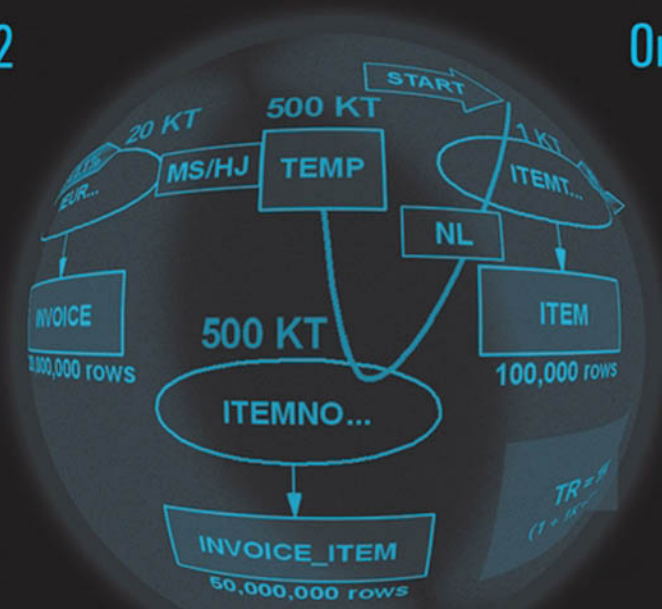
WILEY

Relational Database Index Design and the Optimizers

Tapio Lahdenmäki and Michael Leach

DB2

Oracle



SQL Server

et. al.

Relational Database Index Design and the Optimizers

Relational Database Index Design and the Optimizers

DB2, Oracle, SQL Server, et al.

Tapio Lahdenmäki
Michael Leach

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Lahdenmäki, Tapio.
Relational database index design and the optimizers : DB2, Oracle, SQL server et al / Lahdenmäki and Leach.
p. cm.
Includes bibliographical references and indexes.
ISBN-13 978-0-471-71999-1
ISBN-10 0-471-71999-4 (cloth)
1. Relational databases. I. Leach, Mike, 1942- II. Title.
QA76.9.D3L335 2005
005.75′65—dc22
2004021914

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Contents

Preface

Relational databases have been around now for more than 20 years. In their early days, performance problems were widespread due to limited hardware resources and immature optimizers, and so performance was a priority consideration. The situation is very different nowadays; hardware and software have advanced beyond all recognition. It's hardly surprising that performance is now assumed to be able to take care of itself! But the reality is that despite the huge growth in resources, even greater growth has been seen in the amount of information that is now available and what needs to be done with this information. Additionally, one crucial aspect of the hardware has not kept pace with the times: Disks have certainly become larger and incredibly cheap, but they are still relatively slow with regards to their ability to directly access data. Consequently many of the old problems haven't actually gone away—they have just changed their appearance. Some of these problems can have enormous implications—stories abound of "simple" queries that might have been expected to take a fraction of a second appear to be quite happy to take several minutes or even longer; this despite all the books that tell us how to code queries properly and how to organize the tables and what rules to follow to put the right columns into the indexes. So it is abundantly clear that there is a need for a book that goes beyond the usual boundaries and really starts to think about why so many people are still having so many problems today.

To address this need, we believe we must focus on two issues. First, the part of the relational system (called the SQL optimizer) that has to decide how to find the required information in the most efficient way, and secondly how the indexes and tables are then scanned. We want to try to put ourselves in the optimizer's place; perhaps if we understood why it might have problems, we might be able to do things differently. Fortunately it is quite surprising how little we really need to understand about the optimizers, but what there is though is remarkably important. Likewise, a very important way in which this book differs from other books in its field, is that we will not be providing a massive list of rules and syntax to use for coding SQL and designing tables or even indexes. This is not a reference book to show exactly which SQL WHERE clause should be used, or what syntax should be employed, for every conceivable situation. If we tried to follow a long list of complicated, ambiguous, and possibly incomplete instructions, we would be following all the others who have already trod the same path. If on the other hand we appreciate the impact of what we are asking the relational system to undertake and how we can influence that impact, we will be able to understand, control, minimize, or avoid the problems being encountered.

The second objective of this book is to show how we can use this knowledge to quantify the work being performed in terms of CPU and elapsed time. Only in this way can we truly judge the success of our index and table design; we need to use actual figures to show what the optimizer would think, how long the scans would take, and what modifications would be required to provide satisfactory performance. But most importantly, we have to be able to do this quickly and easily; this in turn means that it is vital to focus on the few really major issues, not on the relatively unimportant detail under which many people drown. This is key—to focus on a very few, crucially important areas—and to be able to say how long it would take or how much it would cost.

We have also one further advantage to offer, which again arises as a result of focusing on what really matters. For those who may be working with more than one relational product (even from the same vendor), instead of reading and digesting multiple sets of widely varying rules and recommendations, we are using a single common approach which is applicable to all relational products. All "genuine" relational systems have an optimizer that has the same job to do; they all have to make decisions and then scan indexes and tables. They all do these things in a startlingly similar way (although they have their own way of describing them). There are, of course, some differences between them, but we can handle this with little difficulty.

