a 1MB `sort_area_size` with a 20MB `pga_aggregate_target`.

The basic model that appeared in all the `10053 trace` files was

- *With CPU costing disabled*: `Total IO sort cost` = (`Blocks to sort` + `IO cost per pass` * `Merge passes`) / 2
- *With CPU costing enabled*: `Total IO sort cost` = (`Blocks to sort` + `IO cost per pass` * `Merge passes`)

Since the `Blocks to sort` was a constant (588) throughout the tests, the total cost of the sort is easy to grasp if we can understand what's going on with the `IO cost per pass`.

Table 13-4 reproduces some of the interesting results; the four figures in each column are `IO cost per pass` / `Initial runs` / `Merge passes`, and then the `Total IO sort cost`.

Table 13-4. *Comparing Costs in Different Environments*

Target Sort Memory	S_A_S No CPU Costing	S_A_S CPU Costing	P_A_T Assuming 5% Limit No CPU Costing	P_A_T Assuming 5% Limit CPU Costing
64KB	784 / 131 / 5 =>2,254	148 / 131/ 5 => 1,328	n/a	n/a
128KB	784 / 56 / 4 => 1,862	148 / 56 / 4 => 1,180	n/a	n/a
256KB	756 / 27 / 2 => 1,050	148 / 27 / 2 => 884	n/a	n/a

Table 13-4. *Comparing Costs in Different Environments*

Target Sort Memory	S_A_S No CPU Costing	S_A_S CPU Costing	P_A_T Assuming 5% Limit No CPU Costing	P_A_T Assuming 5% Limit CPU Costing
320KB	720 / 22 / 2 => 1,014	148 / 22 / 2 => 884	n/a	n/a
384KB	696 / 18 / 2 => 990	148 / 18 / 2 => 884	n/a	n/a
448KB	726 / 16 / 2 => 1,020	148 / 16 / 2 => 884	n/a	n/a
512KB	714 / 14 / 1 => 651	148 / 14 / 1 => 736	882 / 10 / 4 => 2,058	148 / 10 / 4 => 1,180
640KB	687 / 11 / 1 => 638	148 / 11/ 1 => 736	882 / 8 / 3 => 1,617	148 / 8 / 3 => 1,032
1,024KB	693 / 8 / 1 => 643	148 / 8 / 1 => 736	735 / 5 / 2 => 1,029	148 / 5 / 2 => 884
1,536KB	686 / 5 / 1 => 637	148 / 5 / 1 => 736	756 / 4 / 1 => 672	148 / 4 / 1 => 736
2,048KB	687 / 4 / 1 => 638	148 / 4 / 1 => 736	706 / 3 / 1 => 647	148 / 3 / 1 => 736
3,072KB	686 / 3 / 1 => 637	148 / 3 / 1 => 736	699 / 2 / 1 => 644	148 / 2 / 1 => 736
4,096KB	696 / 2 / 1 => 642	148 / 3 / 1 => 736	696 / 2 / 1 => 642	148 / 2 / 1 => 736
6,976KB	678 / 1 / 1 => 633	0 / 1 / 0 => 0	700 / 2 / 1 => 644	148 / 2 / 1 => 736
10,240KB	720 / 1 / 1 => 654	0 / 1 / 0 => 0	687 / 2 / 1 => 638	148 / 2 / 1 => 736
15,360KB	686 / 1 / 1 => 637	0 / 1 / 0 => 0	686 / 2 / 1 => 637	148 / 2 / 1 => 736
20,480KB	764 / 1 / 1 => 676	0 / 1 / 0 => 0	683 / 2 / 1 => 636	148 / 2 / 1 => 736
30,720KB	692 / 1 / 1 => 640	0 / 1 / 0 => 0	700 / 2 / 1 => 644	148 / 2 / 1 => 736

There are a number of interesting anomalies here, and, as we saw with the costing of hash joins, the changes in cost as you try to change your strategy are probably more important than the exact details of how specific costs are calculated.

The first point to pick out is that when you try to convert from using the `sort_area_size` to the `pga_aggregate_target`, the cost figures don't align very well if you work with the 5% figure that is quoted as the limiting workarea size. In fact, if you cross-check the first few costs for this sort in columns 2 and 4 (the two "No CPU Costing" columns), you see that the figures for `pga_aggregate_target` might be a better match for the `sort_area_size` if you assumed a 1% limit, rather than the 5% limit.

The next point to pick out is that if you are currently running with manual work areas, and decide to try enabling CPU costing without switching to automatic work areas, the cost of some sorts will drop dramatically. In fact, this behavior looks to me like a correction in the costing model—the drop to zero cost occurs at about the point in the model where the sort will complete entirely in memory. (It is unfortunate that the model doesn't necessarily give a true image of the memory required—but at least the model becomes self-consistent if this drop does occur.)

The third point about these figures shows up in the larger memory sizes—I had about 2MB of data to sort (300,000 rows of 6 bytes each if you ignore the overheads introduced by the sorting), which turns into about 7MB if you allow for all the extra bytes and pointers. But even when the allowed memory was 30MB, the optimizer assigned a large cost to a sort that I knew would complete in memory. There are numerous little oddities with three of the four models

that result in sorts that will become in-memory sorts at run time being given a cost that seems to be unreasonably high.

There are a couple of extra points, relevant to the CPU costing options, that aren't visible in this table. When you change the MBRC (achieved multiblock read count), the cost of the sort changes—the cost of the IO is affected by the assumed size of the IO requests, though not in a completely linear fashion as far as I could tell. When you change the mreadtim (time for a typical multiblock read in milliseconds), the cost of the sort changes—and in the test case, I had minimized the mreadtim, which means that the costs of sorts (especially the in-memory sorts) will be even higher when you have realistic times in your system.

There are also a couple of special cases—boundary conditions—on the system statistics, which make the optimizer change strategies. For example, if mreadtim / sreadtim is greater than MBRC, then the optimizer changes the formula for calculating the cost of the sort. In one test case, I got a cost of 5,236 for a sort when my mreadtim was 65 milliseconds, and a cost of 2,058 when I changed the mreadtim to 66 milliseconds—and I don't think you would expect the cost of a sort to drop sharply when the disk response time has slowed down. (And this behavior is not consistent with the way that a tablescan cost is calculated—in that case, an over-large mreadtim is not treated as an anomaly.)

Another anomaly that showed up was the problem of getting the optimizer to cost correctly for an in-memory sort when using the automatic workarea_size_policy and pga_aggregate_target. The following extract comes from the 10053 trace file when I ran my test case with my pga_aggregate_target set to 2GB:

```
ORDER BY sort
    SORT resource      Sort statistics
      Sort width:     598 Area size:  1048576 Max Area size: 104857600   Degree: 1
      Blocks to Sort:      588 Row size:           16 Rows:     300000
      Initial runs:          2 Merge passes:        1 IO Cost / pass:        148
      Total IO sort cost: 736
      Total CPU sort cost: 270405074
      Total Temp space used: 7250000
```

Notice how the Area size: is set to just 1MB. This seems to be a hard limit set by parameter _smm_min_size—even though the run-time engine will happily take much more memory. But it is this limit that introduces a boundary condition on the costing of the sort—so the optimizer will not cost this sort as an in-memory sort as it needs a minimum 300,000 * 16 bytes = 4.6MB, which is larger than the _smm_min_size.

In this example, I had to restrict the number of rows in my query to just 65,536 before the optimizer would cost the sort as an in-memory sort (at which point the IO cost per sort dropped to zero). The fact that the I/O component of the sort cost *can* drop to zero for this configuration of pga_aggregate_target and CPU costing is, of course, a good thing because it is reasonably representative of real life. This is definitely a plus point for CPU costing if you are running an OLTP system, as you are likely to find that SQL operating in the scale of the "small report" is probably going to fall into the range of calculated and actual optimal sorting. The fact that the I/O component of cost drops to 0 only when the estimated volume of data falls below such an extreme limit is *not* such a good thing—and may encourage people to start tweaking hidden parameters in the hope of making the optimizer behave a little more sensibly.

I believe that the strategic option is to enable CPU costing, and take advantage of the automatic workarea_size_policy. But I do think you are likely to have problems in transition, and

need to do a thorough regression test. As a starting guess for the `pga_aggregate_target`, the best figure for minimal changes to current execution plans may be about 100 times your current setting of `sort_area_size` (rather than the 20 times implied by the known 5% limit that applies to run-time sizing of work areas). I would, however, advise you to consider using session-based code to switch to manual work area sizing and explicit values for the `sort_area_size` for critical batch-like tasks.

In many cases, the changes in costs for sorting are likely to leave execution plans unchanged because most uses of sorting are due to `order by`, `group by`, or `distinct`, which have to be done whatever the cost (though watch out for the hashing options for `group by` and `distinct` that 10*g*R2 brings). There will be cases though where paths may change dramatically, perhaps swapping from merge to hash or hash to merge, because of a sharp (and unreasonable) change in cost. You may also find cases where **subquery unnesting**, or **complex view merging**, suddenly does (or does not) happen the way it used to because of a change in the cost of a `distinct` requirement that wasn't even directly visible in your SQL.

Merge Joins

Finally, we can address the mechanism and cost of the merge join—usually called the *sort/merge join* because it requires both its input to be sorted on the join columns.

The concept is very simple and, like the hash join, starts by decomposing the join into two independent queries, before combining the separate results sets:

- Acquire the first data set, using any access and filter predicates that may be relevant, and sort it on the join columns.
- Acquire the second data set, using any access and filter predicates that may be relevant, and sort it on the join columns.
- For each row in the first data set, find the start point in the second data set, and scan through it until you hit a row that doesn't join (at which point you ought to know that you can stop because the second data set is in sorted order).

In Chapter 12, I pointed out that a hash join was basically a back-to-front nested loop into a single-table hash cluster. Read the description of a merge join again, and you realize that it's nearly a nested loop join from an index organized table into an index organized table (although the sorted data sets aren't actually indexed, and Oracle may be using a binary chop to find the start point for each pass through the second data set). Just like the hash join, the merge join breaks a single query into two separate queries that initially operate to acquire data; unlike the hash join, the first query need not complete before the second query starts.

The Merge Mechanism

Depending on your point of view, you could decide that the number of variants for the mechanics of a merge join is anything between one and five. Having decided on the number, you can then double it to represent the cases where the first data set can be acquired in presorted order—for example, because of a convenient index—and need not be fully instantiated before the second set is acquired. (Script `no_sort.sql` in the online code suite is an example that demonstrates how you can prove this point.)

If you take the view that there are many variations to consider, you would probably come up with a list like the following:

- One to one
- One to many one on an equality (e.g., parent/child join)
- One to many on a range (e.g., `t2.dt between t1.dt - 3 and t1.dt + 3`)
- Many to many on an equality
- Many to many on a range

Pictorially, you could represent the options as shown in Figure 13-2, once the data sets had been sorted.

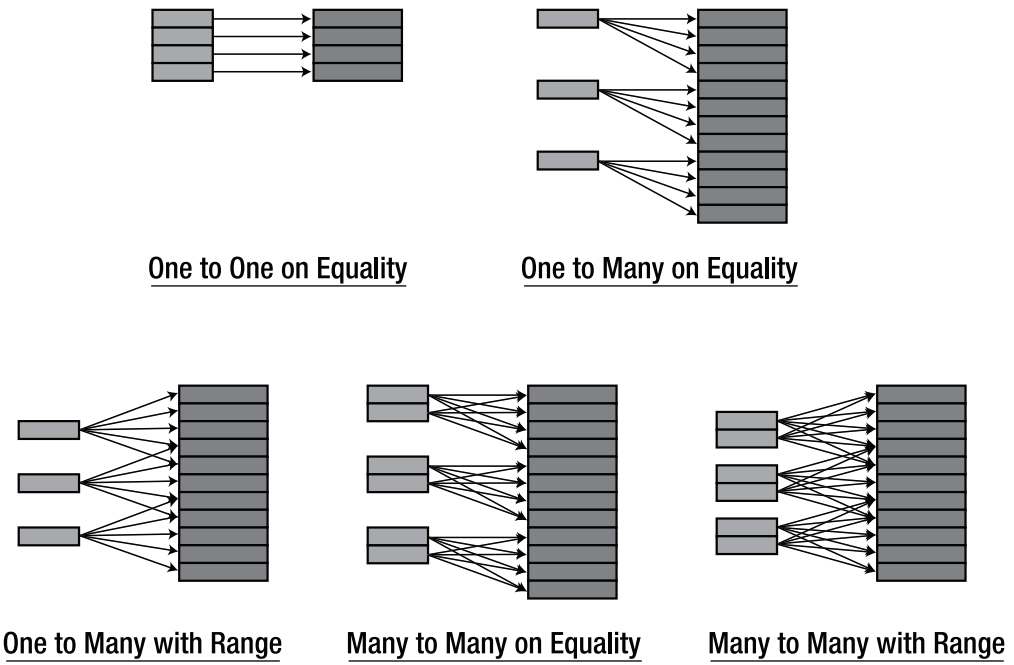

Figure 13-2. *Variations on merge joins*

But if you wanted to view the merge join as a single possible strategy, you could argue that the mechanism is always the same—for each row in the first input, find the first legal join row from the second input, and walk through the second input in order until you just overshoot the last legal join row.

When you look at these pictures, it is fairly obvious that the first two samples are somehow simpler than the last three—and the last three aren't really very different from each other. In the first two examples, the run-time engine can simply walk down the two stacks and keep going in order, never stepping backwards. In the more complex examples, the run-time engine

has to be able to step backwards in the second data stack each time it moves forwards a row in the first data stack.

One side effect of this is that the merge join is not completely symmetrical—despite comments in the manual, the second data is always fully acquired, sorted, and instantiated; it cannot be acquired piece-wise (for example, by walking an index) like the first data set. Given the increasing complexity of the work that has to be done, especially in the last three joins, we might expect the cost of the merge join to vary with the type.

Script `merge_samples.sql` in the online code suite demonstrates queries that match the pictures, and produces the execution plan results:

```
alter session set hash_join_enabled = false;

create table t1
as
with generator as (
        select  --+ materialize
                rownum         id
        from    all_objects
        where   rownum <= 3000
)
select
        rownum                  id,
        rownum                  n1,
        trunc((rownum - 1)/2)   n2,
        lpad(rownum,10,'0')     small_vc,
        rpad('x',100,'x')       padding
from
        generator        v1,
        generator        v2
where
        rownum <= 10000
;

alter table t1 add constraint t1_pk primary key(id);

--        Create table t2 in the same fashion
--        Collect statistics using dbms_stats here

--        The following statement has different predicates,
--        corresponding to the five pictures in the diagram

select
        count(distinct t1_vc ||t2_vc)
from
        (
        select /*+ no_merge ordered use_merge(t2) */
                t1.small_vc        t1_vc,
                t2.small_vc        t2_vc
```

```
        from
                t1,
                t2
        where
                t1.n1 <= 1000 and t2.id = t1.id
        --        t1.n1 <= 1000 and t2.n2 = t1.id
        --        t1.n1 <= 1000 and t2.id between t1.id - 1 and t1.id + 1
        --        t1.n1 <= 1000 and t2.n2 = t1.n2
        --        t1.n1 <= 1000 and t2.n2 between t1.n2 - 1 and t1.n2 + 1
        )
;
```

TEMPUS FUGIT

When testing 9*i* and 8*i*, I set `hash_join_enabled` to `false` to stop the optimizer from choosing a hash join on the simple equality queries. Unfortunately, 10*g* ignored this setting—so I had to include the hints `ordered` and `use_merge(t2)` to force the optimizer to use a merge join.

Things keep changing, and it's a real nuisance how bits of code suddenly become obsolete and stop working. Watch out, for example, for the parameter `_optimizer_sortmerge_join_enabled`, introduced in 9*i*, one day it may silently default to `false`—and all your merge joins will change to nested loop or hash joins!

The surprising thing about the five different execution plans is that the final cost is the same for all of them, even though there are slight variations in the plans and distinct changes in the cardinality of the joins. For example:

```
Execution Plan (9.2.0.6 autotrace, no system statistics, 1 to 1 with equality)
----------------------------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=105 Card=1 Bytes=14)
1    0   SORT (GROUP BY)
2    1     VIEW (Cost=105 Card=1000 Bytes=14000)
3    2       MERGE JOIN (Cost=105 Card=1000 Bytes=34000)
4    3         SORT (JOIN) (Cost=38 Card=1000 Bytes=19000)
5    4           TABLE ACCESS (FULL) OF 'T1' (Cost=29 Card=1000 Bytes=19000)
6    3         SORT (JOIN) (Cost=68 Card=10000 Bytes=150000)
7    6           TABLE ACCESS (FULL) OF 'T2' (Cost=29 Card=10000 Bytes=150000)

Execution Plan (9.2.0.6 autotrace, no system statistics, many to many with range)
----------------------------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=105 Card=1 Bytes=14)
1    0   SORT (GROUP BY)
2    1     VIEW (Cost=105 Card=25002 Bytes=350028)
3    2       MERGE JOIN (Cost=105 Card=25002 Bytes=850068)
4    3         SORT (JOIN) (Cost=38 Card=1000 Bytes=19000)
5    4           TABLE ACCESS (FULL) OF 'T1' (Cost=29 Card=1000 Bytes=19000)
6    3         FILTER
```

```
7    6          SORT (JOIN)
8    7            TABLE ACCESS (FULL) OF 'T2' (Cost=29 Card=10000 Bytes=150000)
```

Even though the cardinality of the join (see line 3) changes from an estimated 1,000 to an estimated 25,002, the cost of the query stays constant at 105. And that cost is essentially the cost of producing two sorted sets of data—lines 4 and 6 in the first plan: 38 + 68 = 106 (the difference between the sum and the total in the execution plan appears because `explain plan` reports every step to the nearest whole number, but the arithmetic carries forward with exact fractions retained).

Chapter 10 had an example of a merge join with a range test to demonstrate some of the oddities in the calculations for join cardinality. At the time I postponed any explanation of an odd `filter` line that appeared in the execution plan. The same filter line has appeared in the preceding plan, and now is the time to explain it.

If you switch from `autotrace` to `dbms_xplan.display()` so that you can pick up the access predicates and filter predicates, the predicates that go with the preceding plan—identified by line number—are

```
5 - filter("T1"."N1"<=1000)
6 - filter("T2"."N2"<="T1"."N2"+1)
7 - access("T2"."N2">="T1"."N2"-1)
    filter("T2"."N2">="T1"."N2"-1)
```

You can also run the query then examine the view `v$sql_plan_statistics` to find out how many times Oracle executed lines 6, 7, and 8, and the number of rows supplied from each row source.

The filter operation (line 6) was executed 1,000 times to produce 5,996 rows (which is reasonable—apart from the very first and last rows in the `t1` selection, I was expecting six rows in `t2` for each row in `t1`).

The table access full of `t2` (line 8) was executed once to produce 10,000 lines (which is reasonable—we only scan the table once and get all the rows from it because there are no simple restrictions).

The strange line is line 7—the sort line—which (according to `v$sql_plan_statistics`) gets executed 1,000 times (effectively once for each row in table `t1`) and returns 9,502,996 rows!

Clearly something is not quite right in the way the execution plan is reporting to us. We have not done 1,000 separate sorts, we did one sort to generate the data—at least, that's what the `10032 trace` file tells us. The problem appears because we have done 1,000 separate accesses into the sorted data, and our `between` clause has been turned into two separate predicates.

The access predicate in line 7 (`"T2"."N2">="T1"."N2"-1`) has told Oracle where to start in the sorted data set; but the filter predicate in the same line has made Oracle check every row to ensure that the predicate (`"T2"."N2">="T1"."N2"-1`) is still true—despite the fact that it has to be because the data set is sorted. For every row in `t1`, Oracle has located the first relevant row in the second sorted data set, and scanned from there *to the end* of the sorted data set. That's 10,000 rows checked for the first row in `t1`; 9,998 rows checked for the second row in `t1`; 9,996 rows checked for the third row in `t1`; and so on—for a total of more than 9.5 million rows.

At line 6, the line with the `filter` operation, Oracle has then used the predicate (`"T2"."N2"<="T1"."N2"+1`) to discard all but six rows for each row from the `t1` data set. It looks as if the optimizer has turned the original `between` predicate into two separate predicates, and

then forgotten the special meaning of those two predicates (just as it once did when calculating the cost of a simple `between` clause on a single table).

The optimizer should be able to use the predicate (`"T2"."N2">="T1"."N2"-1`) as the access predicate to start ranging through the second data set, and the predicate (`"T2"."N2"<="T1"."N2"+1`) as the filter predicate in the same line. This predicate should be tested against each row during the ranging step, and the ranging process should stop the first time the test fails—not run to the end of entire data set. Just to complete the picture (although it's a complete mess and virtually illegible), Figure 13-3 compares what Oracle *should be* doing with what Oracle *is* doing in this case:

Figure 13-3. *Range-based merge join*

The CPU penalty for this error can be enormous, and in this example, it was actually cheaper to run the query as a nested loop with 1,000 tablescans of `t2` rather than doing the merge join, which should have been more efficient.

A Merge Join Without the First Sort

As I have already pointed out, the manuals do suggest that if both data sets can be acquired in order, then neither set has to be sorted. In fact, the `10053 trace` shows that the only "no-sort" option considered by the optimizer is for the outer (first) table. This results in two costing sections for the merge join, which we will examine next.

One of the examples from script `merge_samples.sql` in the online code suite is able to acquire the data in the first table by a rather expensive indexed access path, so produces the following trace when costing the join:

```
SM Join
  Outer table:
    resc: 29  cdn: 9001  rcz: 15  deg: 1  resp: 29
  Inner table: T1
    resc: 29  cdn: 9001  rcz: 15  deg:  1  resp: 29
    using join:1 distribution:2 #groups:1

    SORT resource      Sort statistics
                -- Outer table costing
```

```
    SORT resource      Sort statistics
                -- Inner table costing
  Merge join  Cost:  127  Resp:  127

SM Join (with index on outer)
  Access path: index (scan)                -- identifies index path used
      Index: T2_PK
  TABLE: T2
      RSC_CPU: 0   RSC_IO: 182
  IX_SEL:  9.0009e-001  TB_SEL:  9.0009e-001
  Outer table:
    resc: 182  cdn: 9001  rcz: 15  deg: 1  resp: 182
  Inner table: T1
    resc: 29  cdn: 9001  rcz: 15  deg:  1  resp: 29
    using join:1 distribution:2 #groups:

    SORT resource      Sort statistics
                -- Inner table costing only

  Merge join  Cost:  246  Resp:  246
```

Note that the second `SM Join` calculation will only appear if the indexed access path had not already been selected as the cheapest way of acquiring the first data set anyway—modify the example to select only 1,000 rows rather than 9,000 rows from the first table, and the no-sort option is selected automatically—and you will notice that the execution plan then shows a sort line only for the second table.

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=53 Card=1 Bytes=14)
1  0  SORT (GROUP BY)
2  1    VIEW (Cost=53 Card=999 Bytes=13986)
3  2      MERGE JOIN (Cost=53 Card=999 Bytes=29970)
4  3        TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=22 Card=1000 Bytes=15000)
5  4          INDEX (RANGE SCAN) OF 'T1_PK' (UNIQUE) (Cost=4 Card=1000)
6  3        SORT (JOIN) (Cost=31 Card=1000 Bytes=15000)
7  6          TABLE ACCESS (BY INDEX ROWID) OF 'T2' (Cost=22 Card=1000 Bytes=15000)
8  7            INDEX (RANGE SCAN) OF 'T2_PK' (UNIQUE) (Cost=4 Card=1)
```

Even when the optimizer selects a no-sort merge, you will find that the cost of the merge join is still (very close to) the sum of the two separate accesses to the table. In this case, you can see that the cost at line 3 is the sum of the costs at lines 4 and 6 (53 = 22 + 31).

The Cartesian Merge Join

In Chapter 6, we came across an example where the optimizer used **transitive closure** to turn a perfectly reasonable join into a Cartesian merge join. And I mentioned at the time that the cost of the join was a little surprising. It's time to take a look at what has happened in this case (see script `cartesian.sql` in the online code suite):

```
select
        count(t1.small_vc),
        count(t2.small_vc)
from
        t1, t2
where
        t1.n1 = 5
and     t2.n1 = 5
;
```

Table t1 has 800 rows, and table t2 has 1,000 rows. In both tables, one-tenth of the rows have n1 = 5. This is the execution plan for the preceding query—taken from view v$sql_plan_statistics after using the alter session command to set the parameter _rowsource_execution_statistics to true. (You can get the same view populated by setting statistics_level to all, or sql_trace to true. The hidden parameter is just more precisely targeted—but obviously should not be used on production systems.)

```
Id Starts   Rows Plan (9.2.0.6 - v$sql_plan_statistics)
-- ------ ------ -------------------------------------------------
 0              SELECT STATEMENT (all_rows) Cost (324,,)
 1      1      1   SORT   (aggregate)
 2      1  8,000     MERGE JOIN (cartesian) Cost (324,8000,112000)
 3      1     80       TABLE ACCESS (analyzed) T1 (full)  Cost (4,80,560)
 4     80  8,000       BUFFER  (sort)  Cost (320,100,700)
 5      1    100         TABLE ACCESS T2 (full)  Cost (4,100,700)
```

The three figures in brackets at the end of each line are the (cost, cardinality, and bytes) respectively—and as you can see, the cost of the merge join (line 2) is 324, which seems to be the cost of the table scan in line 3 plus the cost of the buffer (sort) in line 4. But the cost of the buffer sort seems to be the cost of the table scan in line 5 multiplied by the cardinality from line 3—in effect, the costing for the Cartesian merge join seems to be very similar to the costing for a nested loop, namely *cost of first table + cardinality of first table * cost of second table*.

The second and third columns show the last_starts and last_output_rows figures from the dynamic performance view v$sql_plan_statistics: these appear to be telling us that the buffer (sort) happened 80 times (once for each row acquired from table t1), and that 8,000 rows were actually sorted. This is exactly the same behavior as we saw with the more normal many-to-many range-based merge join—and is just as unbelievable. We may have visited the sorted data 80 times, but did we really sort it 80 times? I hope not.

SORT AND BUFFER (SORT)

I haven't yet worked out if there is any significant difference between these two options—the optimizer doesn't seem to use volume of data as the basis for choosing one over the other. The buffer (sort) that appears in this Cartesian merge join used to be the standard sort operation when the test case ran under 8*i*, and no other changes seem to have appeared at the same time.

Whatever misdirection we are receiving from the optimizer with the plan statistics, the enormous cost of this query is clearly an error. Oracle has extracted a tiny amount of data from the second table, packed it into memory, and then assigned the cost of the full tablescan to that subset whenever it thinks it is going to use the data. We can see the source of this anomaly when we look at the `10053 trace` file. The critical section looks like this:

```
Join order[1]:  T1[T1]#0  T2[T2]#1
Now joining: T2[T2]#1 *******
NL Join
  Outer table: cost: 4  cdn: 80  rcz: 7  resp:  4
  Inner table: T2
    Access path: tsc  Resc: 4
    Join:  Resc:  324  Resp:  324
  Best NL cost: 324  resp: 324
Join cardinality:  8000 = outer (80) * inner (100) * sel (1.0000e+000)  [flag=0]
Join result: cost: 324  cdn: 8000  rcz: 14
Best so far: TABLE#: 0  CST:          4  CDN:         80  BYTES:        560
Best so far: TABLE#: 1  CST:        324  CDN:       8000  BYTES:     112000
```

Look carefully at this extract—I haven't deleted any lines from it. The optimizer trace shows nothing but the calculation for the nested loop join. I said that the cost of the merge join seemed to be very similar to the costing of a nested loop, and it's true—the optimizer is using the nested loop cost as the cost of the merge join.

The upshot of this mechanism is that Cartesian merge joins are going to be given a very high cost—which is a pity, really, because sometimes a Cartesian join will be a quick, efficient starting point for a complex query, and the optimizer might choose to ignore it. So this is one case where you may have to hint the SQL.

Strangely, this example is very similar to our very CPU-intensive earlier query that actually examined 9.5 million rows. The difference is that this example has become a Cartesian join because the join predicate disappeared under transitive closure. The earlier query did not lose it's joins predicate though transitive closure—so the arithmetic was completely different. In one case we have a cost that is unreasonably high for a query that won't do a huge amount of work; in the other we have a cost that is small for a query that will burn up a massive amount of CPU.

Aggregates and Others

After looking at the effects of basic sorting and the merge join, it's time say something about other reasons that Oracle might have for sorting. For example, how should the costs of the following queries compare (see script `agg_sort.sql` in the online code suite)?

```
select  col1, col2           from t1 order by col1, col2;
select  distinct col1, col2  from t1;
select  col1, col2, count(*) from t1 group by col1, col2;
select  col1, col2, count(*) from t1 group by col1, col2 order by col2, col1;
select  max(col1), max(col2) from t1;
```

A typical execution plan for any of these five queries would have the following structure:

```
Execution Plan (9.2.0.6 autotrace, no system statistics)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=18 Card=5000 Bytes=20000)
   1    0   SORT (ORDER BY) (Cost=18 Card=5000 Bytes=20000)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=4 Card=5000 Bytes=20000)
```

The only significant change as you move from query to query is in line 1 of the plan, and the differences can be summarized as shown in Table 13-5.

Table 13-5. *Sample Costs for Different Sorting Requirements*

Requirement	Optimizer Option	Final Cost
`order by`	`sort (order by)`	18
`distinct`	`sort (unique)`	18
`group by`	`sort (group by)`	18
`group by ... order by`	`sort (group by)`	18
`max()`	`sort (aggregate)`	3

There are two anomalies to note in particular. The obvious one is that the final cost for the `max()` query is much less than the final cost for all the other queries, which all have the same cost. (In fact, when you enable CPU costing, you find that the other four queries also have exactly the same CPU cost, despite the obviously different requirements for processing.)

In theory, you might expect Oracle to sort the data as the first step of any of these operations. However, the `max()` example can be optimized without sorting—despite the appearance of the `sort` line in the execution plan. The run-time code simply has to scan the incoming data and keep track of the largest value so far. The optimizer calculations reflect this operation exactly.

If we look at the other operations, it is clear that the `order by` will need a sort. Similarly, the `distinct` could be implemented by sorting the data, scanning the sorted result set, and reporting a row only when it differed from the previous row. The optimizer doesn't bother trying to put a figure on the tiny CPU difference involved in the last step of this operation. The same approach applies to the `group by`, except we maintain running totals (in this case a count) and pass forward the totals along with the row data whenever the row differs from the previous row.

The second anomaly shows up when you examine the query that does the `group by` followed by the `order by`—look carefully at the code, and you will see that my `group by` clause is on `col1`, `col2`, but my order by clause is on `col2`, `col1`—the opposite way around.

It would be very easy to assume that Oracle would have to sort the data once to do the `group by`, then sort it again to do the `order by`. However, the optimizer is smart enough to determine that both the `group by` and the `order by` could be satisfied by a single sort operation.

THE SQL MUST USE ORDER BY

We have been told endlessly that we should always include an `order by` clause if we want the data in a specific order, and not rely on an accidental sort performed by the `group by`. Although this is a dictate that we ought to follow unswervingly, that doesn't mean the optimizer actually has to perform both operations if it can find a cheaper alternative.

There is an interesting follow-up to this point, though. In 9*i*, Oracle introduced a new algorithm for multicolumn aggregation. Script `gby_onekey.sql` in the online code suite demonstrates the principle. I have generated a small random data set, and queried it with the following SQL statement:

```
select
        col1, col2, count(*)
from
        t1
group by
        col1, col2
;
```

The output looks like this:

```
COL1 COL2   COUNT(*)
---- ---- ----------
A    Y             1
B    V             1
B    Y             1
Y    TO            1
Z    DU            1
...
AD   K             1
CC   K             1
CE   DZ            1
...
```

The results are not in order—a very clear, and simple, demonstration that you should never trust a `group by` to do the implicit `order by` that you want. This little feature is controlled by the hidden parameter `_gby_onekey_enabled`; set this to `false` and Oracle falls back to its old behavior, which happens to produce the data in the "correct" order.

"GROUP BY" BUG

There is a slightly exotic bug relating to this new grouping mechanism that appears in some special cases of sort/merge joins. MetaLink bug 3487660 applies. Essentially, there is a case where the optimizer assumes that a `group by` implies an `order by`, and fails to do a sort on the first input to a merge join when it really needs to do one.

So if 9*i* has acquired a different mechanism for dealing with the `group by`, what's going to happen when we change this query to include an `order by` immediately after the `group by`? Will there be a second sort to get the data in the right order after the grouping has completed?

If we look at the execution plan, or the `10053 trace`, there is no difference between what we see from 8*i* and 9*i*; the plan still claims that it's going to do nothing but a `sort (group by)`, so the answer appears to be no—but we know that the run-time engine doesn't always do what the optimizer says it's going to. If we look at the session statistics, we find that they record one sort, with exactly the right number of rows sorted—which seems to be a reasonable confirmation of the execution plan. But how can Oracle manage to get the job done with just one sort?

The answer comes from part of the `10032 trace`. Compare the two sets of statistics—the first set comes from the query without the `order by`, the second set comes from the query with the `order by`:

```
---- Sort Statistics ------------------------
Input records                             50
Output records                            50
Total number of comparisons performed     245
  Comparisons performed by in-memory sort 245

---- Sort Statistics ------------------------
Input records                             50
Output records                            50
Total number of comparisons performed     222
  Comparisons performed by in-memory sort 222
```

There really is just one sort, but the number of comparisons performed changes when I include the `order by`. If you repeat the tests with the new feature disabled, you will find that that Oracle is simply falling back to the original grouping mechanism when the `order by` clause appears, so that it doesn't have to do an extra sort to get the data in order.

A NEW GROUP BY IN 10*G*R2

One of the new features in 10gR2 is a `hash group by` operation (with a corresponding `hash unique` operation), which in principle could be much faster than a `sort group by` in many cases. It would be wise to check your code to see if you still have any SQL where you are depending on the implicit sorting effect of a `group by` to return the data in order.

It will be interesting to see if it's possible to find a data set in 10*g*R2 where the optimizer decides to use a `hash group by` followed by a `sort order by`—or whether a combined `group by/order by` will always be turned into a single `sort group by` when the grouping and ordering columns are the same.

Another change appeared in the upgrade from 8*i* to 9*i* in the underlying optimizer calculations for the `group by` and `distinct` operations. Here's a query from the `script agg_sort_2.sql` in the online code suite, and a section of the 9.2.0.6 version of the `10053 trace` file for that query (the `10053 trace` file for a simple `select distinct` is very similar):

```
select
        col1, col2, count(*)
from
        t1
group by
        col1, col2
;

SINGLE TABLE ACCESS PATH
  TABLE: T1     ORIG CDN: 5000  ROUNDED CDN: 5000  CMPTD CDN: 5000
  Access path: tsc  Resc:  3  Resp:  3
  BEST_CST: 3.00  PATH: 2  Degree:  1
Grouping column cardinality [      COL1]    25
Grouping column cardinality [      COL2]    71
*************************************
GENERAL PLANS
***********************
Join order[1]:  T1[T1]#0
GROUP BY sort
GROUP BY cardinality:  1256, TABLE cardinality:  5000
    SORT resource      Sort statistics
      Sort width:      58 Area size:   208896   Max Area size:  10485760  Degree: 1
      Blocks to Sort:       10 Row size:           15 Rows:       5000
      Initial runs:          1 Merge passes:        1 IO Cost / pass:         19
      Total IO sort cost: 11
      Total CPU sort cost: 0
      Total Temp space used: 0
Best so far: TABLE#: 0  CST:         14  CDN:       5000  BYTES:      20000
    Total Temp space used: 0
```

The first thing to note is the `GROUP BY cardinality` information. The optimizer has picked up the `num_distinct` values for `col1` and `col2` separately in the single-table access details, and used them to produce a grouping cardinality in the general plans section. But how do you get 1,256 from 25 and 71? Answer—multiply them together and divide by the square root of 2 (1,256 = 24 * 71 / 1.4142). And in general, the optimizer estimates the number of distinct combinations of `N` columns by multiplying the individual `num_distinct` values, and then dividing by the square root of 2 `(N – 1)` times.

As a sanity check, the optimizer compares this with the number of rows in the table, and reports the lower of the two values as the cardinality of the result set.

In 9*i* (and with the default value of `_new_sort_cost_estimate` unchanged), this has a knock-on effect on the cost. Earlier on in the chapter, I pointed out that you will often see that the `Total IO sort cost` is `(Blocks to Sort + IO Cost per pass) / 2`: in this example this would be (10 + 19) / 2, which would be reported as 15. But the `Total IO sort cost` is reported as 11. Something has changed. (In 8*i*, and allowing for the small changes due to rounding errors, etc., you would see the equivalent of 15 as the total I/O sort cost.)

The difference is interesting—from 9*i* onwards, the cost has been adjusted to allow for the fact that the number of rows returned by the query is less than the total number of rows in the

table. It seems that in general, the `IO cost per pass` is *total volume of input data* + *total volume of output data*. The value 19 is

```
Input in blocks + output in blocks =
Input Rows * Row size / blocksize  + output Rows * Row size / blocksize =
5,000 * 15 / 8192 + 5,000 * 15 / 8192 =
18.3105                                        -- which rounds up to 19.
```

But in our grouping case, the optimizer has decided that the grouping cardinality is 1,256, so the result changes:

```
5,000 * 15 / 8192 + 1,256 * 15 / 8192 =
11.455         -                              -- which rounds up to 12
```

Put that 12 into the normal formula (for a one merge pass sort):

```
(blocks to sort + IO Cost per pass) / 2 =
(10 + 12) / 2 =
11
```

So in this case, the `10053 trace` is showing us one figure for the `IO Cost per pass` (19) and using another (12) to calculate the `Total IO sort cost`. It seems the trace file has not caught up with the new feature.

There is a trap to watch out for here if you are still waiting to upgrade. If you have complex queries that use aggregate functions to crunch large numbers of rows down to small result sets, then this particular cost change may make a significant difference to the cost of some parts of those queries—which may change the overall execution plan. This simple example showed a change on a single table aggregation from 18 down to 14 (because the grouping cost dropped from 15 down to 11) on the upgrade. This was on a query that crunched 5,000 rows down to 1,256. The difference on a query that reduced a 100,000-row input to a 100-row output could be much more dramatic.

Indexes

There is little to say about indexes—if you have to build an index by reading a table, most of the work comes from sorting the data, but Oracle only costs for acquiring the data (i.e., the tablescan, index fast full scan, or index full scan that will be used to get the necessary columns and rowids).

If you want to work out the memory requirements for building a B-tree index, remember that it is basically a simple sort. But one little detail you have to remember is that you are sorting with one more column than you expect—the **rowid**, which is 6 bytes for a normal table, but 8 bytes for a cluster index, and 10 bytes for a global index on a partitioned table (but only 6 for a local index on a partitioned table).

As you might guess, the memory requirements for building a bitmap index are highly variable—just as the size of a bitmap index is highly variable. At present I don't know how you can work out a general estimate of the memory requirements for building a bitmap index.

There is one other special feature of indexes to consider if you are still using the manual `workarea_size_policy` or running 8*i*—for versions 8*i* and 9*i*, all the memory comes from the PGA (and not the UGA), whether or not you set a `sort_area_retained_size`. In 10*g*, memory is allocated from the UGA—which is little surprising and sounds like a threat, but is irrelevant if

you are running with the automatic `workarea_size_policy` because the memory will be freed properly as soon as the index build is finished.

Set Operations

Oracle offers three set operations—union, intersect, and minus, as shown in Table 13-6. You might also want to include `union all` in your list of set operations, but technically it's not a set operation, as sets do not allow duplicates, and the result of a `union all` may include duplicates (it's also a very boring operation—Oracle just handles a couple of queries separately and runs them one after the other).

Table 13-6. *The Set Operators in SQL*

Operation	The Result Set Contains
`union`	One example of every possible row in the inputs
`intersect`	One example of each row that appear in both inputs
`minus`	One example of each row from the first input that does not appear in the second input

The need to eliminate "duplicates" will probably make you think of sorting and the `distinct` operator—so let's take a look at what Oracle does with these set operations. In the script `set_ops.sql` in the online code suite, I have created two very simple tables:

```
create table t1 as
select
        rownum id,
        ao.*
from
        all_objects ao
where
        rownum <= 2500
;

create table t2 as
select
        rownum id,
        ao.*
from
        all_objects ao
where
        rownum <= 2000
;
```

With this definition, table t2 happens to be a proper subset of table t1, containing exactly 2,000 of the rows that also exist in t1. To demonstrate the effects of the three set operations, I need only run three queries (plus one spare to demonstrate the nonsymmetrical nature of the minus operator), with the results shown in Table 13-7.

Table 13-7. *Example Results from the Set Operators*

Operation	The Result Set Looks Like

`select * from t1`
`union`
`select * from t2`
	The 2,500 rows from t1

`select * from t1`
`intersect`
`select * from t2`
	The 2,000 rows in t2

`select * from t1`
`minus`
`select * from t2`
	The 500 rows in t1 that are not in t2

`select * from t2`
`minus`
`select * from t1`
	The empty set—the zero rows that are in t2 but not in t1

The result sets are very easy to describe, because we know that the rows in the two sets we are operating on are already unique, and that one is a subset of the other. Things become a little more interesting when we start using base queries where there may be duplicates. Let's examine an example where we select only the owner and object_type from the two tables.

If we know that sets are supposed to contain no duplicates, we might decide that we are supposed to enforce this uniqueness requirement before we do a set operation, so we might write

```
select distinct owner, object_type from t1
intersect
select distinct owner, object_type from t2
;
```

On the other hand, if we understand how set operators are supposed to work—returning a proper set by eliminating duplicates—we might decide that it is safe do to the following and let Oracle sort out the problem of duplicates by itself:

```
select       owner, object_type from t1
intersect
select       owner, object_type from t2
;
```

Here are the two execution plans:

```
Execution Plan (9.2.0.6 - using distinct - no system stats)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=30 Card=15 Bytes=375)
   1    0   INTERSECTION
   2    1     SORT (UNIQUE) (Cost=15 Card=15 Bytes=180)
   3    2       TABLE ACCESS (FULL) OF 'T1' (Cost=6 Card=2500 Bytes=30000)
   4    1     SORT (UNIQUE) (Cost=15 Card=15 Bytes=195)
   5    4       TABLE ACCESS (FULL) OF 'T2' (Cost=6 Card=2000 Bytes=26000)