The audience for which this book is intended, is quite literally, anyone who feels it is to his or her benefit to know something about SQL performance or about how to design tables and indexes effectively, as well as those having a direct responsibility for designing indexes, anyone coding SQL statements as queries or as part of application programs, and those who are responsible for maintaining the relational data and the relational environment. All will benefit to a varying degree if they feel some responsibility for the performance effects of what they are doing.

Finally, a word regarding the background that would be appropriate to the readers of this book. A knowledge of SQL, the relational language, is assumed. A general understanding of computer systems will probably already be in place if one is even considering a book such as this. Other than that, perhaps the most important quality that would help the reader would be a natural curiosity and interest in how things work—and a desire to want to do things better. At the other extreme, there are also two categories of the large number of people with many years of experience in relational systems who might feel they would benefit; first those who have managed pretty well over the years with the detailed rule books and would like to relax a little more by understanding why these rules apply; second, those who have already been using the techniques described in this book for many years but who have not appreciated the implications that have been brought into play by the introduction of the new world hardware.

Most of the ideas and techniques used in this book are original and consequently few external references will be found to other publications and authors. On the other hand, as is always the case in the production of a book such as this,

we are greatly indebted to numerous friends and colleagues who have assisted in so many ways and provided so much encouragement. In particular we would like to thank Matti Ståhl for his detailed input and critical but extremely helpful advice throughout the development of the book; Lennart Henäng, Ari Hovi, Marja Kärmeniemi, Timo Raitalaakso for their invaluable assistance and reviews, and Akira Shibamiya for his original work on relational performance formulae. In addition we are indebted to scores of students and dozens of database consultants for providing an insight into their real live problems and solutions. Finally, a very special thanks go to Meta and Lyn without whose encouragement and support this book would never have been completed; Meta also brilliantly encapsulated the heart of the book in her special design for the bookcover. Solutions to the end-of-chapter exercises and other materials relating to this text can be found at this ftp address: ftp://ftp.wiley.com/public/sci_tech_med/relational_database/.

TAPIO LAHDENMÄKI
MICHAEL LEACH

Smlednik, Slovenia
Shrewsbury, England
April 2005

Chapter 1

Introduction

- To understand how SQL optimizers decide what table and index scans should be performed to process SQL statements as efficiently as possible
- To be able to quantify the work being done during these scans to enable satisfactory index design
- Type and background of audience for whom the book is written
- Initial thoughts on the major reasons for inadequate indexing
- Systematic index design.

ANOTHER BOOK ABOUT SQL PERFORMANCE!

Relational databases have been around now for over 20 years, and that's precisely how long performance problems have been around too—and yet here is *another* book on the subject. It's true that this book focuses on the index design aspects of performance; however, some of the other books consider this area to a greater or lesser extent. But then a lot of these books have been around for over 20 years, and the problems still keep on coming. So perhaps there is a need for a book that goes beyond the usual boundaries and starts to think about why so many people are still having so many problems.

It's certainly true that the world of relational database systems is a very complex one—it has to be if one reflects on what really has to be done to satisfy SQL statements. The irony is that the SQL is so beautifully simple to write; the concept of tables and rows and columns is so easy to understand. Yet we could be searching for huge amounts of information from vast sources of data held all over the world—and we don't even need to know where it is or how it can be found. Neither do we have to worry about how long it's going to take or how much it's going to cost. It all seems like magic. Maybe that's part of the problem—it's too easy; but then of course, it *should* be so easy.

We still recognize that problems will arise—and huge problems at that. Stories abound of "simple" queries that might have been expected to take a fraction of a second appear to be quite happy to take several minutes or even longer. But then, we have all these books, and they tell us how to code the query

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

properly and how to organize the table and what rules to follow to put the right columns into the index—and often it works. But we still seem to continue to have performance problems, despite the fact that many of these books are really very good, and their authors really know what they are talking about.

Of particular interest to us in this book is the part of the relational system (called the SQL optimizer) that *decides how to find all the information required in the most efficient way it can.* In an ideal world, we wouldn't even need to know it exists, and indeed most people are quite happy to leave it that way! Having made this decision, the optimizer directs scans of indexes and tables to find our data. In order to understand what's going through the optimizer's mind, we will also need to appreciate what is involved in these scans.

So what we want to do in this book is *first* to try to put ourselves in the optimizer's place; how it decides what table and index scans should be performed to process SQL statements as efficiently as possible. Perhaps if we understand *why* it might have problems, we could do things differently; not by simply following a myriad of incredibly complex rules that, even if we can understand them might or might not apply, but by understanding what it is trying to do.

A major concern that one might reasonably be expected to have on hearing this is that it would appear to be too complex or even out of the question. But it is quite surprising *how little we really need to understand*; what there is, though, is *incredibly important*.

Likewise, perhaps the first, and arguably the *most important, difference* this book has from other books in its field is that we will *not* be providing a massive list of rules and syntax to use for coding SQL and designing tables or even indexes. This is *not* a reference book to show exactly which SQL WHERE clause should be used, or what syntax should be employed, for every conceivable situation. If we try to follow a long list of complicated, ambiguous, and possibly even incomplete instructions, we will be following all the others who have already trod the same path. If on the other hand we understand the impact of what we are asking the relational system to undertake, and how we can influence that impact, we will be able to understand, avoid, minimize, and control the problems being encountered.

A *second* objective of this book is to show how we can use this knowledge to *quantify* the work being performed. Only in this way can we truly judge the success of our index design; we need to be able to use *actual figures* to show what the optimizer would think, how long the scans would take, and what modifications would be required to provide satisfactory performance. But most importantly, we have to be able to do this *quickly and easily*; this in turn means that it is vital to focus on a few major issues, not on the relatively unimportant detail under which many people drown. This is key—to *focus on a very few, crucially important issues*—and to be able to say *how long it would take* or *how much it would cost*.

We have also one further advantage to offer, which again arises as a result of focusing on what really matters. For those who may be working with more than one relational product (even from the same vendor), instead of needing to read

and digest multiple sets of widely varying rules and recommendations, we are using a single common approach that is applicable to *all* relational products. All "genuine" relational systems have an optimizer that has the same job to do; they all have to scan indexes and tables. They all do these things in a startlingly similar way (although they have their own way of describing them). There are, of course, *some* differences between them, but we can handle this with little difficulty.

It is for exactly the same reason that the *audience* for which this book is intended is, quite literally, *anyone* who feels it is to his or her benefit to know something about SQL performance or about how to design indexes effectively. Those having a direct responsibility for designing indexes, anyone coding SQL statements as queries or as part of application programs, and those who are responsible for maintaining the relational data and the relational environment will all benefit to a varying degree if they feel some responsibility for the performance effects of what they are doing.

Finally, a word regarding the *background* that would be appropriate to the readers of this book. A knowledge of SQL, the relational language, is assumed; fortunately this knowledge can easily be obtained from the wealth of material available today. A general understanding of computer systems will probably already be in place if one is *even considering* a book such as this. Other than that, perhaps the most important quality that would help the reader would be a natural curiosity and interest in how things work—and a desire to want to do things better. At the other extreme, there are also two categories of the many people with well over 20 years of experience in relational systems who might feel they would benefit; first, those who have managed pretty well over the years with the detailed rule books and would like to relax a little more by understanding why these rules apply; second, those who have already been using the techniques described in this book for many years. The reason why *they* may well be interested *now* is that over the years hardware has progressed beyond all recognition. The problems of yesteryear are no longer the problems of today. *But still the problems keep on coming!*

We will begin our discussion by reflecting on why, so often, indexing is still the source of so many problems.

INADEQUATE INDEXING

Important Note

The following discussion makes use of some concepts and terminology used in relational database systems and disk subsystems. If the reader has little or no background in these areas, this chapter may be read at this time at a superficial level only, bypassing some of the more technical details that are provided to justify the statements and conclusions made. More detail on these areas will be found in Chapters 2, 3, and 4.

For many years, inadequate indexing has been the most common cause of performance disappointments. The most widespread problem appears to be that indexes do not have sufficient columns to support all the predicates of a WHERE clause. Frequently, there are not enough indexes on a table; some SELECTs may have no useful index; sometimes an index has the right columns but in the wrong order.

It is relatively easy to improve the indexing of a relational database because no program changes are required. However, a change to a production system always carries some risk. Furthermore, while a new index is being created, update programs may experience long waits because they are not able to update a table being scanned for a CREATE INDEX. For these reasons, and, of course, to achieve acceptable performance from the first production day of a new application, indexing should be in fairly good shape *before* production starts. Indexing should then be finalized soon after cutover, without the need for numerous experiments.

Database indexes have been around for decades, so why is the average quality of indexing still so poor? One reason is perhaps because many people assume that, with the huge processing and storage capacity now available, it is no longer necessary to worry about the performance of seemingly simple SQL. Another reason may be that few people even think about the issue at all. Even then, for those who do, the fault can often be laid at the door of numerous relational database textbooks and educational courses. Browsing through the library of relational database management system (DBMS) books will quite possibly lead to the following assessment:

- The *index design* topics are short, perhaps only a few pages.
- The negative side effects of indexes are emphasized; indexes consume disk space and they make inserts, updates, and deletes slower.
- Index design guidelines are vague and sometimes questionable. Some writers recommend indexing all restrictive columns. Others claim that index design is an art that can only be mastered through trial and error.
- Little or no attempt is made to provide a simple but effective approach to the whole process of index design.

Many of these warnings about the cost of indexes are a legacy from the 1980s when storage, both disk and semiconductor, was *significantly* more expensive than it is today.