Execution Plan  (9.2.0.6 - NO distinct - no system stats)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=37 Card=2000 Bytes=56000)
   1    0   INTERSECTION
   2    1     SORT (UNIQUE) (Cost=19 Card=2500 Bytes=30000)
   3    2       TABLE ACCESS (FULL) OF 'T1' (Cost=6 Card=2500 Bytes=30000)
   4    1     SORT (UNIQUE) (Cost=18 Card=2000 Bytes=26000)
   5    4       TABLE ACCESS (FULL) OF 'T2' (Cost=6 Card=2000 Bytes=26000)
```

Whether we included the `distinct` operator or not, the optimizer has specified a `sort (unique)` (lines 2 and 4) before taking the intersection. Moreover, if you run the queries with `event 10032` set to check the number of rows sorted and the number of comparisons made, you will find that the run-time engine has done exactly the same amount of work in both cases.

It is "common knowledge" that you do not need to include the `distinct` operator as I have done in the first example. In fact, some people will say it is an error to include the `distinct` because it will introduce redundant sorting (which is not the case, as this example shows). So, in general, queries using set operators tend to be written without the `distinct`.

Look closely at the costs and cardinalities, though. They are completely different. When we introduce the `distinct` operator, the optimizer estimates the number of rows that will be produced by each of the two halves of the query using the normal `Group by Cardinality` calculations, which results in lower costs and lower final cardinality. (The difference in costs disappears in this example if you enable CPU costing—but that's just the normal oddity of how costing for in-memory sorts can change as you enable CPU costing.)

SET OPERATIONS AND DISTINCT

Contrary to the received wisdom on set operations, it seems that the practice of omitting the `distinct` operator is actually a bad idea—the optimizer does a better job of estimating the cardinality of the base result set if you include the `distinct`. It shouldn't matter, but at present it does.

When it comes to the cardinality of the result from the set operator itself, we are back on familiar territory. Running the tests under 10*g*, I got the following sets of figures shown in Table 13-8. As you can see, once the optimizer has worked out the cardinalities of the base queries, it uses a worst-case scenario to work out the results of the set operation.

Table 13-8. *Cardinality Methods for Sets Operators*

Data Set	Group Cardinality	Reason
t1	20	owner (4) * object_type (7) / 1.4142
t2	15	owner (3) * object_type (7) / 1.4142
union	35	t1 value + t2 value
intersect	15	least (t1 value, t2 value)
minus	20	First (i.e., t1) value

You have to be a little careful with the assumptions you make about how the SQL will be operated. It is possible for things to change, and it may well have been true that in 8.0 or 7.3 the presence of the apparently redundant `distinct` would have resulted in an extra pair of sorts taking place. Things do change, and bugs can appear.

If you look in the `10053 trace`, by the way, you won't find any information about the set operation itself—there will be a section for each of the two separate queries, followed by a final cost that represents the set operation. But there is no explanation or justification for the final figures. So you can only assume, as I do, that the optimizer just adds the cost of the separate queries to get the final cost of the set operation.

Consider, as a final warning then, the following CTAS (which also appears in script `set_ops.sql` in the online code suite):

```
create table t_intersect as
select distinct *
from (
        select owner, object_type from t1
        intersect
        select owner, object_type from t2
)
;
```

Clearly, the `distinct` in the outer `select` statement is redundant—but we have already seen that Oracle is very good at avoiding sorts that are not really needed (not just in set operations, but also in our earlier example that featured a `group by` followed by an `order by`). So what's the execution plan for the CTAS? Here's the output from `dbms_xplan` in 9.2.0.6 with CPU costing enabled:

```
-------------------------------------------------------------------------------
| Id  | Operation             |  Name       | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | CREATE TABLE STATEMENT |             |  2000 | 56000 |    16  (25)|
|   1 |  LOAD AS SELECT        |             |       |       |            |
|   2 |   VIEW                 |             |  2000 | 56000 |            |
|   3 |    INTERSECTION        |             |       |       |            |
|   4 |     SORT UNIQUE        |             |  2500 | 30000 |     8  (25)|
|   5 |      TABLE ACCESS FULL | T1          |  2500 | 30000 |     7  (15)|
|   6 |     SORT UNIQUE        |             |  2000 | 26000 |     8  (25)|
|   7 |      TABLE ACCESS FULL | T2          |  2000 | 26000 |     7  (15)|
-------------------------------------------------------------------------------
```

The optimizer has very cleverly noticed that the `distinct` was redundant: according to the execution plan, the result set from the in-line view in line 2 has not been sorted.

And here's the 10*g* version of the plan—again with CPU costing enabled:

```
-------------------------------------------------------------------------------
| Id  | Operation              | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | CREATE TABLE STATEMENT |      |  2000 | 56000 |    20  (15)| 00:00:01 |
|   1 |  LOAD AS SELECT        |      |       |       |            |          |
|   2 |   SORT UNIQUE          |      |  2000 | 56000 |    18  (17)| 00:00:01 |
|   3 |    VIEW                |      |  2000 | 56000 |    17  (12)| 00:00:01 |
|   4 |     INTERSECTION       |      |       |       |            |          |
|   5 |      SORT UNIQUE       |      |  2500 | 32500 |     7  (15)| 00:00:01 |
|   6 |       TABLE ACCESS FULL| T1   |  2500 | 32500 |     6   (0)| 00:00:01 |
|   7 |      SORT UNIQUE       |      |  2000 | 26000 |     7  (15)| 00:00:01 |
|   8 |       TABLE ACCESS FULL| T2   |  2000 | 26000 |     6   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Notice the `sort unique` at line 2? It looks like 10*g* has reintroduced the redundant sort that 9*i* was smart enough to eliminate.

Fortunately, we can always check up on the sorting that actually happens by enabling `event 10032`—and guess what—the 9*i* execution plan is lying. The `10032 trace` files from both 9*i* and 10*g* show the same characteristic pattern. And I've printed an extract of the 10*g* trace here:

```
Input records                          2000         -- T1
Output records                         9

Input records                          2500         -- T2
Output records                         10

Input records                          9            -- The intersection view
Output records                         9

Input records                          9
Output records                         9

Input records                          2500
Output records                         10

Input records                          2000
Output records                         9
```

You can see that Oracle has sorted the two tables—2,500 and 2,000 rows, respectively, to produce 10 and 9 rows of output, respectively. The third sort takes place after the intersection has produced its 9 rows. If you remove the redundant distinct operation from the query, this third sort operation disappears. Both versions do the redundant sort—but only 10*g* admits to it in the execution plan.

■ **Note** I pointed out much earlier on in this chapter that the sort statistics get printed twice—the way the second printouts appear in reverse order in this example gives you an interesting insight into the stack-driven nature of the run-time engine.

One of the most important things you can learn from this example is that even though the optimizer will sometimes lie to you, when it comes to sorting, the `10032 trace` can tell you what has really happened.

Final Warning

I've already made this point several times in this book, but here it is just one last time. What the engine does at run time isn't necessarily what the optimizer thought about when costing. Here's a lovely example of an amazing run-time trick that the optimizer (currently) knows nothing about. The script is `short_sort.sql` in the online code suite:

```
create table t1
as
with generator as (
        select  --+ materialize
                rownum         id,
                substr(dbms_random.string('U',6),1,6)        sortcode
        from    all_objects
        where   rownum <= 5000
)
select
        /*+ ordered use_nl(v2) */
        substr(v2.sortcode,1,4) || substr(v1.sortcode,1,2) sortcode
from
        generator        v1,
        generator        v2
where
        rownum <= 1 * 1048576
;

--        Collect statistics using dbms_stats here

select  sortcode
from    t1
order by
        sortcode
;
```

```
select * from (
        select * from t1 order by sortcode
)
where
        rownum <= 10
;
```

The cost for both these queries is the same—most of it coming from the line where the sort operation occurs; for example, this is the plan for the second query in my usual 9*i* environment:

```
Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2757 Card=10 Bytes=50)
   1    0   COUNT (STOPKEY)
   2    1     VIEW (Cost=2757 Card=1048576 Bytes=5242880)
   3    2       SORT (ORDER BY STOPKEY) (Cost=2757 Card=1048576 Bytes=7340032)
   4    3         TABLE ACCESS (FULL) OF 'T1' (Cost=266 Card=1048576 Bytes=7340032)
```

The session statistics showed a million rows sorted for each query, and when I ran a `10032 trace` at the same time, it confirmed a million rows sorted. Yet, the first query needed about 5 seconds before it was ready to return any data, while the second query returned all its data in less than half a second. Why?

Checking the `10032 trace` file, you will see that the first query did 20 million comparisons to sort its data, and the second did 1 million comparisons. When running the query with the `rownum` limitation, the run-time engine simply scanned the table, keeping a cache of the top 10 values. It didn't really sort 1,000,000 rows, it merely checked each row to see if it was larger than the smallest item in the current cache and should replace it. At the end of the scan, it only had 10 rows to sort. (And in 10*g*, the `10032 trace` also showed that the full sort had taken 28MB and still spilled to disk, while the second sort had needed just 8,192 bytes.)

The second query is much quicker than the first—but the optimizer doesn't yet have a costing model that matches the run-time operation.

Summary

The mechanisms for using `sort_area_size` and `sort_area_retained_size` for manual `workarea_size_policy` still apply to Shared Servers (MTS), so you do need to know a little about how they work.

The code underpinning the use of automatic `workarea_size_policy` and the `pga_aggregate_target` uses the same sorting mechanisms, but makes a much better tactical decision on the amount of memory allowed. Since the work done by the CPU in sorting increases with the size of the binary tree, Oracle adopts a strategy of attempting to sort completely in memory (to minimize disk I/O), but backing off to the minimum memory (plus a safety allowance) that would allow for a onepass sort when it discovers that it cannot achieve an optimal sort. This minimizes subsequent CPU consumption without changing the total I/O resource requirement.

Although the use of CPU costing and the automatic `workarea_size_policy` is the correct strategic choice, there is still a large gap between the optimizer's model for sorting and the actual run-time activity, which may make it difficult for people to feel confident about making the change.

The sort/merge join allows for joins based on range-based predicates, which makes them more flexible than hash joins. The costs for a sort/merge join seem to be dictated almost entirely by the cost of the two steps for acquiring the sorted data—the merge itself typically seems to be free, no matter how complex it is. The only exception to this is the Cartesian merge join.

The mechanism for optimizing range-based joins, however, requires the second data set to be instantiated so that a merge can be restarted for each row of the first input. There seems to be a defect in the mechanism that can make this type of merge join very resource-intensive at run time, even though the cost does not reflect the extra work.

`Distinct`, `group by`, and `order by` are all underpinned by sorting (until 10*g*R2), but the only cost catered to by the optimizer is the sort cost. There does seem to be a component in the cost calculation that allows for the size of the output on grouping and distinct from 9*i* onwards.

Sorting can be labor-intensive, and sometimes you may see Oracle doing more work than you think is reasonable for a given execution plan. There are cases where the optimizer and the run-time engine do not agree on how to operate a query—in the case of sorting, the `10032 trace` is very good at telling you what has really happened.

Test Cases

The files in the download for this chapter are shown in Table 13-9.

Table 13-9. *Chapter 13 Test Cases*

Script	Comments
`sort_demo_01.sql`	Sets up first demonstration of how sorting works
`snap_myst.sql`	Reports changes in current session's statistics
`snap_ts.sql`	Reports changes in I/O statistics at the tablespace level
`c_mystats.sql`	Create view used by script `snap_myst.sql`—has to be run as `SYS`
`sort_demo_01a.sql`	Runs further tests against the data set from `sort_demo_01.sql`
`sort_demo_01b.sql`	Runs tests using CPU costing against the data set from `sort_demo_01.sql`
`pat_dump.sql`	Sets `pga_aggregate_target` according to input parameter and run query
`sas_dump.sql`	Sets `sort_area_size` according to input parameter and run query
`pat_nocpu_harness.sql`	Generate a script to `pga_aggregate_target` and call `pat_dump.sql` repeatedly
`pat_cpu_harness.sql`	As `pat_nocpu_harness.sql`, but with some system statistics set
`sas_nocpu_harness.sql`	Generates a script to set `sort_area_size` and call `sas_dump.sql` repeatedly
`sas_cpu_harness.sql`	As `sas_nocpu_harness.sql`, but with some system statistics set
`no_sort.sql`	A "no-sort" merge can sort the second set before getting the whole first set
`cartesian.sql`	Example of a Cartesian merge join

Table 13-9. *Chapter 13 Test Cases*

Script	Comments
agg_sort.sql	Various reasons for sorting
gby_onekey.sql	Demonstration of 9*i* on sorting on multiple columns
merge_samples.sql	Examples of merge joins
set_ops.sql	Examples of union, intersect, and minus
short_sort.sql	Shows how the optimizer fails to cost a new run-time trick
setenv.sql	Sets a standardized environment for SQL*Plus

CHAPTER 14

The 10053 Trace File

It would be impossible to write a good book on the cost based optimizer without mentioning the **optimizer debug trace event: 10053**. It is well known that the `10053 trace` file can tell you a lot about what the optimizer is doing to calculate the cardinalities and costs involved in evaluating an execution plan. The information is never complete, and there are all sorts of oddities that might exist in such a trace file. But, for your entertainment, this chapter walks through the `10053 trace` from a simple four-table join run under 10*g*R1, describing some of the features that it exposes.

To enable the `10053 trace`, you need to issue one of these statements:

```
alter session set events '10053 trace name context forever';
alter session set events '10053 trace name context forever, level 1';
alter session set events '10053 trace name context forever, level 2';
```

The first two do exactly the same thing—the last option produces a slightly shorter trace file because it doesn't include the optimizer parameter listing that is normally at the start of the trace file.

To stop tracing, issue

```
alter session set events '10053 trace name context off';
```

You will find that the `10053 trace` does not work on SQL embedded in PL/SQL unless you are running 10*g*.

The visual impact of this chapter had to be totally different from the rest of the book. The original trace file is reprinted in the "code" format, and there is a lot of it. To allow the eye a break, I have included subheadings at each new join order. Note that I put the comments *before* the text that I am commenting on, although there are a couple of notes that have to refer backwards as well as forwards.

Note Inevitably, 10*g*R2 does things differently. The order of the sections is different, there are numerous extra hints clues and explanations (including a list of descriptions for all the little abbreviations like CDN), the system statistics are reported, and the final execution plan with predicates and outline is included.

The Query

In earlier versions of Oracle, the query was printed near the top of the `10053 trace` file. This moved to the bottom in the later versions of 9.2 and 10.1, so I have listed here the query I used. The script to create the data is `big_10053.sql` in the online code suite.

You will notice that the table names give an obvious clue about the intent of the SQL, and that I have put the tables in the order that I think Oracle probably ought to visit them.

I have also followed a convention in the `where` clause that each predicate should read

```
column_i_want = value_i_know
```

This is obviously the case with lines like the following:

```
ggp.small_num_ggp between 100 and 150
```

It is less obvious, despite my intention, that this should be true on lines like this one:

```
gp.id_ggp = ggp.id
```

When I write a line like this, it is because I am assuming that the query will follow an order of operation that means it will already have visited the `greatgrandparent(ggp)` table before getting to this line, so that the expression `ggp.id` is now a constant.

By writing SQL that follows this convention, I hope to give future readers of my code some clues about my understanding of the purpose of the SQL, the path I expect Oracle to take, and the indexes that might exist to support this query.

You will note that, for more complex queries, I also tend to put a couple of empty comment lines on either side of each group of predicates that "belong with" a specific table.

```
select
        count(ggp.small_vc_ggp),
        count(gp.small_vc_gp),
        count(p.small_vc_p),
        count(c.small_vc_c)
from
        greatgrandparent        ggp,
        grandparent             gp,
        parent                  p,
        child                   c
where
        ggp.small_num_ggp between 100 and 150
/*                              */
and     gp.id_ggp = ggp.id
and     gp.small_num_gp between 110 and 130
/*                              */
and     p.id_gp = gp.id
and     p.id_ggp = gp.id_ggp
and     p.small_num_p between 110 and 130
/*                              */
and     c.id_p = p.id
and     c.id_gp = p.id_gp
and     c.id_ggp = p.id_ggp
and     c.small_num_c between 200 and 215
```

The Execution Plan

The one thing that the trace file does not report is an image of the final execution plan; you have to infer what it would look like by analyzing the contents of the identified join order. To make life a little easier, I have also run the query through `dbms_xplan` so that you can see it here before you begin reading the trace file. You can get a similar plan (without the predicate information) in the trace file by setting `event 10132` at the same time as you set `event 10053`.

Slightly counterintuitively, Oracle 10.1.0.4 picked a path that didn't quite follow the "obvious" order implied by the table names. And at run time, the path that Oracle took did use marginally less resources on its chosen plan than it did on the plan I was expecting.

■ **Note** In the following output, I've modified the `Name` column to show `GP` for table `grandparent` and `GGP` for table `greatgrandparent`. This was to allow the result to fit the page without ugly line wraps in the middle of the table. I've deleted the quotation marks that usually appear in the `Predicate Information` section for the same reason.

```
-------------------------------------------------------------------------------------
| Id  | Operation                      | Name    | Rows| Bytes | Cost (%CPU)| Time |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |         |   1 |    96 | 361 (1)| 00:00:04 |
|   1 |  SORT AGGREGATE                |         |   1 |    96 |        |          |
|   2 |   NESTED LOOPS                 |         |   1 |    96 | 361 (1)| 00:00:04 |
|   3 |    NESTED LOOPS                |         |   1 |    77 | 360 (1)| 00:00:04 |
|   4 |     NESTED LOOPS               |         |   6 |   300 | 348 (1)| 00:00:04 |
|*  5 |      TABLE ACCESS FULL         | GP      | 110 |  2530 | 128 (1)| 00:00:02 |
|*  6 |      TABLE ACCESS BY INDEX ROWID| PARENT |   1 |    27 |   2 (0)| 00:00:01 |
|*  7 |       INDEX RANGE SCAN         | P_PK    |   1 |       |   1 (0)| 00:00:01 |
|*  8 |     TABLE ACCESS BY INDEX ROWID | CHILD   |   1 |    27 |   2 (0)| 00:00:01 |
|*  9 |      INDEX RANGE SCAN          | C_PK    |   1 |       |   1 (0)| 00:00:01 |
|* 10 |    TABLE ACCESS BY INDEX ROWID  | GGP     |   1 |    19 |   1 (0)| 00:00:01 |
|* 11 |     INDEX UNIQUE SCAN          | GGP_PK  |   1 |       |   0 (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   5 - filter(GP.SMALL_NUM_GP<=130 AND GP.SMALL_NUM_GP>=110)
   6 - filter(P.SMALL_NUM_P<=130 AND P.SMALL_NUM_P>=110)
   7 - access(P.ID_GGP=GP.ID_GGP AND P.ID_GP=GP.ID)
   8 - filter(C.SMALL_NUM_C<=215 AND C.SMALL_NUM_C>=200)
   9 - access(C.ID_GGP=P.ID_GGP AND C.ID_GP=P.ID_GP AND C.ID_P= P.ID)
  10 - filter(GGP.SMALL_NUM_GGP>=100 AND GGP.SMALL_NUM_GGP<=150)
  11 - access(GP.ID_GGP=GGP.ID)
```

The Environment

One thing you can't see from the trace file (until 10*g*R2) is the state of the system statistics (used for CPU costing). Before running the query, I had executed the following anonymous PL/SQL block:

```
begin
        dbms_stats.set_system_stats('MBRC',8);
        dbms_stats.set_system_stats('MREADTIM',20);
        dbms_stats.set_system_stats('SREADTIM',10);
        dbms_stats.set_system_stats('CPUSPEED',500);
end;
/
```

Remember that the cost formula (rearranged) from the *Oracle Performance Tuning Guide and Reference* that I quoted in Chapter 1 tells us that

```
Cost = (
        #SRds +
        #MRds * mreadtim / sreadtim +
        #CPUCycles / (cpuspeed * sreadtim)
)
```

This means that, depending on exactly how each version makes use of rounding and truncating at different parts of the calculation, the cost of a tablescan will be approximately

```
1 + ceil((high water mark / MBRC) * (mreadtim / sreadtim)) + a CPU cost =
1 + ceil((high water mark / 8) * 20/10)   +  (a bit) =
1 + ceil((high water mark / 4))  +  (a bit)
```

Given the number of blocks in the four tables (which you will see later), we get the costs shown in Table 14-1 for the I/O components of the tablescans.

Table 14-1. *The Table Sizes and Tablescan I/O Costs*

Table Name	Blocks	I/O Cost
greatgrandparent	250	64
grandparent	500	126 (127)
parent	2,500	626 (627)
child	10,000	2,501

The figures in brackets are the actual values acquired from a separate test—I frequently find that my predications are out by the odd plus or minus one (which seems to float depending on the version of Oracle). In this case, I think I will ascribe the error to a computational error dealing with a 0.5 that appears from *high water mark* / 4.

Finally, to convert CPU resource figures into costs, we have to apply the last line of the generic formula:

```
Cost = (#CPUCycles / cpuspeed) / sreadtim) =
        #CPUCycles / (cpuspeed * sreadtim)
```

Since the `cpuspeed` figure is stored in **MHz**—which means millions of cycles (or Oracle operations) per second, and the `sreadtim` is stored in milliseconds, we have to apply a fudge factor of 1,000 to ensure that we are using the same time units throughout. So with a CPU speed of 500, and a single block read time of 10,000 microseconds, we need to divide any figures for CPU cycles by 5,000,000 before adding them to the cost.

The Trace File

The first section of the `10053 trace` file output lists the actual values used by the optimizer as a result of bind variable peeking. (The sample section that follows is a fake to demonstrate the format, as my query didn't use any bind variables.) It is unfortunate that the names of the bind variables are not given here, only their position. You just have to count your way through the statement to match variables with their values.

```
*****************************************
Peeked values of the binds in SQL statement
*****************************************
bind 0:dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=03 oacfl2=0000 size=48 offset=0
   bfp=065ddc30 bln=22 avl=02 flg=05
   value=100
bind 1:dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=03 oacfl2=0000 size=0 offset=24
   bfp=065ddc48 bln=22 avl=03 flg=01
   value=150
bind 3:dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=03 oacfl2=0000 size=0 offset=48
   No bind buffers allocated
```

Parameter Settings

The second section of the output lists the current settings of the parameters relevant to the optimizer. If you compare the full list of options available in `x$ksppi`, you may find some items that look as if they should be in this list. Conversely, if you look at `v$sql_optimizer_env` (see Appendix B), you will find that there are a lot more in this list than appear in the view.

The 10*g* output conveniently separates the parameter list into two sets—those modified by the session, and those that use the default value.