MYTHS AND MISCONCEPTIONS

Even recent books, such as one published as late as 2002 (1), suggest that only the *root page* of a B-tree index will normally stay in memory. This was an appropriate assumption 20 years ago, when memory was typically so small that the database buffer pool could contain only a few hundred pages, perhaps less than a megabyte. Today, the size of the database buffer pools may be hundreds

of thousands of pages, one gigabyte (GB) or more; the read caches of disk servers are typically even larger—64 GB, for instance. Although databases have grown as disk storage has become cheaper, it is now realistic to assume that *all the nonleaf pages* of a B-tree index will usually remain in memory or the read cache. Only the leaf pages will normally need to be read from a disk drive; this, of course, makes index maintenance much faster.

The assumption *only root pages stay in memory* leads to many obsolete and dangerous recommendations, of which the following are just a few examples.

Myth 1: No More Than Five Index Levels

This recommendation is often made in relational literature, usually based on the assumption that *only root pages stay in memory*. With current processors even when all nonleaf pages are in the database buffer pool, *each index level* could add as much as 50 microseconds (μs) of central processing unit (CPU) time to an index scan. If a nonleaf page is *not* in the database buffer pool, but *is* found in the read cache of the disk server, the elapsed time for reading the page may be about 1 millisecond (ms). These values should be contrasted with the time taken by a random read from a disk drive, perhaps 10 ms. To see what this effectively means, we will take a simple illustration.

The index shown in Figure 1.1 corresponds to a 100-million-row table. There are 100 million index rows with an average length of 100 bytes. Taking the distributed free space into account, there are 35 index rows per leaf page. If the DBMS does not truncate the index keys in the nonleaf pages, the number of index entries in these pages is also 35.

The probable distribution of these pages as shown in Figure 1.1, together with their size, can be deduced as follows:

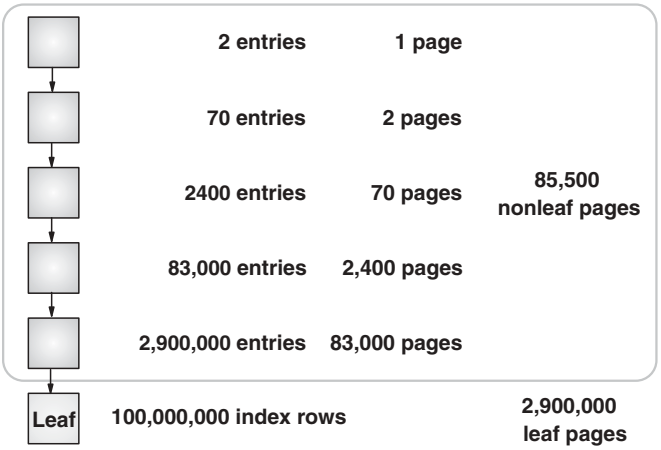


Figure 1.1 Large index with six levels.

- The *index in total* holds about 3,000,000 4 K pages, which requires 12 GB of disk space.
- The total size of the *leaf pages* is $2{,}900{,}000 \times 4$ K, which is almost 12 GB. It is reasonable to assume that these will normally be read from a disk drive (10 ms).
- The size of the *next level* is $83{,}000 \times 4$ K, which is 332 megabytes (MB); if the index is *actively used*, then these pages may stay in the read cache (perhaps 64 GB in size) of the disk server, if not in the database buffer pool (say 4 GB for index pages).
- The *upper levels*, roughly $2500 \times 4\ \text{K} = 10$ MB, will almost certainly remain in the database buffer pool.

Accessing *any* of these 100,000,000 index rows in this six-level index will then take between 10 and 20 ms. This is true even if many index rows have been added and the index is disorganized, but more about this in Chapter 11. Consequently, it makes little sense to set arbitrary limits to the number of levels.

Myth 2: No More Than Six Indexes per Table

In its positive attitude toward indexes, the *Oracle SQL Tuning Pocket Reference* (2) by Mark Gurry is an agreeable exception to the comments made earlier. As the title implies, the book focuses on helping the Oracle 9i optimizers, but it also criticizes standards that set an upper limit for the number of indexes per table on page 63:

> *I have visited sites which have a standard in place that no table can have more than six indexes. This will often cause almost all SQL statements to run beautifully, but a handful of statements to run badly, and indexes can't be added because there are already six on the table.*
>
> . . .
>
> *My recommendation is to avoid rules stating a site will not have any more than a certain number of indexes.*
>
> . . .
>
> *The bottom line is that all SQL statements must run acceptably. There is ALWAYS a way to achieve this. If it requires 10 indexes on a table, then you should put 10 indexes on the table.*

Myth 3: Volatile Columns Should Not Be Indexed

Index rows are held in key sequence, so when one of the columns is updated, the DBMS *may* have to move the corresponding row from its old position in the index to its new position, to maintain this sequence. This new position may be in the *same* leaf page, in which case only the one page is affected. However, particularly if the modified key is the first or only column, the new index row may have to be moved to a *different* leaf page; the DBMS must then update two leaf pages. *Twenty years ago*, this might have required *six* random disk reads if

the index had four levels; three for the original, two nonleaf and one leaf, together with a further three for the new. When a random disk read took 30 ms, moving one index row could add 6×30 ms $= 180$ ms to the response time of the update transaction. It is hardly surprising that volatile columns were seldom indexed.

These days when *three levels* of a four-level index, the nonleaf pages, stay in memory and a random read from a disk drive takes 10 ms, the corresponding time becomes 2×10 ms $= 20$ ms. Furthermore, many indexes are multicolumn indexes, called *compound* or *composite* indexes, which often contain columns that make the index key unique. When a volatile column is the last column of such an index, updating this volatile column *never* causes a move to another leaf page; consequently, with current disks, updating the volatile column adds only 10 ms to the response time of the update transaction.

Example

A few years ago, the DBAs of a well-tuned DB2 installation having an average local response time of *0.2 s*, started transaction-level exception monitoring. Immediately, they noticed that a simple browsing transaction regularly took more than *30 s*; the longest observed local response time was *a couple of minutes*. They quickly traced the problem to inadequate indexing on a 2-million-row table. Two problems were diagnosed:

- A volatile column STATUS, updated up to twice a second, was absent from the index, although it was an *obvious essential requirement*. A predicate using the column STATUS was ANDed to five other predicates in the WHERE clause.
- An ORDER BY required a sort of the result rows.

These two index design decisions had been made consciously, based on widely used recommendations. The column STATUS was much more volatile than most of the other columns in this installation. This is why the DBAs had not dared to include it in the index. They were also afraid that an extra index, which would have eliminated the sort, would have caused problems with INSERT performance because the insert rate to this table was relatively high. They were particularly worried about the *load* on the disk drive.

Following the realization of the extent of the problem caused by these two issues, rough estimates of the index overhead were made, and they decided to create an additional index containing the five columns, together with STATUS at the end. This new index solved both problems. The longest observed response time went down *from a couple of minutes to less than a second*. The UPDATE and INSERT transactions were not compromised and the disk drive containing the new index was not overloaded.

Disk Drive Utilization

Disk drive load and the required speed of INSERTs, UPDATEs, and DELETEs still set an upper limit to the number of indexes on a table. However, this ceiling

is much higher than it was 20 years ago. A reasonable request for a new index should not be rejected intuitively. With current disks, an indexed volatile column may become an issue only if the column is updated perhaps *more than 10 times a second*; such columns are not very common.

SYSTEMATIC INDEX DESIGN

The first attempts toward an index design method originate from the 1960s. At that time, textbooks recommended a matrix for predicting how often each field (column) is read and updated and how often the records (rows) containing these fields are inserted and deleted. This led to a list of columns to be indexed. The indexes were generally assumed to have only a single column, and the objective was to minimize the number of disk input/outputs (I/Os) during peak time. It is amazing that this approach is still being mentioned in recent books, although a few, somewhat more realistic writers, do admit that the matrix should only cover the most common transactions.

This *column activity matrix approach* may explain the column-oriented thinking that can be found even in recent textbooks and database courses, such as *consider indexing columns with these properties* and *avoid indexing columns with those properties*.

In the 1980s, the *column-oriented approach* began to lose ground to a *response-oriented approach*. Enlightened DBAs started to realize that the objective of indexing should be to make *all* database calls fast enough, given the hardware capacity constraints. The pseudo-relational DBMS of IBM S/38 (later AS/400, then the iSeries) was the vanguard of this attitude. It automatically built a good index for each database call. This worked well with simple applications. Today, many products propose indexes for each SQL call, but indexes are not created automatically, apart from primary key indexes and, sometimes, foreign key indexes.

As applications became more complex and databases much larger, the importance and complexity of index design became obvious. Ambitious projects were undertaken to develop tools for automating the design process. The basic idea was to collect a sample of production workload and then generate a set of index candidates for the SELECT statements in the workload. Simple evaluation formulas or a cost-based optimizer would then be used to decide which indexes were the most valuable. This sort of product has become available over the last few years but has spread rather slower than expected. Possible reasons for this are discussed in Chapter 16.

Systematic index design consists of two processes as shown in Figure 1.2. First, it is necessary to find the SELECTs that are, or will be, too slow with the current indexes, at least with the worst input; for example, “the largest customer” or “the oldest date”. Second, indexes have to be designed to make the slow SELECTs fast enough without making other SQL calls noticeably slower. Neither of these tasks is trivial.

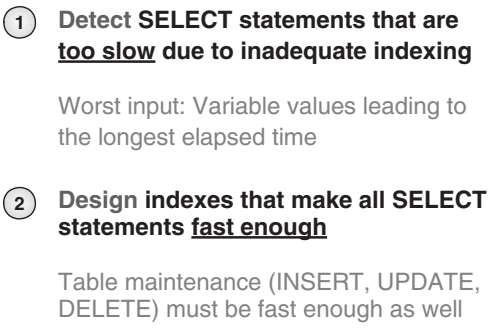


Figure 1.2 Systematic index design.