If you think you know a lot about the optimizer and how it works, take a look at some of the "feature-oriented" parameters, typically the ones with the `true/false` values. I find it quite worrying to see how many clues I get that there are details of the optimizer that I've probably never ever seen in action.

```
************************************
PARAMETERS USED BY THE OPTIMIZER
********************************
  ************************************
  PARAMETERS WITH ALTERED VALUES
  ******************************
  ************************************
  PARAMETERS WITH DEFAULT VALUES
  ******************************
  optimizer_mode_hinted               = false
  optimizer_features_hinted           = 0.0.0
  parallel_execution_enabled          = true
  parallel_query_forced_dop           = 0
  parallel_dml_forced_dop             = 0
  parallel_ddl_forced_degree          = 0
  parallel_ddl_forced_instances       = 0
  _query_rewrite_fudge                = 90
  optimizer_features_enable           = 10.1.0.4
  _optimizer_search_limit             = 5
  cpu_count                           = 1
  active_instance_count               = 1
  parallel_threads_per_cpu            = 2
  hash_area_size                      = 131072
  bitmap_merge_area_size              = 1048576
  sort_area_size                      = 65536
  sort_area_retained_size             = 0
  _sort_elimination_cost_ratio        = 0
  _optimizer_block_size               = 8192
  _sort_multiblock_read_count         = 2
  _hash_multiblock_io_count           = 0
  db_file_multiblock_read_count       = 8
  _optimizer_max_permutations         = 2000
  pga_aggregate_target                = 204800 KB
  _pga_max_size                       = 204800 KB
  _sort_space_for_write_buffers       = 1
  _query_rewrite_maxdisjunct          = 257
  _smm_auto_min_io_size               = 56 KB
  _smm_auto_max_io_size               = 248 KB
  _smm_min_size                       = 204 KB
  _smm_max_size                       = 10240 KB
  _smm_px_max_size                    = 61440 KB
  _cpu_to_io                          = 0
  _optimizer_undo_cost_change         = 10.1.0.4
  parallel_query_mode                 = enabled
  parallel_dml_mode                   = disabled
  parallel_ddl_mode                   = enabled
  optimizer_mode                      = all_rows
```

```
sqlstat_enabled                  = false
_optimizer_percent_parallel      = 101
_always_anti_join                = choose
_always_semi_join                = choose
_optimizer_mode_force            = true
_partition_view_enabled          = true
_always_star_transformation      = false
_query_rewrite_or_error          = false
_hash_join_enabled               = true
cursor_sharing                   = exact
_b_tree_bitmap_plans             = true
star_transformation_enabled      = false
_optimizer_cost_model            = choose
_new_sort_cost_estimate          = true
_complex_view_merging            = true
_unnest_subquery                 = true
_eliminate_common_subexpr        = true
_pred_move_around                = true
_convert_set_to_join             = false
_push_join_predicate             = true
_push_join_union_view            = true
_fast_full_scan_enabled          = true
_optim_enhance_nnull_detection   = true
_parallel_broadcast_enabled      = true
_px_broadcast_fudge_factor       = 100
_ordered_nested_loop             = true
_no_or_expansion                 = false
optimizer_index_cost_adj         = 100
optimizer_index_caching          = 0
_system_index_caching            = 0
_disable_datalayer_sampling      = false
query_rewrite_enabled            = true
query_rewrite_integrity          = enforced
_query_cost_rewrite              = true
_query_rewrite_2                 = true
_query_rewrite_1                 = true
_query_rewrite_expression        = true
_query_rewrite_jgmigrate         = true
_query_rewrite_fpc               = true
_query_rewrite_drj               = true
_full_pwise_join_enabled         = true
_partial_pwise_join_enabled      = true
_left_nested_loops_random        = true
_improved_row_length_enabled     = true
_index_join_enabled              = true
_enable_type_dep_selectivity     = true
_improved_outerjoin_card         = true
```

```
_optimizer_adjust_for_nulls         = true
_optimizer_degree                   = 0
_use_column_stats_for_function      = true
_subquery_pruning_enabled           = true
_subquery_pruning_mv_enabled        = false
_or_expand_nvl_predicate            = true
_like_with_bind_as_equality         = false
_table_scan_cost_plus_one           = true
_cost_equality_semi_join            = true
_default_non_equality_sel_check     = true
_new_initial_join_orders            = true
_oneside_colstat_for_equijoins      = true
_optim_peek_user_binds              = true
_minimal_stats_aggregation          = true
_force_temptables_for_gsets         = false
workarea_size_policy                = auto
_smm_auto_cost_enabled              = true
_gs_anti_semi_join_allowed          = true
_optim_new_default_join_sel         = true
optimizer_dynamic_sampling          = 2
_pre_rewrite_push_pred              = true
_optimizer_new_join_card_computation = true
_union_rewrite_for_gs               = yes_gset_mvs
_generalized_pruning_enabled        = true
_optim_adjust_for_part_skews        = true
_force_datefold_trunc               = false
statistics_level                    = typical
_optimizer_system_stats_usage       = true
skip_unusable_indexes               = true
_remove_aggr_subquery               = true
_optimizer_push_down_distinct       = 0
_dml_monitoring_enabled             = true
_optimizer_undo_changes             = false
_predicate_elimination_enabled      = true
_nested_loop_fudge                  = 100
_project_view_columns               = true
_local_communication_costing_enabled = true
_local_communication_ratio          = 50
_query_rewrite_vop_cleanup          = true
_slave_mapping_enabled              = true
_optimizer_cost_based_transformation = linear
_optimizer_mjc_enabled              = true
_right_outer_hash_enable            = true
_spr_push_pred_refspr               = true
_optimizer_cache_stats              = false
_optimizer_cbqt_factor              = 50
```

```
  _optimizer_squ_bottomup             = true
  _fic_area_size                      = 131072
  _optimizer_skip_scan_enabled        = true
  _optimizer_cost_filter_pred         = false
  _optimizer_sortmerge_join_enabled   = true
  _optimizer_join_sel_sanity_check    = true
  _mmv_query_rewrite_enabled          = false
  _bt_mmv_query_rewrite_enabled       = true
  _add_stale_mv_to_dependency_list    = true
  _distinct_view_unnesting            = false
  _optimizer_dim_subq_join_sel        = true
  _optimizer_disable_strans_sanity_checks = 0
  _optimizer_compute_index_stats      = true
  _push_join_union_view2              = true
  _optimizer_ignore_hints             = false
  _optimizer_random_plan              = 0
  _query_rewrite_setopgrw_enable      = true
  _optimizer_correct_sq_selectivity   = true
  _disable_function_based_index       = false
  _optimizer_join_order_control       = 3
  _optimizer_push_pred_cost_based     = true
***************************************
Column Usage Monitoring is ON: tracking level = 1
***************************************
```

Query Blocks

Query blocks (which, loosely speaking, means visible subqueries or in-line views) acquired names in 10*g*. You can explicitly name a block with the `qb_name` hint, and use it to refer to object aliases in global hints. If you don't supply a name, Oracle uses a generated name for each block. If you have a complex query that includes (for example) a subquery or view that cannot be merged, then you will find that the `10053 trace` file may have a number of totally separate sections—one section for each nonmergeable query block, and a section where the nonmergeable parts are joined.

The query block signature helps you to see how the query has been broken down into parts by listing the tables, or more precisely the aliases, (`fro(N)`), that appear in that block. The aliases seem to be reported in alphabetical order.

```
QUERY BLOCK SIGNATURE
*********************
qb name was generated
signature (optimizer): qb_name=SEL$1 nbfros=4 flg=0
  fro(0): flg=0 objn=58048 hint_alias="C"@"SEL$1"
  fro(1): flg=0 objn=58045 hint_alias="GGP"@"SEL$1"
  fro(2): flg=0 objn=58046 hint_alias="GP"@"SEL$1"
  fro(3): flg=0 objn=58047 hint_alias="P"@"SEL$1"
```

Stored Statistics

The next section of the output is a simple restatement of the basic statistics on the tables involved, and the indexes that might be of use. These figures are simply echoing the figures from `user_tables`, `user_tab_col_statistics`, and `user_indexes` (or the partition equivalents). If you are using partitioning, you will see comments here about (`using composite stats`) or, for a known single partition, a label identifying the partition by number, e.g., `PARTITION [0]`. These comments appear only at the table and index level, not at the column level.

The columns reported are only those that could be used for filtering and access at this point—earlier versions of Oracle will report them in different locations in the trace.

There is a little oddity about histograms: I did not create any histograms for this query, nevertheless Oracle reports `1 uncompressed buckets`. This is just a way of saying that I have nothing but the low/high (or as they now appear in the 10*g* trace `Min:/Max:`) values for this column.

In the absence of histograms, you will note that the `NDV` (number of distinct values) for each column = `1/Density`.

It is an interesting little detail that the tables appear to be listed in the reverse order that they appear in the `from` clause.

```
***************************************
BASE STATISTICAL INFORMATION
***********************

Table stats    Table: CHILD   Alias:  C
  TOTAL ::  CDN: 40000  NBLKS:  10000  AVG_ROW_LEN:  1632
  COLUMN:       ID_P(NUMBER)  Col#: 3      Table: CHILD   Alias: C
    Size: 4  NDV: 10000  Nulls: 0  Density: 1.0000e-004 Min: 1  Max: 10000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  COLUMN:      ID_GP(NUMBER)  Col#: 2      Table: CHILD   Alias: C
    Size: 4  NDV: 2000  Nulls: 0  Density: 5.0000e-004 Min: 1  Max: 2000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  COLUMN:     ID_GGP(NUMBER)  Col#: 1      Table: CHILD   Alias: C
    Size: 4  NDV: 1000  Nulls: 0  Density: 1.0000e-003 Min: 1  Max: 1000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
Index stats
  Index: C_PK  COL#: 1 2 3 4
    TOTAL ::  LVLS: 1   #LB: 108  #DK: 40000  LB/K: 1  DB/K: 1  CLUF: 40000
***********************

Table stats    Table: PARENT   Alias:  P
  TOTAL ::  CDN: 10000  NBLKS:  2500  AVG_ROW_LEN:  1627
  COLUMN:      ID_GP(NUMBER)  Col#: 2      Table: PARENT   Alias: P
    Size: 4  NDV: 2000  Nulls: 0  Density: 5.0000e-004 Min: 1  Max: 2000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
```

```
  COLUMN:     ID_GGP(NUMBER)  Col#: 1      Table: PARENT   Alias: P
    Size: 4  NDV: 1000  Nulls: 0  Density: 1.0000e-003 Min: 1  Max: 1000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  COLUMN:         ID(NUMBER)  Col#: 3      Table: PARENT   Alias: P
    Size: 4  NDV: 10000  Nulls: 0  Density: 1.0000e-004 Min: 1  Max: 10000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  COLUMN:      ID_GP(NUMBER)  Col#: 2      Table: PARENT   Alias: P
    Size: 4  NDV: 2000  Nulls: 0  Density: 5.0000e-004 Min: 1  Max: 2000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  COLUMN:     ID_GGP(NUMBER)  Col#: 1      Table: PARENT   Alias: P
    Size: 4  NDV: 1000  Nulls: 0  Density: 1.0000e-003 Min: 1  Max: 1000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
Index stats
  Index: P_PK  COL#: 1 2 3
    TOTAL ::  LVLS: 1   #LB: 24  #DK: 10000  LB/K: 1  DB/K: 1  CLUF: 10000
***********************

Table stats    Table: GRANDPARENT   Alias: GP
  TOTAL ::  CDN: 2000  NBLKS:  500  AVG_ROW_LEN:  1623
  COLUMN:     ID_GGP(NUMBER)  Col#: 1      Table: GRANDPARENT   Alias: GP
    Size: 4  NDV: 1000  Nulls: 0  Density: 1.0000e-003 Min: 1  Max: 1000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  COLUMN:         ID(NUMBER)  Col#: 2      Table: GRANDPARENT   Alias: GP
    Size: 4  NDV: 2000  Nulls: 0  Density: 5.0000e-004 Min: 1  Max: 2000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  COLUMN:     ID_GGP(NUMBER)  Col#: 1      Table: GRANDPARENT   Alias: GP
    Size: 4  NDV: 1000  Nulls: 0  Density: 1.0000e-003 Min: 1  Max: 1000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
Index stats
  Index: GP_PK  COL#: 1 2
    TOTAL ::  LVLS: 1   #LB: 6  #DK: 2000  LB/K: 1  DB/K: 1  CLUF: 2000
***********************

Table stats    Table: GREATGRANDPARENT   Alias: GGP
  TOTAL ::  CDN: 1000  NBLKS:  250  AVG_ROW_LEN:  1619
  COLUMN:         ID(NUMBER)  Col#: 1      Table: GREATGRANDPARENT   Alias: GGP
    Size: 4  NDV: 1000  Nulls: 0  Density: 1.0000e-003 Min: 1  Max: 1000
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
Index stats
```

```
  Index: GGP_PK  COL#: 1
    TOTAL ::  LVLS: 1   #LB: 2  #DK: 1000  LB/K: 1  DB/K: 1  CLUF: 250
_OPTIMIZER_PERCENT_PARALLEL = 0
```

Single Tables

The next section is about the "single table" access paths, where the optimizer works out for each table individually the number of rows that will be acquired, and the cheapest cost of getting them, based on the assumption that the only access paths into the table are those governed by the constants (literal or bind) that are supplied in the original query.

At this point, the optimizer ignores any column conditions that are relevant to join conditions; however, it is allowed to read constraints from the database and apply transitive closure to create new conditions, for example:

```
If you have the predicates
        col1 = 'x' and col2 = col1
then Oracle can infer
        col2 = 'x'
```

or

```
if you have a constraint
        colX = upper(colx)
then Oracle could take the predicate
        upper(colX) = 'ABC'
and infer the predicate
        colX = 'ABC'
```

This treatment of constraints seems to have undergone some changes in 10.1, and no longer works when the constants are replaced by bind variables. This may be a bug, but it may have been a deliberate change to avoid incorrect results with nulls

Following is a worked example for the `greatgrandparent` table:

```
We have a predicate: small_num_ggp between 100 and 150
Number of distinct values (NDV)                    = 200
Number of nulls (NULLS)                            = 0
Density (DENS)                                     = 0.005  (1/200)
Low value (Min)                                    = 0
High values (Max)                                  = 199
According to the selectivity formula for a 'between' range,
        Selectivity = (val2 - val1) / (high - low)  + 2 / num_distinct =
                      50 / 199 + 2/200 =
                      0.261256
```

Multiply by 1,000 rows and round to get a computed cardinality of 261. The cost for the tablescan was given in Table 14-1, but you will notice that the `BEST_CST` (best cost) for acquiring the data is given here as 64.41, while the cost reported Table 14-1 was only 64. The difference is the CPU component of the cost. It is unfortunate that the trace file does not show the breakdown of total cost into the `IO cost` and `CPU cost` as it does in other places.

```
***************************************
SINGLE TABLE ACCESS PATH
  COLUMN: SMALL_NUM_(NUMBER)  Col#: 2      Table: GREATGRANDPARENT   Alias: GGP
    Size: 4  NDV: 200  Nulls: 0  Density: 5.0000e-003 Min: 0  Max: 199
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  TABLE: GREATGRANDPARENT  Alias: GGP
    Original Card: 1000   Rounded: 261  Computed: 261.26  Non Adjusted: 261.26
  Access Path: table-scan  Resc:  64  Resp:  64
  BEST_CST: 64.41  PATH: 2  Degree:  1

***************************************
SINGLE TABLE ACCESS PATH
  COLUMN: SMALL_NUM_(NUMBER)  Col#: 3      Table: GRANDPARENT   Alias: GP
    Size: 4  NDV: 400  Nulls: 0  Density: 2.5000e-003 Min: 0  Max: 399
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  TABLE: GRANDPARENT  Alias: GP
    Original Card: 2000   Rounded: 110  Computed: 110.25  Non Adjusted: 110.25
  Access Path: table-scan  Resc:  128  Resp:  128
  BEST_CST: 127.82  PATH: 2  Degree:  1

***************************************
SINGLE TABLE ACCESS PATH
  COLUMN: SMALL_NUM_(NUMBER)  Col#: 4      Table: PARENT   Alias: P
    Size: 4  NDV: 2000  Nulls: 0  Density: 5.0000e-004 Min: 0  Max: 1999
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  TABLE: PARENT  Alias: P
    Original Card: 10000   Rounded: 110  Computed: 110.05  Non Adjusted: 110.05
  Access Path: table-scan  Resc:  631  Resp:  631
  BEST_CST: 631.09  PATH: 2  Degree:  1

***************************************
SINGLE TABLE ACCESP PATH
  COLUMN: SMALL_NUM_(NUMBER)  Col#: 5      Table: CHILD   Alias: C
    Size: 4  NDV: 10000  Nulls: 0  Density: 1.0000e-004 Min: 0  Max: 9999
    No Histogram: #BKT: 1
        (1 uncompressed buckets and 2 endpoint values)
  TABLE: CHILD  Alias: C
    Original Card: 40000   Rounded: 68  Computed: 68.01  Non Adjusted: 68.01
  Access Path: table-scan  Resc:  2517  Resp:  2517
  BEST_CST: 2517.49  PATH: 2  Degree:  1
```

Sanity Checks

This next section is new to 10*g*, relating to the multicolumn join sanity check feature. If there are two (or more) join conditions between two tables, the optimizer used to treat each condition separately for the purposes of deciding which tables should supply the selectivity. In 10*g*, the optimizer takes the selectivity from just one side of the join. It has two strategies for doing this.

If the join covers the whole of a concatenated index, then Oracle will consider the `distinct_keys` column from `user_indexes` view as a possible join selectivity. If there is no suitable index, Oracle considers the product of the selectivities from one table or the other, but does not pick and choose from each side.

There is a slight oddity in the reporting of the `concatenated index card`. In the example, the `child` table is reported with a cardinality of 10,000—which is the `distinct_keys` from the `parent` index, which is where the join is going. But the optimizer is simply telling us that there has to be at most this number of distinct values in our selection from the `child` table because it is joined to the `parent` table, and we actually know that there are only 10,000 values in the `parent` table. Similarly, the `parent` table is reported with a cardinality of 2,000—which is the `distinct_keys` from the `grandparent` table, which is where the join is going.

You will find entries for concatenated indexes in 9*i* trace files, but you won't find the entries for the multicolumn join key.

```
Table:  CHILD  Concatenated index card: 10000.000000
Table:  PARENT  Concatenated index card: 2000.000000
Table:  CHILD  Multi-column join key card: 40000.000000
Table:  PARENT  Multi-column join key card: 10000.000000
Table:  PARENT  Multi-column join key card: 10000.000000
Table:  GRANDPARENT  Multi-column join key card: 2000.000000
***************************************
OPTIMIZER STATISTICS AND COMPUTATIONS
***************************************
```

General Plans

Finally, we start to work on joins. In this case, there are 24 permutations that we might have to deal with. The first six might be the ones that start with `greatgrandparent`:

```
ggp-gp-p-c,  ggp-gp-c-p,  ggp-p-gp-c,  ggp-p-c-gp,  ggp-c-gp-p,  ggp-c-p-gp.
```

After this, Oracle might do six starting with `grandparent`, then six starting with `parent`, then six starting with `child`. Will it check all of them, and which one will be first?

The first question is easy—the answer is, *Not necessarily*. For a very short list of tables (up to and including five), the optimizer *may* consider every join order, but even then it may dismiss some as a waste of effort almost immediately.

The second question requires a little more thought. Consider the results from the single table access path section, shown in Table 14-2.

Table 14-2. *Summary of "Single Table Access Path" Calculations*

Table Name	Cost	Computed Cardinality
greatgrandparent	64.41	261.26
grandparent	127.82	110.25
parent	631.09	110.05
child	2,517.49	68.01

Join order[1]

When we check, the first join order is: child → parent → grandparent → greatgrandparent. This does give us two possible options for guessing. Either the first join order is in order of computed cardinality, or it is in reverse order of cost. (A quick test shows it to be the order of cardinality.)

In fact, there is a special "single-row table" rule. Tables that are known to produce exactly one row (and this typically means a table with a single value predicate on a unique/primary key, although it could be a simple aggregate view) are promoted to the front of the list, and stay there. Inevitably these tables may end up being involved in Cartesian joins, but a Cartesian join between two single-row tables is still just one row, so the effect on the cost is irrelevant and the tables do not play a part in the main costing work thereafter.

There is an exception to the "single-row table" rule: if you use the leading() hint to specify the first table(s) in the join order, the "single-row table" rule has to be discarded, and every table except the leading table(s) will then participate in the full optimization process.

You will notice that each table has its alias (possibly system generated) and a number following its name in the Join order[] line. You will find that these aliases and numbers are used to refer to the table at various stages later on in the trace file.

```
GENERAL PLANS
***********************
Join order[1]: CHILD[C]#0  PARENT[P]#1  GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3
```

Now joining (1)

We only ever join one more table at a time. For each table, Oracle tries a nested loop, a sort/merge (sometimes twice) and a hash join, in that order. The hints use_nl, use_merge, and use_hash apply specifically at this Now joining point, and tell Oracle which join mechanism it *must* use.

In the case of the nested loop join, we see the cost as

```
Outer table cost (2517) plus
        Outer table cardinality (68) * best inner table unit cost (1) = 2585
```

Of course, you will see that the Best NL cost: is quoted at the end of the section as 2,586. In isolation (in a separate test), the cost of scanning and acquiring a few rows from the parent table was 2,517.82, which explains the small error that our calculation produces. The best we can ever do from a 10053 trace is always going to be a little inaccurate, because the information supplied is (a) not intended for us and (b) not complete.

I suspect the P_PK index is evaluated twice because one piece of code uses the standard formula for single-table access by index (unique), and another uses the special case of unique index guaranteeing at most one row (eq-unique).

You will notice the ix_sel and ix_sel_with_filters (which would be labeled tb_sel in earlier versions of Oracle) are zero for the eq_unique option. This conveniently reflects the fact that the special cost calculation for unique access on unique indexes is blevel + 1 and need not involve subtle calculation of leaf block and table block visits.

Since our query predicates don't do anything odd with the indexes (such as missing out columns or using ranges on leading columns), we will see ix_sel = ix_sel_with_filters all the way through this example.

```
Now joining: PARENT[P]#1 *******
NL Join
  Outer table: cost: 2517  cdn: 68  rcz: 27  resp:  2517
  Inner table: PARENT  Alias: P
    Access Path: table-scan  Resc: 629
    Join:  Resc:  45302  Resp:  45302
  Access Path: index (unique)
    Index: P_PK
    rsc_cpu: 15620   rsc_io: 1
    ix_sel:  1.0000e-004    ix_sel_with_filters:  1.0000e-004
    NL Join:  resc: 2586  resp: 2586
  Access Path: index (eq-unique)
    Index: P_PK
    rsc_cpu: 15820   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 2586  resp: 2586
  Best NL cost: 2586  resp: 2586
```

The join cardinality is calculated at the end of the nested loop cost calculation. For this reason (I assume) the nested loop calculation always appears in the trace file, even if you have put a use_merge or use_hash hint into your query to block the use of a nested loop.

Note how a sanity check has kicked in. The traditional selectivity mechanism has produced a value that is smaller than 1/(*number of rows in table*), so one of the other figures has been used—in this case the selectivity of the fully utilized P_PK index.

```
Using concatenated index cardinality for table PARENT
Revised join selectivity: 1.0000e-004 = 7.9445e-007 * (1/10000) * (1/7.9445e-007)
Join Card:  0.75 = outer (68.01) * inner (110.05) * sel (1.0000e-004)
```

We now move on to the sort/merge join. In this example, we try one mechanism for the SM Join. A little lower down is an example where we try two different options for the SM Join.

Historically, the cost of a sort/merge join has shown some bizarre statistics—in particular the relationship between the figures that went into the I/O cost per pass and the Total I/O sort cost seemed to be based on an unusual rationale.

The critical calculations are in the Row size (sum of relevant average column lengths), and the Total Rows (computed cardinality) for each table. Oracle works out the CPU cost of sorting these rows, and the possible volume of I/O that will need to take place if the memory requirement exceeds the limit. Since we are using pga_aggregate_target here, and have a very small

pair of sorts, the `Area size:` is the `_smm_min_size` (0.1% of the 200MB we have for the `pga_aggregate_target`), and the `Max Area size:` is the `_smm_max_size` (5% of the target).

In this case, the data to be sorted from both tables is very small, so there is no I/O cost. (You can also see that there is no estimated `Total Temp space used:`.)

The cost of the join is therefore

```
Outer table data acquisition cost +
Outer table sort cost (CPU only in this case) +
Inner table data acquisition cost +
Inner table sort cost (CPU only in this case)
```

To turn the reported `Total CPU sort cost` into standard units, we divide by 5,000,000 (the 500 MHz * 10,000 microsecond `sreadtim`) to get

```
Cost of join =
        2,517 + 5,018,650 / 5,000,000 +
        631 + 5,033,608 / 5,000,000 =
        3,148 + 2 and a bit.
```

You can see at the end of the following section that the actual cost quoted for the merge is 3,151, which compares well with the 3,150 plus a bit that I have just calculated. Since I can't guarantee the timing, or direction, of rounding used, I can't be certain whether the difference of one is due to rounding, or to a small cost that is directly attributable to the merge operation itself.

```
SM Join
  Outer table:
    resc: 2517  cdn: 68  rcz: 27  deg: 1  resp: 2517
  Inner table: PARENT  Alias: P
    resc: 631  cdn: 110  rcz: 27  deg:  1  resp: 631
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:             68
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5018650
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  3151  Resp:  3151
```

Finally the hash join. For optimal hash joins, the cost of the join is basically the cost of acquiring the two data sets, plus a little bit. In this case:

```
        Outer table cost = 2517
        Inner table cost = 631
        Hash join cost = 3149 = 2517 + 631 + 1
```

The extra one is the `Hash join one ptn` (partition) cost.

```
HA Join
  Outer table:
    resc: 2517  cdn: 68  rcz: 27  deg: 1  resp: 2517
  Inner table: PARENT  Alias: P
    resc: 631  cdn: 110  rcz: 27  deg:  1  resp: 631
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 3149   Resp: 3149
```

We have now tried all three (four) join mechanisms, with the following best costs in each case:

```
        NL        2586
        SM        3151
        HA        3149
```

So we report the cost of the cheapest option, and remember that as the cost of producing the intermediate data set.

```
Join result: cost: 2586  cdn: 1  rcz: 54
```

Now joining (2)

The optimizer only ever joins two tables—so it's time to join the next table in the current join order to our intermediate data set. The `Outer table` isn't really a table, of course.

```
Now joining: GRANDPARENT[GP]#2 *******
NL Join
  Outer table: cost: 2586  cdn: 1  rcz: 54  resp:  2586
  Inner table: GRANDPARENT  Alias: GP
    Access Path: table-scan  Resc: 128
    Join:  Resc:  2714  Resp:  2714
  Access Path: index (unique)
    Index: GP_PK
    rsc_cpu: 15589   rsc_io: 1
    ix_sel:  5.0000e-004    ix_sel_with_filters:  5.0000e-004
    NL Join:  resc: 2587  resp: 2587
  Access Path: index (eq-unique)
    Index: GP_PK
    rsc_cpu: 15789   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 2587  resp: 2587
  Best NL cost: 2587  resp: 2587
Using concatenated index cardinality for table GRANDPARENT
```

```
Revised join selectivity: 5.0000e-004 = 8.3417e-005 * (1/2000) * (1/8.3417e-005)
Join Card:  0.04 = outer (0.75) * inner (110.25) * sel (5.0000e-004)

SM Join
  Outer table:
    resc: 2586  cdn: 1  rcz: 54  deg: 1  resp: 2586
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           70 Total Rows:              1
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000000
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           36 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  2716  Resp:  2716

HA Join
  Outer table:
    resc: 2586  cdn: 1  rcz: 54  deg: 1  resp: 2586
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 2714   Resp: 2714

Join result: cost: 2587  cdn: 1  rcz: 77
```

Now joining (3)

We've added the third table to the list, so now we add the fourth table to the intermediate result. Notice that the cost of the `Outer table` is now the cost of the join of the previous three tables.

```
Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 2587  cdn: 1  rcz: 77  resp:  2587
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 64
    Join:  Resc:  2651  Resp:  2651
```

```
  Access Path: index (unique)
    Index: GGP_PK
    rsc_cpu: 15558   rsc_io: 1
    ix_sel:  1.0000e-003    ix_sel_with_filters:  1.0000e-003
    NL Join:  resc: 2588  resp: 2588
  Access Path: index (eq-unique)
    Index: GGP_PK
    rsc_cpu: 15758   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 2588  resp: 2588
  Best NL cost: 2588  resp: 2588
Join Card:  0.04 = outer (0.04) * inner (261.26) * sel (3.8168e-003)

SM Join
  Outer table:
    resc: 2587  cdn: 1  rcz: 77  deg: 1  resp: 2587
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           95 Total Rows:              1
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000000
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           31 Total Rows:            261
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5094402
      Total Temp space used: 0
  Merge join  Cost:  2653  Resp:  2653

HA Join
  Outer table:
    resc: 2587  cdn: 1  rcz: 77  deg: 1  resp: 2587
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 2652   Resp: 2652

Join result: cost: 2588  cdn: 1  rcz: 96
```

Best so far

We've got to the end of the first join order. Every time we get to the end of a join order, the optimizer checks the cost against the best cost so far, and if the latest cost is better, it gets reported and remembered. The first cost is, of course, the best so far because there has been no previous cost.

The instances of TABLE# correspond to the table numbers listed in the first join order.

The figures by CST, CDN, and BYTES are the figures that would appear in the plan table as the columns cost, cardinality, and bytes. You can see in the Join result at the end of the previous section that we have a fields called cdn (cardinality) and rcz (the record size)—the BYTES figure that follows comes from the product of these two figures.

```
Best so far: TABLE#: 0  CST:       2517  CDN:         68  BYTES:       1836
Best so far: TABLE#: 1  CST:       2586  CDN:          1  BYTES:         54
Best so far: TABLE#: 2  CST:       2587  CDN:          1  BYTES:         77
Best so far: TABLE#: 3  CST:       2588  CDN:          1  BYTES:         96
**********************
```

Join order[2]

We move on to the second join order. For a small number of tables (typically five or less), Oracle will do a simple cyclic permutation of the tables, starting from the end of the first join order, working through all possible permutations. So the second join order simply swaps the last two tables.

```
Join order[2]: CHILD[C]#0  PARENT[P]#1  GREATGRANDPARENT[GGP]#3  GRANDPARENT[GP]#2
```

Wait a moment—why does the next line in the trace file say we are joining greatgrandparent? Surely it should be parent? No—because Oracle has remembered that Join order [1] started with child → parent, and has carried the intermediate results of that join forward to this join order.

Remember, as the SQL statements get "larger," the amount of memory used to optimize them increases. This is partly because Oracle has to remember more and more intermediate results as the number of tables increases.

```
Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 2586  cdn: 1  rcz: 54  resp:  2586
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 64
    Join:  Resc:  2650  Resp:  2650
  Best NL cost: 2650  resp: 2650
Join Card:  195.53 = outer (0.75) * inner (261.26) * sel (1.0000e+000)
**********************
```

What happened to the SM Join and the HA Join? Where's the line saying Now Joining: GRANDPARENT? We are about to jump into Join order [3] without completing Join order [2]. Look at the cost of completing the nested loop join to greatgrandparent (2,650)—it has already exceeded the best cost (2,588) for the complete query, so there is no point in going on with this join order, and the optimizer takes a shortcut.

The reason why you don't see the merge join or hash join is that the join to greatgrandparent is a Cartesian join, so the best possible join cost would have to be at least the sum of the outer table cost plus the inner table cost. There is no point in saying this three times.

There is a subtle point in this join order that isn't immediately obvious. Oracle has treated the join from child → grandparent as a Cartesian join—but we know (from the table definition code) that the primary key of the child table actually starts with the key of the greatgrandparent, and that that column should have propagated down through the join. But this is an example where Oracle does not apply transitive closure.

We can see (rearranging our query) that

```
        child.id_ggp = parent.id_ggp
and     parent.id_ggp = grandparent.id_ggp
and     grandparent.id_ggp = greatgrandparent.id
```

So we can infer that child.id_ggp = greatgrandparent.id. But the optimizer does not attempt to apply this logic.

Join order[3]

As we go into Join Order[3], the change cycles another step closer to the front of the join order. We've tried all the orders that start with (child, parent), so we move on to starting with (child, grandparent).

Again we stop without testing the whole join order, and without testing the merge and hash joins on the very first join.

Moreover, we don't even report the join (child, grandparent, greatgrandparent, parent) that ought to be the next order. There's no point in looking at it; we already know that *anything* starting with (child, grandparent) is too expensive.

```
Join order[3]: CHILD[C]#0  GRANDPARENT[GP]#2  PARENT[P]#1  GREATGRANDPARENT[GGP]#3
Now joining: GRANDPARENT[GP]#2 *******
NL Join
  Outer table: cost: 2517  cdn: 68  rcz: 27  resp:  2517
  Inner table: GRANDPARENT  Alias: GP
    Access Path: table-scan  Resc: 126
    Join:  Resc:  11074  Resp:  11074
  Best NL cost: 11074  resp: 11074
Join Card:  7497.70 = outer (68.01) * inner (110.25) * sel (1.0000e+000)
**********************
```

Join order[4]

The same short-circuiting effect appears as we cycle the next table into second place (child, greatgrandparent); we evaluate the first join (and only the nested loop option) and find that it is more expensive than the best so far, so we don't complete the join order, and don't examine any other join orders that start the same way.

```
Join order[4]: CHILD[C]#0  GREATGRANDPARENT[GGP]#3  PARENT[P]#1  GRANDPARENT[GP]#2
Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 2517  cdn: 68  rcz: 27  resp:  2517
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 63
    Join:  Resc:  6796  Resp:  6796
  Best NL cost: 6796  resp: 6796
Join Card:  17766.99 = outer (68.01) * inner (261.26) * sel (1.0000e+000)
**********************
```

Join order[5]

This is another long one. This isn't too surprising when we look at the names of the tables—we are starting with a pair that we know to be related, and by the time we join the third table (grandparent), we will still have some joinable data. We wouldn't be so lucky if the third table were greatgrandparent.

```
Join order[5]: PARENT[P]#1  CHILD[C]#0  GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3
Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 631  cdn: 110  rcz: 27  resp:  631
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  277488  Resp:  277488
  Access Path: index (scan)
    Index: C_PK
    rsc_cpu: 15543   rsc_io: 2
    ix_sel:  5.0000e-011    ix_sel_with_filters:  5.0000e-011
    NL Join:  resc: 851  resp: 851
  Best NL cost: 851  resp: 851
Using concatenated index cardinality for table PARENT
Revised join selectivity: 1.0000e-004 = 7.9445e-007 * (1/10000) * (1/7.9445e-007)
Join Card:  0.75 = outer (110.05) * inner (68.01) * sel (1.0000e-004)

SM Join
  Outer table:
    resc: 631  cdn: 110  rcz: 27  deg: 1  resp: 631
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
```

```
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:             68
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5018650
      Total Temp space used: 0
  Merge join  Cost:  3151  Resp:  3151

HA Join
  Outer table:
    resc: 631  cdn: 110  rcz: 27  deg: 1  resp: 631
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 3149   Resp: 3149
```

Our current best cost for the full join is 2,588, so it's worth going on after getting this 851 as the cost of joining the first two tables.

```
Join result: cost: 851  cdn: 1  rcz: 54

Now joining: GRANDPARENT[GP]#2 *******
NL Join
  Outer table: cost: 851  cdn: 1  rcz: 54  resp:  851
  Inner table: GRANDPARENT  Alias: GP
    Access Path: table-scan  Resc: 128
    Join:  Resc:  979  Resp:  979
  Access Path: index (unique)
    Index: GP_PK
    rsc_cpu: 15589   rsc_io: 1
    ix_sel:  5.0000e-004    ix_sel_with_filters:  5.0000e-004
    NL Join:  resc: 852  resp: 852
  Access Path: index (eq-unique)
    Index: GP_PK
    rsc_cpu: 15789   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 852  resp: 852
  Best NL cost: 852  resp: 852
Using concatenated index cardinality for table GRANDPARENT
Revised join selectivity: 5.0000e-004 = 8.3417e-005 * (1/2000) * (1/8.3417e-005)
Join Card:  0.04 = outer (0.75) * inner (110.25) * sel (5.0000e-004)

SM Join
  Outer table:
    resc: 851  cdn: 1  rcz: 54  deg: 1  resp: 851
```

```
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           70 Total Rows:              1
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000000
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           36 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  981  Resp:  981

HA Join
  Outer table:
    resc: 851  cdn: 1  rcz: 54  deg: 1  resp: 851
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 980   Resp: 980
```

Our current best cost for the full join is 2,588, so it's still worth going on after getting this 852 as the cost of joining the first three tables.

```
Join result: cost: 852  cdn: 1  rcz: 77

Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 852  cdn: 1  rcz: 77  resp:  852
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 64
    Join:  Resc:  917  Resp:  917
  Access Path: index (unique)
    Index: GGP_PK
    rsc_cpu: 15558   rsc_io: 1
    ix_sel:  1.0000e-003    ix_sel_with_filters:  1.0000e-003
    NL Join:  resc: 853  resp: 853
  Access Path: index (eq-unique)
    Index: GGP_PK
    rsc_cpu: 15758   rsc_io: 1
```

```
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 853  resp: 853
  Best NL cost: 853  resp: 853
Join Card:  0.04 = outer (0.04) * inner (261.26) * sel (3.8168e-003)

SM Join
  Outer table:
    resc: 852  cdn: 1  rcz: 77  deg: 1  resp: 852
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           95 Total Rows:              1
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000000
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           31 Total Rows:            261
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5094402
      Total Temp space used: 0
  Merge join  Cost:  919  Resp:  919

HA Join
  Outer table:
    resc: 852  cdn: 1  rcz: 77  deg: 1  resp: 852
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 917   Resp: 917
Join result: cost: 853  cdn: 1  rcz: 96
```

And in `Join order[5]` we have a new best join so far at a cost of 853. This will be our target/limit from now on.

```
Best so far: TABLE#: 1  CST:        631  CDN:        110  BYTES:       2970
Best so far: TABLE#: 0  CST:        851  CDN:          1  BYTES:         54
Best so far: TABLE#: 2  CST:        852  CDN:          1  BYTES:         77
Best so far: TABLE#: 3  CST:        853  CDN:          1  BYTES:         96
***********************
```

Join order[6]

We had (parent, child, grandparent, greatgrandparent) as Join order[5], so the next permutation simply swaps the last two tables. This means we can use the intermediate result from (parent, child) and go straight into joining greatgrandparent—which is another of those joins that we know isn't really a Cartesian join, but Oracle can't see the connection. So we discard Join order [6] almost immediately.

```
Join order[6]: PARENT[P]#1  CHILD[C]#0  GREATGRANDPARENT[GGP]#3  GRANDPARENT[GP]#2
Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 851  cdn: 1  rcz: 54  resp:  851
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 64
    Join:  Resc:  916  Resp:  916
  Best NL cost: 916  resp: 916
Join Card:  195.53 = outer (0.75) * inner (261.26) * sel (1.0000e+000)
**********************
```

Join order[7]

This is another long one, which results in another best so far.

```
Join order[7]: PARENT[P]#1  GRANDPARENT[GP]#2  CHILD[C]#0  GREATGRANDPARENT[GGP]#3
Now joining: GRANDPARENT[GP]#2 *******
NL Join
  Outer table: cost: 631  cdn: 110  rcz: 27  resp:  631
  Inner table: GRANDPARENT  Alias: GP
    Access Path: table-scan  Resc: 126
    Join:  Resc:  14473  Resp:  14473
  Access Path: index (unique)
    Index: GP_PK
    rsc_cpu: 15589   rsc_io: 1
    ix_sel:  5.0000e-004    ix_sel_with_filters:  5.0000e-004
    NL Join:  resc: 741  resp: 741
  Access Path: index (eq-unique)
    Index: GP_PK
    rsc_cpu: 15789   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 741  resp: 741
  Best NL cost: 741  resp: 741
Using concatenated index cardinality for table GRANDPARENT
Revised join selectivity: 5.0000e-004 = 8.3417e-005 * (1/2000) * (1/8.3417e-005)
Join Card:  6.07 = outer (110.05) * inner (110.25) * sel (5.0000e-004)

SM Join
  Outer table:
    resc: 631  cdn: 110  rcz: 27  deg: 1  resp: 631
```

```
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           36 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  761  Resp:  761

HA Join
  Outer table:
    resc: 631  cdn: 110  rcz: 27  deg: 1  resp: 631
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 759   Resp: 759
Join result: cost: 741  cdn: 6  rcz: 50

Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 741  cdn: 6  rcz: 50  resp:  741
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  15844  Resp:  15844
  Access Path: index (scan)
    Index: C_PK
    rsc_cpu: 15543   rsc_io: 2
    ix_sel:  5.0000e-011    ix_sel_with_filters:  5.0000e-011
    NL Join:  resc: 753  resp: 753
  Best NL cost: 753  resp: 753
Using concatenated index cardinality for table PARENT
Revised join selectivity: 1.0000e-004 = 7.9445e-007 * (1/10000) * (1/7.9445e-007)
Join Card:  0.04 = outer (6.07) * inner (68.01) * sel (1.0000e-004)
```

```
SM Join
  Outer table:
    resc: 741  cdn: 6  rcz: 50  deg: 1  resp: 741
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           65 Total Rows:              6
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000699
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:             68
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5018650
      Total Temp space used: 0
  Merge join  Cost:  3261  Resp:  3261

HA Join
  Outer table:
    resc: 741  cdn: 6  rcz: 50  deg: 1  resp: 741
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 3259   Resp: 3259
Join result: cost: 753  cdn: 1  rcz: 77

Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 753  cdn: 1  rcz: 77  resp:  753
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 64
    Join:  Resc:  818  Resp:  818
  Access Path: index (unique)
    Index: GGP_PK
    rsc_cpu: 15558   rsc_io: 1
    ix_sel:  1.0000e-003    ix_sel_with_filters:  1.0000e-003
    NL Join:  resc: 754  resp: 754
  Access Path: index (eq-unique)
    Index: GGP_PK
    rsc_cpu: 15758   rsc_io: 1
```

```
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 754  resp: 754
  Best NL cost: 754  resp: 754
Join Card:  0.04 = outer (0.04) * inner (261.26) * sel (3.8168e-003)

SM Join
  Outer table:
    resc: 753  cdn: 1  rcz: 77  deg: 1  resp: 753
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           95 Total Rows:              1
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000000
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           31 Total Rows:            261
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5094402
      Total Temp space used: 0
  Merge join  Cost:  820  Resp:  820

HA Join
  Outer table:
    resc: 753  cdn: 1  rcz: 77  deg: 1  resp: 753
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 818   Resp: 818
Join result: cost: 754  cdn: 1  rcz: 96

Best so far: TABLE#: 1  CST:        631  CDN:        110  BYTES:       2970
Best so far: TABLE#: 2  CST:        741  CDN:          6  BYTES:        300
Best so far: TABLE#: 0  CST:        753  CDN:          1  BYTES:         77
Best so far: TABLE#: 3  CST:        754  CDN:          1  BYTES:         96
***********************
```

Join order[8]

For this join order, the optimizer remembers the cost of the (parent, grandparent) join, and works through to the end of the join—but doesn't produce a new best cost.

```
Join order[8]: PARENT[P]#1  GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3  CHILD[C]#0
Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 741  cdn: 6  rcz: 50  resp:  741
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 63
    Join:  Resc:  1121  Resp:  1121
  Access Path: index (unique)
    Index: GGP_PK
    rsc_cpu: 15558   rsc_io: 1
    ix_sel:  1.0000e-003    ix_sel_with_filters:  1.0000e-003
    NL Join:  resc: 747  resp: 747
  Access Path: index (eq-unique)
    Index: GGP_PK
    rsc_cpu: 15758   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 747  resp: 747
  Best NL cost: 747  resp: 747
Join Card:  6.05 = outer (6.07) * inner (261.26) * sel (3.8168e-003)

SM Join
  Outer table:
    resc: 741  cdn: 6  rcz: 50  deg: 1  resp: 741
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           65 Total Rows:              6
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000699
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           31 Total Rows:            261
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5094402
      Total Temp space used: 0
  Merge join  Cost:  808  Resp:  808
```

```
HA Join
  Outer table:
    resc: 741  cdn: 6  rcz: 50  deg: 1  resp: 741
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 806   Resp: 806
Join result: cost: 747  cdn: 6  rcz: 69

Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 747  cdn: 6  rcz: 69  resp:  747
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  15850  Resp:  15850
  Access Path: index (scan)
    Index: C_PK
    rsc_cpu: 15543   rsc_io: 2
    ix_sel:  5.0000e-011    ix_sel_with_filters:  5.0000e-011
    NL Join:  resc: 759  resp: 759
  Best NL cost: 759  resp: 759
Using concatenated index cardinality for table PARENT
Revised join selectivity: 1.0000e-004 = 7.9445e-007 * (1/10000) * (1/7.9445e-007)
Join Card:  0.04 = outer (6.05) * inner (68.01) * sel (1.0000e-004)
Join cardinality for NL:  0.04, outer: 6.05, inner: 68.01, sel: 1.0000e-004

SM Join
  Outer table:
    resc: 747  cdn: 6  rcz: 69  deg: 1  resp: 747
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           86 Total Rows:              6
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000699
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:             68
```

```
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5018650
      Total Temp space used: 0
  Merge join  Cost:  3267  Resp:  3267

HA Join
  Outer table:
    resc: 747  cdn: 6  rcz: 69  deg: 1  resp: 747
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 3265   Resp: 3265
***********************
```