The first attempts to detect inadequate indexing at *design time* were based on hopelessly complex prediction formulas, sometimes simplified versions of those used by cost-based optimizers. Replacing calculators with programs and graphical user interfaces did not greatly reduce the effort. Later, extremely simple formulas, like the QUBE, developed in IBM Finland in the late 1980s, or a simple estimation of the number of random I/Os were found useful in real projects. The Basic Question proposed by Ari Hovi was the next and probably the ultimate step in this process. These two ideas are discussed in Chapter 5 and widely used throughout this book.

Methods for improving indexes *after* production cutover developed significantly in the 1990s. Advanced monitoring software forms a necessary base to do this, but an intelligent way to utilize the massive amounts of measurement data is also essential.

This second task of systematic index design went unrecognized for a long time. The SELECTs found in textbooks and course material were so unrealistically simple that the best index was usually obvious. Experience with real applications has taught, however, that even harmless looking SELECTs, particularly joins, often have a huge number of reasonable indexing alternatives. Estimating each alternative requires far too much effort, and measurements even more so. On the other hand, even experienced database designers have made numerous mistakes when relying on intuition to design indexes.

This is why there is a need for an algorithm to design the best possible index for a given SELECT. The concepts of a three-star index and the related index candidates, which are considered in Chapter 4, have proved helpful.

There are numerous success stories regarding the application of these simple, manual index design algorithms. It is not uncommon to see the elapsed times of SELECT calls being reduced by two orders of magnitude; from well over a minute down to well under a second, for instance, with relatively little effort, perhaps from as little as 5 or 10 min with the methods recommended in Chapters 4, 5, 7, and 8.

Chapter 2

Table and Index Organization

- The physical organization of indexes and tables
- The structure and use of the index and table pages, index and table rows, buffer pools, and disk cache
- The characteristics of disk I/Os, random and sequential
- Assisted random and sequential reads: skip-sequential, list prefetch, and data block prefetching
- The significance of synchronous and asynchronous I/Os
- The similarities and differences between database management systems
- Pages and table clustering, index rows, index-only tables, and page adjacency
- The very confusing but important issue of the term *cluster*
- Alternatives to B-tree indexes
- Bitmap indexes and hashing

INTRODUCTION

Before we are in a position to discuss the index design process, we need to understand how indexes and tables are organized and used. Much of this, of course, will depend on the individual relational DBMS; however, these all rely on broadly similar general structures and principles, albeit using very different terminology in the process.

In this chapter we will consider the fundamental structures of the relational objects in use; we will then discuss the performance-related issues of their use, such as the role of buffer pools, disks and disk servers, and how they are used to make the data available to the SQL process.

Once we are familiar with these fundamental ideas, we will be in a position, in the next chapter, to consider the way these relational objects are processed to satisfy SQL calls.

This chapter is merely an introduction. Considerably more detail will be provided throughout the book at a time when it is more appropriate. At the end

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

of the book, a glossary is provided that summarizes all the terms used throughout the text.

Index and Table Pages

Index and table rows are grouped together in *pages*; these are often 4K in size, this being a rather convenient size to use for most purposes, but other page sizes may be used. Fortunately, as far as index design is concerned, this is not an important consideration other than that the page size will determine the number of index and table rows in each page and the number of pages involved. To cater for new rows being added to tables and indexes, a certain proportion of each page may be left free when they are loaded or reorganized. This will be considered later.

Buffer pools and I/O activity (discussed later) are based on pages; for example, an entire page will be read from disk into a buffer pool. This means that *several rows*, not just one, are read into the buffer pool with a single I/O. We will also see that *several pages* may be read into the pool by just one I/O.

INDEX ROWS

An index row is a *useful concept* when evaluating access paths. For a *unique* index, such as the primary key index CNO on table CUST, it is equivalent to an index entry in the leaf page (see Fig. 2.1); the column values are copied from the table to the index, and a pointer to the table row added. Usually, the table *page number* forms a part of this pointer, something that should be kept in mind for a later time. For a *nonunique* index, such as the index CITY on table CUST, the index rows for a particular index value should be visualized as *individual* index entries, each having the same CITY value, but followed by a different pointer value. What is *actually stored* in a nonunique index is, in most cases, one CITY value followed by several pointers. The reason why it is *useful* to visualize these as individual index entries will become clear later.

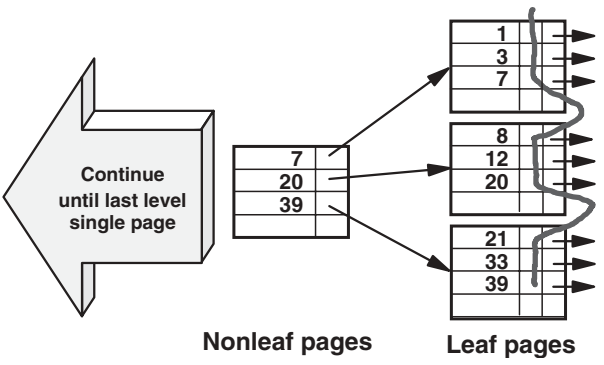


Figure 2.1 Very small index.