Join order[9]

This join order starts with one of those expensive Cartesian joins, and very rapidly terminates—and the related join (`parent`, `greatgrandparent`, `grandparent`, `child`) is not even considered.

```
Join order[9]: PARENT[P]#1  GREATGRANDPARENT[GGP]#3  CHILD[C]#0  GRANDPARENT[GP]#2
Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 631  cdn: 110  rcz: 27  resp:  631
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 63
    Join:  Resc:  7553  Resp:  7553
  Best NL cost: 7553  resp: 7553
Join Card:  28751.26 = outer (110.05) * inner (261.26) * sel (1.0000e+000)
***********************
```

Join order[10]

Again, the join order starts with one of those expensive Cartesian joins, and terminates immediately—and the related join (`grandparent`, `child`, `greatgrandparent`, `parent`) is not considered.

```
Join order[10]: GRANDPARENT[GP]#2  CHILD[C]#0  PARENT[P]#1  GREATGRANDPARENT[GGP]#3
Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 128  cdn: 110  rcz: 23  resp:  128
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  276985  Resp:  276985
  Best NL cost: 276985  resp: 276985
Join Card:  7497.70 = outer (110.25) * inner (68.01) * sel (1.0000e+000)
***********************
```

Join order[11]

We finally get to a join with no Cartesian joins in it. It runs to completion and happens to produce a new best so far that is a significant improvement on the previous record.

```
Join order[11]: GRANDPARENT[GP]#2  PARENT[P]#1  CHILD[C]#0  GREATGRANDPARENT[GGP]#3
Now joining: PARENT[P]#1 *******
NL Join
  Outer table: cost: 128  cdn: 110  rcz: 23  resp:  128
  Inner table: PARENT  Alias: P
    Access Path: table-scan  Resc: 629
    Join:  Resc:  69338  Resp:  69338
  Access Path: index (scan)
    Index: P_PK
    rsc_cpu: 15523   rsc_io: 2
    ix_sel:  5.0000e-007    ix_sel_with_filters:  5.0000e-007
    NL Join:  resc: 348  resp: 348
  Best NL cost: 348  resp: 348
Using concatenated index cardinality for table GRANDPARENT
Revised join selectivity: 5.0000e-004 = 8.3417e-005 * (1/2000) * (1/8.3417e-005)
Join Card:  6.07 = outer (110.25) * inner (110.05) * sel (5.0000e-004)

SM Join
  Outer table:
    resc: 128  cdn: 110  rcz: 23  deg: 1  resp: 128
  Inner table: PARENT  Alias: P
    resc: 631  cdn: 110  rcz: 27  deg:  1  resp: 631
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           36 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  761  Resp:  761
```

```
HA Join
  Outer table:
    resc: 128  cdn: 110  rcz: 23  deg: 1  resp: 128
  Inner table: PARENT  Alias: P
    resc: 631  cdn: 110  rcz: 27  deg:  1  resp: 631
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 759   Resp: 759
Join result: cost: 348  cdn: 6  rcz: 50

Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 348  cdn: 6  rcz: 50  resp:  348
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  15450  Resp:  15450
  Access Path: index (scan)
    Index: C_PK
    rsc_cpu: 15543   rsc_io: 2
    ix_sel:  5.0000e-011    ix_sel_with_filters:  5.0000e-011
    NL Join:  resc: 360  resp: 360
  Best NL cost: 360  resp: 360
Using concatenated index cardinality for table PARENT
Revised join selectivity: 1.0000e-004 = 7.9445e-007 * (1/10000) * (1/7.9445e-007)
Join Card:  0.04 = outer (6.07) * inner (68.01) * sel (1.0000e-004)

SM Join
  Outer table:
    resc: 348  cdn: 6  rcz: 50  deg: 1  resp: 348
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           65 Total Rows:              6
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000699
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:             68
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5018650
      Total Temp space used: 0
  Merge join  Cost:  2868  Resp:  2868
```

```
HA Join
  Outer table:
    resc: 348  cdn: 6  rcz: 50  deg: 1  resp: 348
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 2866   Resp: 2866
Join result: cost: 360  cdn: 1  rcz: 77

Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 360  cdn: 1  rcz: 77  resp:  360
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 64
    Join:  Resc:  425  Resp:  425
  Access Path: index (unique)
    Index: GGP_PK
    rsc_cpu: 15558   rsc_io: 1
    ix_sel:  1.0000e-003    ix_sel_with_filters:  1.0000e-003
    NL Join:  resc: 361  resp: 361
  Access Path: index (eq-unique)
    Index: GGP_PK
    rsc_cpu: 15758   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 361  resp: 361
  Best NL cost: 361  resp: 361
Join Card:  0.04 = outer (0.04) * inner (261.26) * sel (3.8168e-003)

SM Join
  Outer table:
    resc: 360  cdn: 1  rcz: 77  deg: 1  resp: 360
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           95 Total Rows:              1
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000000
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           31 Total Rows:            261
```

```
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5094402
      Total Temp space used: 0
  Merge join  Cost:  427  Resp:  427

HA Join
  Outer table:
    resc: 360  cdn: 1  rcz: 77  deg: 1  resp: 360
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 425   Resp: 425
Join result: cost: 361  cdn: 1  rcz: 96

Best so far: TABLE#: 2  CST:        128  CDN:        110  BYTES:       2530
Best so far: TABLE#: 1  CST:        348  CDN:          6  BYTES:        300
Best so far: TABLE#: 0  CST:        360  CDN:          1  BYTES:         77
Best so far: TABLE#: 3  CST:        361  CDN:          1  BYTES:         96
**********************
```

Join order[12]

This join order saves a little time because it can use the partial result on (grandparent, parent) from Join order [11]. But then it stops after the third join (greatgrandparent) because the cost has exceeded the best so far.

```
Join order[12]: GRANDPARENT[GP]#2  PARENT[P]#1  GREATGRANDPARENT[GGP]#3  CHILD[C]#0
Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 348  cdn: 6  rcz: 50  resp:  348
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 63
    Join:  Resc:  728  Resp:  728
  Access Path: index (unique)
    Index: GGP_PK
    rsc_cpu: 15558   rsc_io: 1
    ix_sel:  1.0000e-003    ix_sel_with_filters:  1.0000e-003
    NL Join:  resc: 354  resp: 354
  Access Path: index (eq-unique)
    Index: GGP_PK
    rsc_cpu: 15758   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 354  resp: 354
  Best NL cost: 354  resp: 354
Join Card:  6.05 = outer (6.07) * inner (261.26) * sel (3.8168e-003)
```

```
SM Join
  Outer table:
    resc: 348  cdn: 6  rcz: 50  deg: 1  resp: 348
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           65 Total Rows:              6
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000699
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           31 Total Rows:            261
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5094402
      Total Temp space used: 0
  Merge join  Cost:  415  Resp:  415

HA Join
  Outer table:
    resc: 348  cdn: 6  rcz: 50  deg: 1  resp: 348
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 413   Resp: 413
Join result: cost: 354  cdn: 6  rcz: 69

Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 354  cdn: 6  rcz: 69  resp:  354
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  15456  Resp:  15456
  Access Path: index (scan)
    Index: C_PK
    rsc_cpu: 15543   rsc_io: 2
    ix_sel:  5.0000e-011    ix_sel_with_filters:  5.0000e-011
    NL Join:  resc: 366  resp: 366
  Best NL cost: 366  resp: 366
Using concatenated index cardinality for table PARENT
Revised join selectivity: 1.0000e-004 = 7.9445e-007 * (1/10000) * (1/7.9445e-007)
```

```
Join Card:  0.04 = outer (6.05) * inner (68.01) * sel (1.0000e-004)
Join cardinality for NL:  0.04, outer: 6.05, inner: 68.01, sel: 1.0000e-004

SM Join
  Outer table:
    resc: 354  cdn: 6  rcz: 69  deg: 1  resp: 354
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           86 Total Rows:              6
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5000699
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:             68
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5018650
      Total Temp space used: 0
  Merge join  Cost:  2874  Resp:  2874

HA Join
  Outer table:
    resc: 354  cdn: 6  rcz: 69  deg: 1  resp: 354
  Inner table: CHILD  Alias: C
    resc: 2517  cdn: 68  rcz: 27  deg:  1  resp: 2517
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 2872   Resp: 2872
***********************
```

Join order[13]

We get a little way through this join order, and then terminate on the third join—again the visible Cartesian join into `child` causes a problem.

```
Join order[13]: GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3  CHILD[C]#0  PARENT[P]#1
Now joining: GREATGRANDPARENT[GGP]#3 *******
NL Join
  Outer table: cost: 128  cdn: 110  rcz: 23  resp:  128
  Inner table: GREATGRANDPARENT  Alias: GGP
    Access Path: table-scan  Resc: 63
    Join:  Resc:  7049  Resp:  7049
```

```
  Access Path: index (unique)
    Index: GGP_PK
    rsc_cpu: 15558   rsc_io: 1
    ix_sel:  1.0000e-003    ix_sel_with_filters:  1.0000e-003
    NL Join:  resc: 238  resp: 238
  Access Path: index (eq-unique)
    Index: GGP_PK
    rsc_cpu: 15758   rsc_io: 1
    ix_sel:  0.0000e+000    ix_sel_with_filters:  0.0000e+000
    NL Join:  resc: 238  resp: 238
  Best NL cost: 238  resp: 238
Join Card:  109.94 = outer (110.25) * inner (261.26) * sel (3.8168e-003)
Join cardinality for NL:  109.94, outer: 110.25, inner: 261.26, sel: 3.8168e-003

SM Join
  Outer table:
    resc: 128  cdn: 110  rcz: 23  deg: 1  resp: 128
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           36 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           31 Total Rows:            261
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5094402
      Total Temp space used: 0
  Merge join  Cost:  194  Resp:  194

HA Join
  Outer table:
    resc: 128  cdn: 110  rcz: 23  deg: 1  resp: 128
  Inner table: GREATGRANDPARENT  Alias: GGP
    resc: 64  cdn: 261  rcz: 19  deg:  1  resp: 64
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 193   Resp: 193
Join result: cost: 193  cdn: 110  rcz: 42
```

```
Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 193  cdn: 110  rcz: 42  resp:  193
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  277050  Resp:  277050
  Best NL cost: 277050  resp: 277050
Join Card:  7476.42 = outer (109.94) * inner (68.01) * sel (1.0000e+000)
**********************
```

Join order[14]

This join order reuses the partial result for (grandparent, greatgrandparent) from `Join order [13]`, but stops after attempting all possible ways of joining on the third (parent) table.

```
Join order[14]: GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3  PARENT[P]#1  CHILD[C]#0
Now joining: PARENT[P]#1 *******
NL Join
  Outer table: cost: 193  cdn: 110  rcz: 42  resp:  193
  Inner table: PARENT  Alias: P
    Access Path: table-scan  Resc: 629
    Join:  Resc:  69403  Resp:  69403
  Access Path: index (scan)
    Index: P_PK
    rsc_cpu: 15523   rsc_io: 2
    ix_sel:  5.0000e-007    ix_sel_with_filters:  5.0000e-007
    NL Join:  resc: 413  resp: 413
  Best NL cost: 413  resp: 413
Using concatenated index cardinality for table GRANDPARENT
Revised join selectivity: 5.0000e-004 = 8.3417e-005 * (1/2000) * (1/8.3417e-005)
Join Card:  6.05 = outer (109.94) * inner (110.05) * sel (5.0000e-004)

SM Join
  Outer table:
    resc: 193  cdn: 110  rcz: 42  deg: 1  resp: 193
  Inner table: PARENT  Alias: P
    resc: 631  cdn: 110  rcz: 27  deg:  1  resp: 631
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           57 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
```

```
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  826  Resp:  826

HA Join
  Outer table:
    resc: 193  cdn: 110  rcz: 42  deg: 1  resp: 193
  Inner table: PARENT  Alias: P
    resc: 631  cdn: 110  rcz: 27  deg:  1  resp: 631
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 824   Resp: 824
***********************
```

Join order[15]

By the time we get to `Join order [15]`, the `greatgrandparent` table has finally reached the front of the join order. Unfortunately, the second table is the `child` table, with that annoying Cartesian join effect that terminates this join, and ensures that the optimizer doesn't even consider the join (`greatgrandparent`, `child`, `grandparent`, `parent`).

```
Join order[15]: GREATGRANDPARENT[GGP]#3  CHILD[C]#0  PARENT[P]#1  GRANDPARENT[GP]#2
Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 64  cdn: 261  rcz: 19  resp:  64
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  656970  Resp:  656970
  Best NL cost: 656970  resp: 656970
Join Card:  17766.99 = outer (261.26) * inner (68.01) * sel (1.0000e+000)
***********************
```

Join order[16]

Again, we get a Cartesian join between the first two tables, which shortcuts this join calculation and eliminates another without printing it.

```
Join order[16]: GREATGRANDPARENT[GGP]#3  PARENT[P]#1  CHILD[C]#0  GRANDPARENT[GP]#2
Now joining: PARENT[P]#1 *******
NL Join
  Outer table: cost: 64  cdn: 261  rcz: 19  resp:  64
  Inner table: PARENT  Alias: P
    Access Path: table-scan  Resc: 629
    Join:  Resc:  164281  Resp:  164281
  Best NL cost: 164281  resp: 164281
Join Card:  28751.26 = outer (261.26) * inner (110.05) * sel (1.0000e+000)
**********************
```

Join order[17]

Finally, we get a join order that introduces an entirely new option—when we join the grandparent to the greatgrandparent, we find that there are two possible strategies for doing the sort merge join. We can avoid sorting the data from the grandparent if we use its primary key index to get to the rows. This will require us to acquire excess data and discard some of it, and the extra work may prove to be more expensive than the cost of the sort we are avoiding.

The calculation stops after three tables—again the Cartesian join has its effect.

```
Join order[17]: GREATGRANDPARENT[GGP]#3  GRANDPARENT[GP]#2  CHILD[C]#0  PARENT[P]#1
Now joining: GRANDPARENT[GP]#2 *******
NL Join
  Outer table: cost: 64  cdn: 261  rcz: 19  resp:  64
  Inner table: GRANDPARENT  Alias: GP
    Access Path: table-scan  Resc: 126
    Join:  Resc:  32906  Resp:  32906
  Access Path: index (scan)
    Index: GP_PK
    rsc_cpu: 23207   rsc_io: 3
    ix_sel:  1.0000e-003    ix_sel_with_filters:  1.0000e-003
    NL Join:  resc: 849  resp: 849
  Best NL cost: 849  resp: 849
Join Card:  109.94 = outer (261.26) * inner (110.25) * sel (3.8168e-003)
Join cardinality for NL:  109.94, outer: 261.26, inner: 110.25, sel: 3.8168e-003

SM Join
  Outer table:
    resc: 64  cdn: 261  rcz: 19  deg: 1  resp: 64
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:1 distribution:2 #groups:1
```

```
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           31 Total Rows:            261
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5094402
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           36 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  194  Resp:  194
```

Then we do the calculation for the second possibility on the merge join—the one that uses the primary key index to avoid a sort. Because the first data set is going to be presorted when it arrives, we see only one section labeled `SORT resource`. Unfortunately, it turns out to be a waste of effort, as it is a more expensive strategy than not using the index.

Despite comments in the documentation, if Oracle were able to acquire the second set of data in sorted order, it would still sort it—I think this is to cater to the general case where the join condition is not a test for equality and the second set of data has to be instantiated so that a range-based join can keep resetting its start position in the second data set.

```
SM Join (with index on outer)
  Access Path: index (no start/stop keys)
    Index: GGP_PK
    rsc_cpu: 2266849   rsc_io: 253
    ix_sel:  1.0000e+000    ix_sel_with_filters:  1.0000e+000
  Outer table:
    resc: 253  cdn: 261  rcz: 19  deg: 1  resp: 253
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           36 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  382  Resp:  382
```

```
HA Join
  Outer table:
    resc: 64  cdn: 261  rcz: 19  deg: 1  resp: 64
  Inner table: GRANDPARENT  Alias: GP
    resc: 128  cdn: 110  rcz: 23  deg:  1  resp: 128
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 193   Resp: 193
Join result: cost: 193  cdn: 110  rcz: 42

Now joining: CHILD[C]#0 *******
NL Join
  Outer table: cost: 193  cdn: 110  rcz: 42  resp:  193
  Inner table: CHILD  Alias: C
    Access Path: table-scan  Resc: 2517
    Join:  Resc:  277050  Resp:  277050
  Best NL cost: 277050  resp: 277050
Join Card:  7476.42 = outer (109.94) * inner (68.01) * sel (1.0000e+000)
***********************
```

Join order[18]

Our final join order reuses the partial result for (greatgrandparent, grandparent) from Join order [17], and stops on the third table.

```
Join order[18]: GREATGRANDPARENT[GGP]#3  GRANDPARENT[GP]#2  PARENT[P]#1  CHILD[C]#0
Now joining: PARENT[P]#1 *******
NL Join
  Outer table: cost: 193  cdn: 110  rcz: 42  resp:  193
  Inner table: PARENT  Alias: P
    Access Path: table-scan  Resc: 629
    Join:  Resc:  69403  Resp:  69403
  Access Path: index (scan)
    Index: P_PK
    rsc_cpu: 15523   rsc_io: 2
    ix_sel:  5.0000e-007    ix_sel_with_filters:  5.0000e-007
    NL Join:  resc: 413  resp: 413
  Best NL cost: 413  resp: 413
Using concatenated index cardinality for table GRANDPARENT
Revised join selectivity: 5.0000e-004 = 8.3417e-005 * (1/2000) * (1/8.3417e-005)
Join Card:  6.05 = outer (109.94) * inner (110.05) * sel (5.0000e-004)
```

```
SM Join
  Outer table:
    resc: 193  cdn: 110  rcz: 42  deg: 1  resp: 193
  Inner table: PARENT  Alias: P
    resc: 631  cdn: 110  rcz: 27  deg:  1  resp: 631
    using join:1 distribution:2 #groups:1
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           57 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
    SORT resource      Sort statistics
      Sort width:          58 Area size:      208896 Max Area size:    10485760
      Degree:               1
      Blocks to Sort:       1 Row size:           40 Total Rows:            110
      Initial runs:         1 Merge passes:        0 IO Cost / pass:          0
      Total IO sort cost: 0      Total CPU sort cost: 5033608
      Total Temp space used: 0
  Merge join  Cost:  826  Resp:  826

HA Join
  Outer table:
    resc: 193  cdn: 110  rcz: 42  deg: 1  resp: 193
  Inner table: PARENT  Alias: P
    resc: 631  cdn: 110  rcz: 27  deg:  1  resp: 631
    using join:8 distribution:2 #groups:1
  Hash join one ptn Resc: 1   Deg: 1
    hash_area: 124 (max=2560)  buildfrag: 1  probefrag: 1 ppasses: 1
  Hash join   Resc: 824   Resp: 824
```

The next two lines appeared in 10*g*. I don't understand the significance of the (`newjo-stop-1`), although the permutation count (`perm:`)of 18 and the maximum permutation (`maxperm:`) of 2,000 are references to the permutations we saw and the parameter `_optimizer_max_permutations` respectively.

The (`newjo-save`) line simply lists the join order that was finally chosen by quoting the table numbers in order, using the numbers given to the tables in the first join order.

```
(newjo-stop-1) k:0, spcnt:0, perm:18, maxperm:2000
(newjo-save)    [2 1 0 3 ]
```

Finally, we have got to the end of the work, and Oracle reports the join order that gave us the best plan. (This is a luxury introduced by 10*g* and only back-ported to the later versions of 9*i*. In earlier versions you had to read backwards up the file to the last appearance of `Best so far:`.)

The final cost is also reported here, broken down into its I/O and CPU components. As we have done earlier in the file, we can divide the quoted CPU cost by 5,000,000 to find the value to add to the I/O cost to get the overall cost.

```
Final - All Rows Plan:
  JOIN ORDER: 11
  CST: 361  CDN: 1  RSC: 361  RSP: 361  BYTES: 96
  IO-RSC: 360  IO-RSP: 360  CPU-RSC: 5892137  CPU-RSP: 5892137
```

Join Evaluation Summary

It's a lot of work to get this far, and a lot of it is very repetitive. To summarize what has happened with the join orders, I have extracted the orders that the optimizer investigated, and listed them separately here. It can be quite informative simply to use `grep` (Unix) or `find` (Windows) to list all the lines starting with `Join order` to see how many join orders the optimizer investigated—and how many subplans.

```
Join order[1]:   CHILD[C]#0  PARENT[P]#1  GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3
Join order[2]:   CHILD[C]#0  PARENT[P]#1  GREATGRANDPARENT[GGP]#3  GRANDPARENT[GP]#2
Join order[3]:   CHILD[C]#0  GRANDPARENT[GP]#2  PARENT[P]#1  GREATGRANDPARENT[GGP]#3
Join order[4]:   CHILD[C]#0  GREATGRANDPARENT[GGP]#3  PARENT[P]#1  GRANDPARENT[GP]#2

Join order[5]:   PARENT[P]#1  CHILD[C]#0  GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3
Join order[6]:   PARENT[P]#1  CHILD[C]#0  GREATGRANDPARENT[GGP]#3  GRANDPARENT[GP]#2
Join order[7]:   PARENT[P]#1  GRANDPARENT[GP]#2  CHILD[C]#0  GREATGRANDPARENT[GGP]#3
Join order[8]:   PARENT[P]#1  GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3  CHILD[C]#0
Join order[9]:   PARENT[P]#1  GREATGRANDPARENT[GGP]#3  CHILD[C]#0  GRANDPARENT[GP]#2

Join order[10]:  GRANDPARENT[GP]#2  CHILD[C]#0  PARENT[P]#1  GREATGRANDPARENT[GGP]#3
Join order[11]:  GRANDPARENT[GP]#2  PARENT[P]#1  CHILD[C]#0  GREATGRANDPARENT[GGP]#3
Join order[12]:  GRANDPARENT[GP]#2  PARENT[P]#1  GREATGRANDPARENT[GGP]#3  CHILD[C]#0
Join order[13]:  GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3  CHILD[C]#0  PARENT[P]#1
Join order[14]:  GRANDPARENT[GP]#2  GREATGRANDPARENT[GGP]#3  PARENT[P]#1  CHILD[C]#0

Join order[15]:  GREATGRANDPARENT[GGP]#3  CHILD[C]#0  PARENT[P]#1  GRANDPARENT[GP]#2
Join order[16]:  GREATGRANDPARENT[GGP]#3  PARENT[P]#1  CHILD[C]#0  GRANDPARENT[GP]#2
Join order[17]:  GREATGRANDPARENT[GGP]#3  GRANDPARENT[GP]#2  CHILD[C]#0  PARENT[P]#1
Join order[18]:  GREATGRANDPARENT[GGP]#3  GRANDPARENT[GP]#2  PARENT[P]#1  CHILD[C]#0
```

Reduced to the simple numeric form to decrease the visual confusion, and listing the "missing" join orders at the spot that we would expect them to be, we get the following:

```
Join order[1]:   #0  #1  #2  #3                    Best so far              ***
Join order[2]:   #0  #1  #3  #2                    Failed on 3rd
Join order[3]:   #0  #2  #1  #3                    Failed on 2nd
      Skipped:   #0, #2, #3, #1
Join order[4]:   #0  #3  #1  #2                    Failed on 2nd
      Skipped:   #0, #3, #2, #1
```

```
Join order[5]:   #1  #0  #2  #3                    Best so far              ***
Join order[6]:   #1  #0  #3  #2                    Failed on 3rd
Join order[7]:   #1  #2  #0  #3                    Best so far              ***
Join order[8]:   #1  #2  #3  #0                                             ***
Join order[9]:   #1  #3  #0  #2                    Failed on 2nd
      Skipped:   #1, #3, #2, #0

Join order[10]:  #2  #0  #1  #3                    Failed on 2nd
       Skipped:  #2, #0, #3, #1
Join order[11]:  #2  #1  #0  #3                    Best so far              ***
Join order[12]:  #2  #1  #3  #0                                             ***
Join order[13]:  #2  #3  #0  #1                    Failed on 3rd
Join order[14]:  #2  #3  #1  #0                    Failed on 3rd

Join order[15]:  #3  #0  #1  #2                    Failed on 2nd
       Skipped:  #3, #0, #2, #1
Join order[16]:  #3  #1  #0  #2                    Failed on 2nd
       Skipped:  #3, #1, #2, #0
Join order[17]:  #3  #2  #0  #1                    Failed on 3rd
Join order[18]:  #3  #2  #1  #0                    Failed on 3rd
```

As you can see from this listing, for small numbers of tables, the optimizer will cycle through all the possible permutations in a very straightforward fashion. But it will simply skip some permutations that cannot possibly be any good, because a similar permutation with the same starting order of tables is known to be a bad choice. In fact, with a potential 24 join orders, Oracle only ran the calculations to completion on six of the possibilities (marked with ***).

For larger numbers of tables, things can be different. In the first place, the optimizer may decide that after testing a few join orders the cost of executing the query is so small that it might as well go ahead without checking any more join orders. Essentially, as soon as the time spent in checking join orders exceeds the predicted run time, Oracle runs with the best its got so far.

This does mean that if the optimizer happens to pick an unsuitable starting table—because it got the cardinality wrong—the standard permutation cycle can cause a problem. In a ten-table join, it would take a long time to cycle the first table out of pole position. So the optimizer has an algorithm that comes into play with larger numbers of tables, and the modified algorithm starts switching the leading tables around if too many join orders have been tested without a reasonable cost appearing.

■ **Note** For users of Oracle 8, I understand that changing the parameter `optimizer_max_permutations` from its default of 80,000 to any other value will make Oracle 8 switch to the new algorithm. For users of Oracle 10, you might note that there is a new parameter called `_optimizer_join_order_control`, which takes the value 3—so Oracle Corp. may have come up with yet another, better algorithm for permuting through the join orders.

Another of the special cases comes into play with large numbers of tables—where "large" means at least seven. As it works through possible join orders, the optimizer simply ignores, and doesn't even list, any that would require too many Cartesian joins (other than single-row Cartesian joins). The parameter `_optimizer_search_limit` seems to be the parameter that controls the limit on the number of Cartesian joins allowed, and has a default value of 5. Of course, it is instructive to see what happens if you give Oracle a query that joins seven tables, with no scope for anything other than Cartesian joins all the way.

There is a lot more to investigate in the `10053 trace`—in particular the information it gives you with more complex SQL statements that require the optimizer to evaluate subplans for several transformations of your query, and then consider various optional execution mechanisms such as star joins. And I haven't even mentioned what goes on when you use the `first_rows(N)` optimization hint.

But I have to leave something for volumes 2 and 3.

Test Cases

The files in the subdirectory for this chapter are shown in Table 14-3.

Table 14-3. *Chapter 14 Test Cases*

Script	Comments
`big_10053.sql`	The script to create the data for this trace file
`setenv.sql`	Sets a standardized environment for SQL*Plus

APPENDIX A

Upgrade Headaches

When you upgrade from one version of Oracle to another, or even from one patch release to the next, there are bound to be some areas of your code where things just go wrong. The optimizer is a prime target for this kind of issue. New features that improve 99% of all known queries may cripple your database because you fall into the remaining 1% of special cases.

This appendix is just a brief summary of features of the optimizer that may cause problems because implementation details have changed across versions 8*i*, 9*i*, and 10*g*. Every feature (and the possible problem it may introduce) is described in greater detail in the body of the book. The purpose of pulling all the changes into one appendix is to give you a quick reference point should you hit a strange performance problem on an upgrade.

SELECTIVITY AND CARDINALITY

Selectivity and cardinality are very closely related—the selectivity represents that *fraction* of rows that will be selected from a set, and the cardinality represents the *number* of rows that will be extracted. As a general rule, then, you can say

```
cardinality = number of rows out = selectivity * number of rows in.
```

In this appendix, I have referenced only the selectivity when commenting on the effects of upgrades on the optimizer. This is simply to reduce the number of times I would have to repeat phrases like, "And when the selectivity changes, the cardinality changes, hence ..."

It's worth emphasizing that I believe that (almost all of) the changes listed in this appendix are generally good ideas. But any change in the optimizer may be sufficient to make a piece of SQL choose a different execution path, and sometimes the change in path will have an unfortunate impact on performance. Fortunately, there are many features that can be disabled by setting an associated hidden parameter—as indicated in Appendix B by the list of parameters controlled by the master parameter `optimizer_features_enable`.

dbms_stats

If you haven't yet moved from using the `analyze` command to using the `dbms_stats` package, you are going to have to one day. When that day comes, make sure you test carefully, as there are three issues that you have to face.

First, some statistics will be different because the two mechanisms just do things differently, so some execution paths are likely to change. A lightweight example is that `analyze` does not include the length byte in the value for `avg_col_length`, which is used to calculate the cost of sorts and hash joins. A heavy-weight example is that the `analyze` command does a poor job of collecting statistics on partitioned tables, particularly the values for `num_distinct`, and the `dbms_stats` package will produce some dramatically different figures.

Second, you need to ensure that you are collecting the same statistics with `dbms_stats` as you are with `analyze`—for example, `analyze table` and `gather_table_stats` have different *default* behavior; the `analyze` command will collect index statistics, the `gather_table_stats` procedure will not.

Finally, dependent on version, the `dbms_stats` package uses normal SQL to collect a lot of the statistics, rather than custom low-level code. This allows parallelism to be invoked automatically, but means that some stats collections may take longer to run than the equivalent `analyze` commands.

Even if you are already running `dbms_stats`, you aren't safe from change. As you upgrade from version to version of Oracle, the *default* behavior of the procedures in the `dbms_stats` package changes. For example, the default `method_opt` for tables in 10*g* is `for all columns size auto`, so you'll be getting histograms you weren't expecting if you let 10*g* do its own thing, because 10*g* will decide which columns need histograms, and how many buckets to use for those histograms.

Moreover, in 10*g* you will find that a job is automatically created that runs every 24 hours to execute a statistics collection against all tables that are missing statistics, or have "stale" statistics. This could be a lot of redundant work, and the amount could vary randomly and dramatically from day to day.

There are a number of design decisions built into the `dbms_stats` that can cause extreme variations in the statistics produced or the time taken to produce them. The most significant of these probably appears when collecting statistics for indexes—according to one MetaLink note, the code wants to use a sample of at least 919 leaf blocks to generate statistics for an index. This can make it impossible to use a small sample size, and the problem can be extreme for bitmap indexes.

Bear in mind that most objects don't need extremely accurate statistics; and a detailed, minimalist approach to collecting statistics is a good thing—if you can find time to set it up and if you can work around the bugs.

Frequency Histograms

If you have been collecting frequency histograms, you may find that `dbms_stats` doesn't work very well for you when you decide that it's time to stop using the `analyze` command.

The `analyze` command will build a frequency histogram on `N` distinct values if you specify `N` buckets. In 9*i* and 10.1, the equivalent `dbms_stats` call may need many more buckets specified before it notices the option for building a frequency histogram.

In fact, somewhere around the 200 mark you may find that you can no longer create a frequency histogram because you can't specify a large enough bucket count. In cases like this, you may have to write your own SQL to collect suitable figures, and use the `dbms_stats.set_column_stats()` procedure to get those figures into the data dictionary. (This defect has been addressed in 10.2.)

CPU Costing

There are two options for CPU costing (system statistics) in 10*g*. Unless you are already familiar with the use of system statistics in 9*i*, you will find that 10*g* will force your system into CPU costing using some special `noworkload` statistics. This will affect the cost of tablescans significantly—but the scale of the change is different from the changes caused by the normal CPU costing algorithm used by 9*i*. In particular, you will also notice a greater variation in costs if you have code that modifies the `db_file_multiblock_read_count` for different processes.

If you decide to start gathering "workload" system statistics in 10*g*, the effects could be quite surprising when compared with `noworkload` system statistics. Moreover, the `dbms_stats.delete_system_stats()` procedure doesn't seem to work, so the only way to get rid of the gathered statistics if you decide you don't want them is to do an explicit `delete` from the `sys.aux_stats$` table. (This is fixed in 10.2—so you do have a legal way of getting back to `noworkload` system statistics after experimenting with gathered system statistics.)

Rounding Errors

The way in which the optimizer deals with rounding varies from version to version. Some results are rounded properly, some are truncated, and for some values the ceiling is used. This can lead to odd changes in estimated costs and cardinalities—which could result in changes in execution plan—as you upgrade. Most significantly, perhaps, as you upgrade from 8*i* to 9*i*, there are cases where 8*i* rounds intermediate results, but 9*i* only rounds at the end of the entire calculation. This can make a big difference to the final result.

One particular point to watch out for with rounding: the parameter `optimizer_index_cost_adj` can be used to scale *down* the cost of index-driven single-block reads, but this increases the *relative* impact of rounding errors. If you are running 9*i*, you should be using system statistics (CPU costing) to deal with any imbalance between tablescans and indexed access paths. CPU costing scales *up* the cost of multiblock reads, which means it reduces the impact of rounding errors.

If you still have to upgrade from 8*i*, the change in rounding strategy may cause you particular problems if you have systems that store multiple sets of reference data in a single table with a `type` column. If you do run into this problem, re-creating the table as a list-partitioned table should help.

Bind Variable Peeking

Bind variables always cause confusion. A bounded range using bind variables is given a selectivity of 0.0025 (0.25% or 1 in 400), which introduces a bias towards indexed access. An unbounded range using bind variables is given a selectivity of 0.05 (5% or 1 in 20), which tends to introduce a bias towards tablescans. An equality using a bind variable uses the value of `user_tab_columns.density`. In all three cases, histograms are ignored and the resources used to build them will have been wasted. (The `density` may have been modified as a histogram was created, so the effort isn't entirely pointless.)

As of 9*i*, the optimizer usually peeks at the actual values whenever it has to optimize a statement using bind variables, and uses the actual values to evaluate an execution path. So when you upgrade from 8*i* to 9*i*, you may find some SQL statements change their execution

path for no apparent reason. Moreover, a statement could change its execution plan from day to day, or even hour to hour, because it gets flushed from the shared pool from time to time, and the next time it gets executed it is reoptimized with a new set of values that happen to produce a different execution path.

Nulls Across Joins

The basic formula for handling nulls across joins seems to have changed between 8*i* and 9*i*. In 8*i*, the optimizer dealt with nulls in join columns by factoring them out in the *join selectivity* formula, but in 9*i* it seems there is an alternative strategy to handle them in the *join cardinality* formula by adding `is not null` as an explicit join predicate for each column involved.

This alternative rule comes into play when `num_nulls` exceeds 5% of `num_rows`.

B-tree to Bitmap Conversions

One of the optimizer's strategies is to range scan B-tree indexes to acquire lists of rowids, convert the lists of rowids into the equivalent bitmaps, and perform bitwise operations to identify a small set of rows. Effectively, the optimizer can take sets of rowids from index range scans and convert them to bitmap indexes on the fly before doing an `index_combine` on the resulting bitmap indexes.

In 8*i*, only tables with existing bitmap indexes could be subject to this treatment, unless the parameter `_b_tree_bitmap_plans` had been set to relax the requirement for a preexisting bitmap index.

In 9*i*, the default value for this parameter changed from `false` to `true`—so you may see execution plans involving bitmap conversions after you've upgraded, even though you don't have a single bitmap index in your database. Unfortunately, because of the implicit packing assumption that the optimizer uses for bitmap indexes, this will sometimes be a very bad idea.

As a related issue, this change can make it worth using the `minimize_records_per_block` option on all your important tables.

Index Skip-Scans

I have not mentioned the index skip-scan in this volume; it is an access path that appeared in 9*i* for using an index to satisfy a query that (a) does not reference the first column(s) of the index, but (b) could use the index quite well by treating it as a collection of several small subindexes.

This is a beneficial feature in some cases, particularly if you have compressed indexes with a very low number of distinct values in the first column(s). But occasionally you may find that a skip-scan is causing performance problems. If necessary, you can disable the feature by setting the parameter `_optimizer_skip_scan_enabled` to `false`.

AND-Equal

An execution mechanism known as the `and_equal` can be used on queries that involve equalities on single-column, nonunique indexes. Although there are lots of (bad) single-column indexes in the world, you probably won't see this mechanism very often under cost based optimization

until you upgrade from 8*i* to 9*i*. As a weak side effect of CPU costing in 9*i*, there may be cases where the optimizer suddenly starts to use the `and_equal` mechanism where previously it was using a single index, or tablescan. The effect will not always be an improvement.

However, with the arrival of 10*g*, the optimizer no longer considers the `and_equal` mechanism unless hinted—and the hint is deprecated. The mechanism has been largely superseded by the `index_combine` mechanism. `Index_combine` uses bit-wise operations on bitmap indexes; it has been around since 8*i*, and is much more flexible than `and_equal`, as it isn't restricted to equality predicates or to single-column, nonunique indexes. (The `index join` is another of the newer, more flexible mechanisms that has also helped to sound the death knell on `and_equal`.)

The optimizer's ability to use `index_combine` as a substitute for `and_equal` (typically) requires the parameter `_b_tree_bitmap_plans` to be set to `true`—which is the default for 9*i* and 10*g* (see earlier). When this parameter is set to `true`, the optimizer can perform B-tree to bitmap conversion on tables that have no bitmap indexes. I suspect this is generally going to be a little more CPU and memory intensive than doing a straight merge join of the sets of rowids directly—but the trade-off is the increased availability of the path.

But there will be cases where a change in mechanism results in a change in cost; sometimes that change in cost will cause a change in the execution path; just occasionally the new mechanism will be less efficient than the old.

In summary, an upgrade from 8*i* to 9*i* may give you some undesirable path changes because the optimizer *starts* using the `and_equal` mechanism more frequently; and an upgrade from 9*i* to 10*g* may give you some undesirable path changes because the optimizer *stops* using the `and_equal` mechanism completely.

Index Hash Join

These don't often appear as the default path in 8*i*, but when you upgrade to 9*i* and enable CPU costing, they may appear more often because tablescans become more expensive. The index hash join (or just index join) works by combining the contents of two (or more) indexes on a table to avoid visiting the table. Unlike the older `and_equal` path, the indexes don't have to be single column, and the predicates don't have to be equalities. To date, I haven't seen an example where the index join caused a performance problem.

In-List Fixed

The optimizer automatically converts an in-list into a set of predicates separated by the `or` operator. The sample code shows the before and after versions of such a conversion:

```
where   colx in (1,8,32)

where   colx = 1
or      colx = 8
or      colX = 32
```

The standard formula that the optimizer uses for calculating the selectivity of predicates combined with `or` is not appropriate to an `or` expansion of an in-list, as the formula includes a factor (catering to double-counting) that is irrelevant in this special case.

However, the optimizer used this standard formula to handle in-lists all the way through 8*i*, and only introduced a corrected formula for in-lists in 9*i*. This means that the selectivity of an in-list goes up in 9*i*. The most visible side effect of this is that an in-list iterator that originally used an index may switch to using a tablescan. Alternatively, the change in selectivity may even result in a change in the join order.

The change of formula has not yet (as far as 10.1.0.4) been applied to the `not in` clause.

Transitive Closure

- If x > 10 and y = x, then the optimizer will infer that y > 10.
- If x = 10 and y = x, then the optimizer will infer that y = 10.

This type of inference is known as **transitive closure**, and the optimizer applies it in two different ways, depending on version and parameter settings. In the case where the constant predicate is not an equality, the optimizer will generate a third predicate for your query, hence the first example will become

```
where   x > 10
and     y > 10
and     x = y
```

In the case where the constant predicate is an equality (as in the second example), the optimizer will generate a second constant predicate, but lose the join predicate if `query_rewrite_enabled = false`, hence

```
where   x = 10
and     y = 10
```

On the other hand, if `query_rewrite_enabled = true`, then 8*i* and 9*i* will keep the join predicate even in this case, hence

```
where   x = 10
and     y = 10
and     x = y
```

Anytime the optimizer introduces an extra predicate, it could affect the cardinality calculations, so transitive closure is always something to be aware of. However, the most important point in this paragraph is that the upgrade to 10*g* may cause havoc with some execution plans because `query_rewrite_enabled` defaults to `true`, and you will suddenly stop losing join predicates to transitive closure, and some of your execution plans will change. (There are changes in 10gR2 that allow for a different strategy with transitive closure on equality, dependent on a new hidden parameter `_transitivity_closure_retain`.)

sysdate Arithmetic Fixed

Consider the following predicates:

```
where date_col between sysdate     and sysdate + 1
where date_col between sysdate - 1 and sysdate
where date_col between sysdate - 1 and sysdate +1
```

Prior to 10*g*, the optimizer treats `sysdate` as a known constant but treats `sysdate +/- 1` like a bind (or unknown value) variable for purposes of calculating selectivity. The difference also appears for function calls like `trunc(sysdate)` and `trunc(sysdate) +/- 1`. The result of this is that the optimizer treats the range-based predicates shown as an **AND** of two separate predicates, using 0.05 (5%) as the selectivity of the bits involving `sysdate +/- N`. The results of the computed cardinality are therefore usually incorrect, and often close to meaningless.

This error is fixed in 10*g*. Suddenly, many range-based queries involving modifications to `sysdate` will give the right selectivity and cardinality. But any change in selectivity or cardinality may have a catastrophic effect on an execution path, even when the change is a correction.

Indexing Nulls

There are some classes of indexed access paths where the calculations change dramatically as you upgrade to 10.1.0.4.

If you have multicolumn B-tree indexes, and the leading edge of the index is used for a range scan, and you have `user_indexes.num_rows < user_tables.num_rows`, then it looks as if the optimizer *loses* the cost of visiting the index leaf blocks in its calculations. You only need one row in the table where every column in the index has a null value for this change to occur, with a consequential change in access path or join order.

pga_aggregate_target

The memory allocation for various "large-scale" memory operations is now driven by the parameter `pga_aggregate_target` if you set `workarea_size_policy` to `auto`. The work area for a single operation (such as a sort, or hash) can be as large as 5% of the target (or 30% when summed across all the slaves in a parallel operation).

The optimizer calculates the cost of an operation based on that 5% (or 30%). If you have set the parameter to a value that is too large to be realistic, then it is possible that the 5% will never be available at run time. This means the optimizer could end up giving hash joins and sort/merge joins a cost that is too low when compared to the actual resource consumption that has to take place when the query runs.

In 9*i*, the `pga_aggregate_target` does not apply to sessions connected through Shared Servers (formerly MTS), so the `sort_area_size` and `hash_area_size` are still relevant. However, the optimizer will use the figures relevant to a dedicated server connection in its calculations even when connected through a Shared Server, so you could find sessions connected through Shared Servers producing unsuitable execution plans. In 10*g*, this has changed, and the PGA takes on the large memory allocations even for Shared Servers.

There are a couple of hidden parameters (`_smm_max_size` and `_pga_max_size`) that affect the maximum size of a single operation—the 5% limit mentioned previously is further limited to `_pga_max_size / 2`.This can have severely limiting effects on a system that does large sort or hash operations, so you may need to switch some sessions temporarily back to using manual workarea size policy, and setting explicit value for the `sort_area_size` and `hash_area_size`.