INDEX STRUCTURE

The nonleaf pages always contain a (possibly truncated) key value, the highest key together with a pointer, to a page at the next lower level, as shown in Figure 2.1. Several index levels may be built up in this way, until there is only a single page, called the *root page*, at the top of the index structure. This type of index is called a B-tree index (a balanced tree index) because the same number of nonleaf pages are required to find each index row.

TABLE ROWS

Each index row shown in Figure 2.1 points to a corresponding row in the table; the pointer usually identifies the page in which the row resides together with some means of identifying its position within the page. Each table row contains some control information to define the row and to enable the DBMS to handle insertions and deletions, together with the columns themselves.

The sequence in which the rows are positioned in the table, as a result of a table load or row inserts, *may* be defined so as to be the same as that of *one* of its indexes. In this case, as the index rows are processed, one after another in key sequence, so the corresponding table rows will be processed, one after another in the same sequence. Both index and table are then accessed in a sequential manner that, as we will see shortly, is a very efficient process.

Obviously, *only one of the indexes* can be defined to determine the sequence of the table rows in this way. If the table is being accessed via *any other* index, as the index rows are processed, one after another in key sequence, the corresponding rows will *not* be held in the table in the same sequence. For example, the first index row may point to page 17, the next index row to page 2, the next to page 85, and so forth. Now, although the index is still being processed sequentially and efficiently, the table is being processed randomly and much less efficiently.

BUFFER POOLS AND DISK I/OS

One of the primary objectives of relational database management systems is to ensure that data from tables and indexes is readily available when required. To enable this objective to be achieved as far as possible buffer pools, held in memory, are used to minimize disk activity. Each DBMS may have several pools according to the type, table or index, and the page size. Each pool will be large enough to hold many pages, perhaps hundreds of thousands of them. The buffer pool managers will attempt to ensure that frequently used data remains in the pool to avoid the necessity of additional reads from disk. How effective this is will be extremely important with respect to the performance of SQL statements, and so will be equally important for the purposes of this book. We will return to this subject on many occasions where the need

arises. For now we must simply be aware of the *relative costs* involved in accessing index or table rows from pages that may or may not be stored in the buffer pools.

Reads from the DBMS Buffer Pool

If an index or table page is found in the buffer pool, the only cost involved is that of the processing of the index or table rows. This is highly dependent on whether the row is rejected or accepted by the DBMS, the former incurring very little processing, the latter incurring much more as we will see in due course.

Random I/O from Disk Drives

Figure 2.2 shows the enormous cost involved in having to wait for a page to be read into the buffer pool from a disk drive.

Again, we must remember that a page will contain several rows; we may be interested in all of these rows, just a few of them, or even only a single row—the cost will be the same, roughly 10 ms. If the disk drives are heavily used, this figure might be considerably increased as a result of having to wait for the disk to become available. In computing terms, 10 ms is an eternity, which is why we will be so interested in this activity throughout this book.

It isn't really necessary to understand how this 10 ms is derived, but for those readers who like to understand where numbers such as this come from, Figure 2.3 breaks it down into its constituent components. From this we can see that we are assuming the disk would actually be busy for about 6 out of the 10 ms. The transfer time of roughly 1 ms refers to the movement of the page from the disk server cache into the database buffer pool. The other 3 ms is an estimate of the queuing time that might arise, based on disk activity of, say, 50 reads per second. These sort of figures would equally apply to directly attached drives; all the figures will, of course, vary somewhat, but we simply need to keep in mind a rough, but not unreasonable, figure of 10 ms.

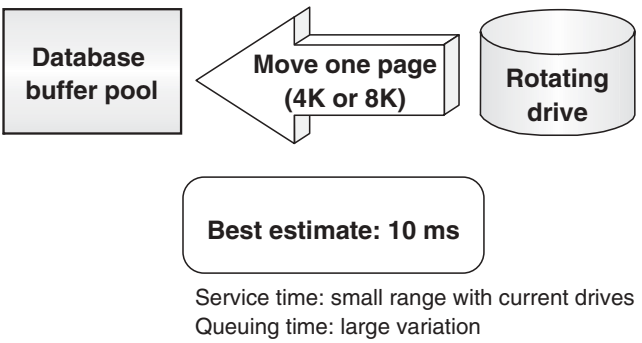


Figure 2.2 Random I/O from disk drive—1.