Sorting

The algorithms that calculate the cost of a sort have been modified in 9*i* when you have system statistics (CPU costing) enabled—and CPU costing is the default for 10*g*. Most significantly, the cost of an in-memory sort no longer includes a spurious component for dumping the sorted data to disk. This means that there may be cases where the cost of a sort is suddenly very much cheaper than it used to be in earlier versions of Oracle, and this could lead to a dramatic change in execution plan. However, if you have also enabled the automatic workarea size policy feature, then the arithmetic changes again, and this problem may not appear.

Grouping

From 9*i* onwards, the cost of sorting for `group by` or `distinct` clauses has been changed to cater to the fact that the size of the result set could be smaller than the size of the input set. If you have complex queries that include aggregate subqueries that crunch a large number of rows down to a small number of rows, then the cost of these subqueries could drop significantly, resulting in a change in execution plans.

Sanity Checks

There are various sanity checks in the newer versions of Oracle that take place on joins involving multiple columns. This may make the CBO use individual selectivities from just one side of the join, rather than picking the selectivity individually for each join condition. It may also result in the optimizer using the value of `1/num_rows` from one of the tables for the overall selectivity, if the selectivity would otherwise fall below that value.

Going Outside the Limits

Very specific to 10.1.0.4, there is a fix for the problem of predicate values that fall outside the low/high values for a column. Historically, the predicates `column = {constant}` and `column between {const1} and {const2}` had a selectivity that matched the column density, or `1/num_distinct`, when the {constant} was outside the low/high values for the column. In 10.1.0.4, there seems to be a bit of "special case" code that adjusts the selectivity depending on how far outside the range the {constant} is. This applies to equalities, in-lists, and ranges.

This means that you may find queries that used to be safe even when the statistics (especially the low/high values) were out of date suddenly change their paths. In most cases, the number of rows predicted by the optimizer drops as the queried values move further and further away from the known limits.

Type Hacking

Very specific to 10.1.0.4, there is an internal "special case" that addresses some of the problems of third-party applications that store numbers (particularly sequenced IDs) in a character column. When the optimizer sees a range-based predicate on a character column where the low and high values look like numbers, and the predicate tests against something that looks

like a number, the optimizer applies the standard arithmetic to the numeric equivalents of the visible values to generate a selectivity.

It is likely that there will be cases where the selectivity increases enormously as a consequence of this, and that this may change the execution plan of some popular queries.

optimizer_mode

See the script `first_rows.sql` in Chapter 1 for an example of problems associated with use of `first_rows`. `first_rows` is deprecated from 9*i*—`first_rows_1` may be the best bet for many OLTP systems, with `first_rows_10` as a good alternative. The default value in 8*i* and 9*i* is `choose`—which generally means rule based optimization if there are no statistics on any of the objects involved in the query, but switching to `all_rows` optimization if any object in the query has statistics, or one of the queried objects has some feature that forces cost based optimization to be invoked.

Rule based optimization is desupported in 10*g* where the default mode is now `all_rows`.

Descending Indexes

There is a bug prior to 10*g* that makes the optimizer double-count the effect of a predicate involving a descending index, hence produces a cardinality that is much too low. This has been fixed in 10*g*, so queries involving descending indexes may change their access paths because the computed cardinality on one table changes.

Complex View Merging

The parameter `_complex_view_merging` defaults to `false` in 8*i*, and `true` from 9*i*. However, 9*i* merges views without considering costs. It is only in 10*g* that the merged and nonmerged costs are considered and the cheaper option taken. When you upgrade from 8*i* to 9*i*, you may want to stop merging in some queries—the `no_merge(view_alias)` hint placed in the outer query block, or the `no_merge` hint placed in the view itself, can be used to do this.

Unnest Subquery

The parameter `_unnest_subquery` defaults to `false` in 8*i* and `true` from 9*i*. However, 9*i* unnests without considering costs. It is only in 10*g* that the costs of the two different options are considered and the cheaper option taken. When you upgrade from 8*i* to 9*i*, you may want to block unnesting in some queries. The `no_unnest` hint, placed in the subquery, can be used to do this.

Unnesting is disabled in 8*i* and 9*i* (even with the hint) if `star_transformation_enabled = true`.

Scalar and Filter Subqueries

In 8*i* and 9*i*, the size of the hash table used at run time to optimize *scalar* and *filter* subqueries seems to be fixed at 256 entries. In 10*g*, the size of the hash table is a fixed memory allocation, which means the number of saved entries can vary.

When you upgrade, this change may improve the performance of some queries because the number of saved values can increase. However, if the input and output values for the subqueries are large (and a subquery returning an unconstrained `varchar2` is the biggest threat), then performance may get worse—with no change in execution plan—as fewer values are saved and the subqueries are executed more frequently.

Parallel Query Changes x 2

There is an interesting change in strategy for costing parallel queries as you go from version to version of Oracle. In 8*i,* queries are optimized for the best serial execution plan before being run parallel. In 9*i*, the optimizer assumes that the query is going to run at full parallelism, and costs accordingly—this tends to bias the plans towards hash joins, and sort/merge joins particularly. With 10*g*, the rules change again, and there is a fudge factor of 90% introduced to make the heavy-duty join strategies slightly less desirable.

Dynamic Sampling

Dynamic sampling was introduced in 9.2 with a default value of 1 and allows the optimizer to take samples of (at least) 32 blocks from tables during optimization if various conditions are met. There are 11 different levels of dynamic sampling, which can all be set at the system, session, query, or table (within query) level.

In 10*g*, the default level for dynamic sampling is 2, which means that any table without statistics will have 32 blocks sampled. This could prove particularly helpful when you mix global temporary tables (GTTs) with normal tables. However, if you regularly run lots of very small, ad hoc queries involving GTTs, the sampling could turn out to be more labor-intensive than the underlying query, so you may want to identify the most appropriate way of disabling the sample or using the `dbms_stats` package to add representative statistics to the table definition.

Temporary Tables

In 10*g*, the default value for parameter `optimizer_dynamic_sampling` is 2, which means that any object that does not have statistics in place will have 32 blocks sampled at run time. Since global temporary tables (GTTs) don't have statistics (unless you've played clever tricks), queries involving global temporary tables may show some surprise changes in execution plan after an upgrade.

Dictionary Stats

Although I don't think it's been stated explicitly in the manuals, there is a brief comment in the notes to the 9.2.0.5 patch set that you should delete the statistics on the data dictionary tables before installing the patch, and then re-create them afterwards—so it would seem that Oracle Corp. has no objection to your having statistics on the dictionary tables in 9.2.

Moreover, given the automatic statistics collection that takes place in 10*g*, you will find that you get statistics on the dictionary tables within 24 hours of creating your first 10*g* database.

If you have been running SQL against the data dictionary (e.g., from packages that dynamically generate new partitions for partitioned tables), then you may find that there is some benefit in having dictionary statistics in place. You may also find that such statistics introduce a performance problem when you upgrade. Don't forget to test your housekeeping code as part of your regression tests on upgrade.

APPENDIX B

Optimizer Parameters

There are lots of parameters that relate to optimization, and a number of them change their default value as you move from version to version. If you have a performance problem with a specific query after upgrading, it is always worth checking to see whether any of the parameters with new default values have names that seem be to related to the problem query.

For example, several people have complained about performance problems with executions plans that suddenly include the operation `bitmap conversion (from rowids)` after upgrading from 8*i* to 9*i*. Check the list of changes and you find that there is a parameter called `_b_tree_bitmap_plans` that has changed from `false` to `true`. Doesn't that sound like a good suspect? Perhaps you could test what happens if you set this back to `false`. (A better option appears in 10*g*, where you could use the `no_index_combine()` hint on the few queries which display the problem.)

There are two other sources of information about parameters that might affect the optimizer's calculations. One set of values appears in the `10053 trace` file; the other set appears (in Oracle 10*g* only) in the dynamic performance views `v$sys_optimizer_env`, `v$ses_optimizer_env`, or `v$sql_optimizer_env`. The views report optimizer-related parameters at the system level, the session level, and the individual child cursor level respectively—the last view also appears as a raw column called `optimizer_env` in `v$sql`.

Perhaps the best place you could go to investigate the effects of parameter changes on the optimizer is to one specific parameter: `optimizer_features_enable`. The particular benefit of this parameter is that it allows you to identify exactly those parameters relating to optimization that might be causing your upgrade performance problems.

optimizer_features_enable

If you take an Oracle 10.1.0.4 database, modify the parameter `optimizer_features_enable` to 8.1.7, 9.2.0, and 10.1.0 in turn, and capture the system parameters in a table, you can then query the table to see which other parameters change at the same time.

Table B-1 summarizes the 48 parameters (many of them hidden parameters) affected. Some of these parameters don't even exist in 9*i* or 8*i*.

Table B-1. *Parameters Affected by* `optimizer_features_enable` *(10g)*

Name	10.1.0.4	9.2.0.6	8.1.7.4
_always_anti_join	choose	choose	off ***
_always_semi_join	choose	choose	off ***
_b_tree_bitmap_plans	true	true	false
_complex_view_merging	true	true	false
_cost_equality_semi_join	true	true	false
_cpu_to_io	0	0	100 ***
_generalized_pruning_enabled	true	true	false
_gs_anti_semi_join_allowed	true	true	false
_index_join_enabled	true	true	false
_load_without_compile	none	none	none
_local_communication_costing_enabled	true	false ***	false
_new_initial_join_orders	true	true	false
_new_sort_cost_estimate	true	true	false
_optim_adjust_for_part_skews	true	true	false
_optim_new_default_join_sel	true	true	false
_optim_peek_user_binds	true	true	false
_optimizer_compute_index_stats	true	false	false
_optimizer_correct_sq_selectivity	true	false	false
_optimizer_cost_based_transformation	linear	off	off
_optimizer_cost_model	choose	choose	io
_optimizer_dim_subq_join_sel	true	false	false
_optimizer_join_order_control	3	0	0
_optimizer_join_sel_sanity_check	true	false	false
_optimizer_max_permutations	2000	2000	80000
_optimizer_new_join_card_computation	true	true	false
_optimizer_skip_scan_enabled	true	true	false
_optimizer_squ_bottomup	true	false	false
_optimizer_system_stats_usage	true	true	false
_optimizer_undo_cost_change	10.1.0	9.2.0	8.1.7
_ordered_nested_loop	true	true	false
_parallel_broadcast_enabled	true	true	false
_partition_view_enabled	true	false	false
_pre_rewrite_push_pred	true	true	false

Table B-1. *Parameters Affected by* `optimizer_features_enable` *(10*g*)*

Name	10.1.0.4	9.2.0.6	8.1.7.4
_pred_move_around	true	true	false
_push_join_predicate	true	true	false
_push_join_union_view	true	true	false
_push_join_union_view2	true	false	false
_query_rewrite_setopgrw_enable	true	false	false
_remove_aggr_subquery	true	false	false
_right_outer_hash_enable	true	false	false
_table_scan_cost_plus_one	true	true	false
_union_rewrite_for_gs	yes_gset_mvs	yes_gset_mvs***	off
_unnest_subquery	true	true	false
optimizer_dynamic_sampling	2	1	0
optimizer_features_enable	10.1.0	9.2.0	8.1.7
optimizer_mode	all_rows	choose	choose
query_rewrite_enabled	true	false	false
skip_unusable_indexes	true	false	false

**** There are some parameters where the default value of the parameter in the "real" version of the database does not agree with the value set by Oracle 10*g*. The differences are as follows:*

* *`_always_anti_join` defaults to `nested_loops`, not `off` in 8.1.7.4.*
* *`_always_semi_join` defaults to `standard`, not `off` in 8.1.7.4.*
* *`_cpu_to_io` defaults to `null`, not `100` in 8.1.7.4.*
* *`_union_rewrite_for_gs` defaults to `choose` not `yes_gset_mvs` in 9.2.0.6.*
* *`_local_communication_costing_enabled` defaults to `true`, not `false` in 9.2.0.6 (this parameter probably relates to RAC, rather than distributed queries. I have never been able to find any difference in costs between distributed queries and single database queries).*

The 10053 Trace File

When you enable the CBO trace (`event 10053`) at level 1, an early section of the trace file shows the list of optimizer-related parameters. This varies quite significantly from version to version, so I have included Table B-2 here, cross-referencing the values reported by the different versions.

I find it particularly useful to refer to this from time to time when upgrading a system, keeping a careful eye on the parameters that change from `false` to `true` (e.g., `unnest_subquery`). Sometimes a performance problem can appear because a new feature has been enabled—and this list may tell you which feature is the one that doesn't work well with your data distribution.

The driving feature of this list is the 10*g* parameter set. Any parameters that are hidden in 10*g* but not hidden in earlier versions have their values marked with ** in the columns of the versions where they used to be visible parameters.

Table B-2. *Cross-reference of Parameters Listed in the 10053 Trace File*

Name	10.1.0.4	9.2.0.6	8.1.7.4
_add_stale_mv_to_dependency_list	true		
_always_anti_join	choose	choose	
_always_semi_join	choose	choose	
_always_star_transformation	false	false	false
_b_tree_bitmap_plans	true	true	false
_bt_mmv_query_rewrite_enabled	true		
_complex_view_merging	true	true	false
_convert_set_to_join	false		
_cost_equality_semi_join	true		
_cpu_to_io	0	0	
_default_non_equality_sel_check	true	true	true
_disable_datalayer_sampling	false		
_disable_function_based_index	false		
_distinct_view_unnesting	false		
_dml_monitoring_enabled	true		
_eliminate_common_subexpr	true		
_enable_type_dep_selectivity	true	true	true
_fast_full_scan_enabled	true	true	true
_fic_area_size	131072		
_force_datefold_trunc	false		
_force_temptables_for_gsets (10*g*) _gsets_always_use_temptables (9*i*)	false	false	
_full_pwise_join_enabled	true		
_generalized_pruning_enabled	true		
_gs_anti_semi_join_allowed	true	true	
_hash_join_enabled	true	true **	true **
_hash_multiblock_io_count	0	0	0
_improved_outerjoin_card	true	true	true
_improved_row_length_enabled	true	true	true
_index_join_enabled	true	true	false
_left_nested_loops_random	true		
_like_with_bind_as_equality	false	false	

Table B-2. *Cross-reference of Parameters Listed in the 10053 Trace File (Continued)*

Name	10.1.0.4	9.2.0.6	8.1.7.4
_local_communication_costing_enabled	true		
_local_communication_ratio	50		
_minimal_stats_aggregation	true		
_mmv_query_rewrite_enabled	false		
_nested_loop_fudge	100	100	100
_new_initial_join_orders	true	true	false
_new_sort_cost_estimate	true	true	
_no_or_expansion	false	false	false
_oneside_colstat_for_equijoins	true	true	true
_optim_adjust_for_part_skews	true		
_optim_enhance_nnull_detection	true	true	true
_optim_new_default_join_sel	true		
_optim_peek_user_binds	true		
_optimizer_adjust_for_nulls	true	true	true
_optimizer_block_size	8192		
_optimizer_cache_stats	false		
_optimizer_cbqt_factor	50		
_optimizer_cbqt_no_size_restriction	true		
_optimizer_compute_index_stats	true		
_optimizer_correct_sq_selectivity	true		
_optimizer_cost_based_transformation	linear		
_optimizer_cost_filter_pred	false		
_optimizer_cost_model	choose	choose	
_optimizer_degree	0		
_optimizer_dim_subq_join_sel	true		
_optimizer_disable_strans_sanity_checks	0		
_optimizer_ignore_hints	false		
_optimizer_join_order_control	3		
_optimizer_join_sel_sanity_check	true		
_optimizer_max_permutations	2000	2000 **	80000 **
_optimizer_mjc_enabled	true		
_optimizer_mode_force	true	true	true

Table B-2. *Cross-reference of Parameters Listed in the 10053 Trace File (Continued)*

Name	10.1.0.4	9.2.0.6	8.1.7.4
_optimizer_new_join_card_computation	true		
_optimizer_percent_parallel	101	101	0 **
_optimizer_push_down_distinct	0		
_optimizer_push_pred_cost_based	true		
_optimizer_random_plan	0		
_optimizer_search_limit	5	5 **	5 **
_optimizer_skip_scan_enabled	true		
_optimizer_sortmerge_join_enabled	true		
_optimizer_squ_bottomup	true		
_optimizer_system_stats_usage	true		
_optimizer_undo_changes	false	false	false
_optimizer_undo_cost_change	10.1.0.4		
_or_expand_nvl_predicate	true	true	true
_ordered_nested_loop	true	true	false
_parallel_broadcast_enabled	true	true **	false **
_partial_pwise_join_enabled	true		
_partition_view_enabled	true	false **	false **
_pga_max_size	204800 KB		
_pre_rewrite_push_pred	true		
_pred_move_around	true	true	
_predicate_elimination_enabled	true		
_project_view_columns	true		
_push_join_predicate	true	true	false
_push_join_union_view	true	true	false
_push_join_union_view2	true		
_px_broadcast_fudge_factor	100		
_query_cost_rewrite	true	true	true
_query_rewrite_1	true		
_query_rewrite_2	true		
_query_rewrite_drj	true		
_query_rewrite_expression	true	true **	true **
_query_rewrite_fpc	true		
_query_rewrite_fudge	90		

Table B-2. *Cross-reference of Parameters Listed in the 10053 Trace File (Continued)*

Name	10.1.0.4	9.2.0.6	8.1.7.4
_query_rewrite_jgmigrate	true		
_query_rewrite_maxdisjunct	257		
_query_rewrite_or_error	false		
_query_rewrite_setopgrw_enable	true		
_query_rewrite_vop_cleanup	true		
_remove_aggr_subquery	true		
_right_outer_hash_enable	true		
_slave_mapping_enabled	true		
_smm_auto_cost_enabled	true		
_smm_auto_max_io_size	248 KB		
_smm_auto_min_io_size	56 KB		
_smm_max_size	10240 KB		
_smm_min_size	204 KB		
_smm_px_max_size	61440 KB		
_sort_elimination_cost_ratio	0	0	0
_sort_multiblock_read_count	2		
_sort_space_for_write_buffers	1		
_spr_push_pred_refspr	true		
_subquery_pruning_enabled	true	true	true
_subquery_pruning_mv_enabled	false		
_system_index_caching	0	0	
_table_scan_cost_plus_one	true	true	false
_union_rewrite_for_gs	yes_gset_mvs		
_unnest_subquery	true	true	false
_use_column_stats_for_function	true	true	true
active_instance_count	1		
bitmap_merge_area_size	1048576		
cpu_count	2		
cursor_sharing	exact		
db_file_multiblock_read_count	8	8	8
flashback_table_rpi	non_fbt		
hash_area_size	131072	2097152	2097152

Table B-2. *Cross-reference of Parameters Listed in the 10053 Trace File (Continued)*

Name	10.1.0.4	9.2.0.6	8.1.7.4
optimizer_dynamic_sampling	2	1	
optimizer_features_enable	10.1.0.4	9.2.0	8.1.7
optimizer_features_hinted	0.0.0		
optimizer_index_caching	0	0	0
optimizer_index_cost_adj	100	100	100
optimizer_mode	all_rows	choose	choose
optimizer_mode_hinted	false		
parallel_ddl_forced_degree	0		
parallel_ddl_forced_instances	0		
parallel_ddl_mode	enabled		
parallel_dml_forced_dop	0		
parallel_dml_mode	disabled		
parallel_execution_enabled	true		
parallel_query_forced_dop	0		
parallel_query_mode	enabled		
parallel_threads_per_cpu	2		
pga_aggregate_target	204800 KB		
query_rewrite_enabled	false	false	false
query_rewrite_integrity	enforced	enforced	enforced
skip_unusable_indexes	true		
sort_area_retained_size	0		
sort_area_size	65536	1048576	2097152
sqlstat_enabled	false		
star_transformation_enabled	false	false	false
statistics_level	typical		
workarea_size_policy	auto		

The following parameters are reported in the 8*i* and 9*i* trace files, exist in 10*g*, but are not reported in the 10*g* trace file:

```
_OPTIMIZER_CHOOSE_PERMUTATION = 0
_SUBQUERY_PRUNING_COST_FACTOR = 20
_SUBQUERY_PRUNING_REDUCTION_FACTOR = 50
_USE_NOSEGMENT_INDEXES = FALSE
```

The following parameter is reported only in the 9*i* trace file, exists in 10*g*, but is not reported in the 10*g* trace file:

```
_OPTIMIZER_DYN_SMP_BLKS = 32
```

The following parameter is reported only in the 9*i* trace file and does not exist in 10*g*:

```
_SORTMERGE_INEQUALITY_JOIN_OFF = FALSE
```

v$sql_optimizer_env

There is a short list of optimizer-related parameters, shown in Table B-3, held with every cursor in the SGA. This can be queried through view `v$sql_optimizer_env`. The same list exists at the session level (as `v$ses_optimizer_env`) and at the system level (as `v$sys_optimizer_env`). Given the long list of parameters that gets dumped into the `10053 trace`, it is a little strange that this list is so short. The values listed here are the defaults for 10.1.0.4 from a system with a single hyper-threading CPU, in the order they appear when you query the view.

Table B-3. *Parameters Recorded for Every Cursor*

Name	Value
parallel_execution_enabled	true
optimizer_features_enable	10.1.0.4
cpu_count	2
active_instance_count	1
parallel_threads_per_cpu	2
hash_area_size	131072
bitmap_merge_area_size	1048576
sort_area_size	65536
sort_area_retained_size	0
db_file_multiblock_read_count	8
pga_aggregate_target	204800 KB
parallel_query_mode	enabled
parallel_dml_mode	disabled
parallel_ddl_mode	enabled
optimizer_mode	all_rows
cursor_sharing	exact
star_transformation_enabled	false
optimizer_index_cost_adj	100
optimizer_index_caching	0

Table B-3. *Parameters Recorded for Every Cursor (Continued)*

Name	Value
query_rewrite_enabled	false
query_rewrite_integrity	enforced
workarea_size_policy	auto
optimizer_dynamic_sampling	2
statistics_level	typical
skip_unusable_indexes	true

Index

Symbols

A

B

C

D

E

F

G

H

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z