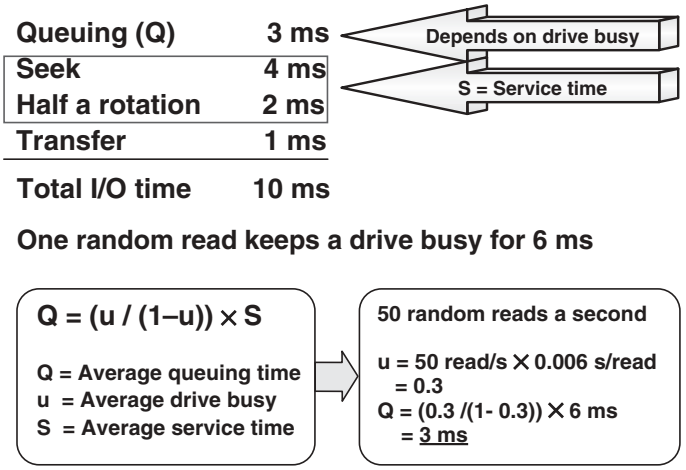


Figure 2.3 Random I/O from disk drive—2.

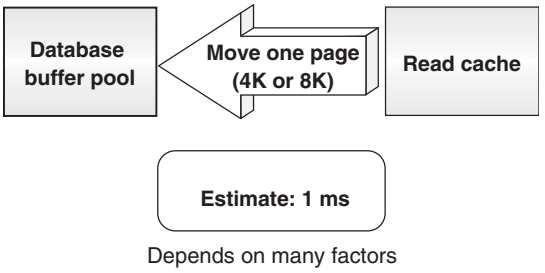


Figure 2.4 Read from disk server cache.

Reads from the Disk Server Cache

Fortunately, disk servers in use today provide their own memory (or cache) in order to reduce this huge cost in terms of elapsed time. Figure 2.4 shows the read of a single table or index page (again equivalent to reading a number of table or index rows) from the cache of the disk server. Just as with the buffer pools, the disk server is trying to hold frequently used data in memory (cache) rather than incurring the heavy disk read cost. If the page required by the DBMS is not in the buffer pool, a read is issued to the disk server who will check to see if it is in the server cache and only perform a read from a disk drive if it is not found there. The figure of 10 ms may be considerably reduced to a figure as low as 1 ms if the page is found in the disk server read cache.

In summary then, the ideal place for an index or table page to be when it is requested is in the database buffer pool. If it is not there, the next best place for it to be is in the disk server read cache. If it is in neither of these, a slow read from disk will be necessary, perhaps involving a long wait for the device to become available.

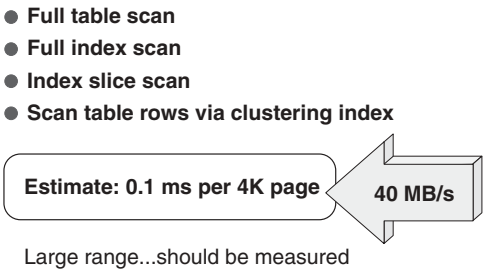


Figure 2.5 Sequential reads from disk drives.

Sequential Reads from Disk Drives

So far, we have only considered reading a single index or table page into the buffer pool. There will be many occasions when we actually want to read several pages into the pool and process the rows in sequence. Figure 2.5 shows the four occasions when this will apply. The DBMS will be aware that several index or table pages should be read sequentially and will identify those that are not already in the buffer pool. It will then issue multiple-page I/O requests, where the number of pages in each request will be determined by the DBMS; only those pages not already in the buffer pool will be read because those that are already in the pool may contain updated data that has not yet been written back to disk.

There are two very important advantages to reading pages sequentially:

- Reading several pages together means that the time per page will be reduced; with current disk servers, the value may be as low as 0.1 ms for 4K pages (40 MB/s).
- Because the DBMS knows in advance which pages will be required, the reads can be performed before the pages are actually requested; this is called prefetch.

The terms *index slice* and *clustering index* referred to in Figure 2.5 will be addressed shortly. Terms used to refer to the sequential reads described above include *Sequential Prefetch*, *Multi-Block I/Os*, and *Multiple Serial Read-Ahead Reads*.

Assisted Random Reads

We have seen how heavy the cost of random reads can be, and how buffer pools and disk caches can help to minimize this cost. There are in addition other occasions where the cost can be reduced, sometimes naturally, sometimes deliberately invoked by the optimizer. From an educational point of view, it will be highly desirable to use a single term to represent these facilities—*assisted random reads*. Please note that this term is *not* one that is used by any of the DBMSs.

Automatic Skip-Sequential

By definition, an access pattern will be skip-sequential if a set of noncontiguous rows are scanned in one direction. The I/O time per row will thus be automatically shorter than with random access; the shorter the skips, the greater the benefit. This would occur, for example, when table rows are being read via a clustering index and index screening takes place, as we will see in due course. This benefit can be enhanced in two ways:

1. The disk server may notice that access to a drive is taking place sequentially, or almost sequentially, and starts to read several pages ahead.
2. The DBMS may notice that a SELECT statement is accessing the pages of an index or table sequentially, or almost sequentially, and starts to read several pages ahead; this is called dynamic prefetch in DB2 for z/OS.

List Prefetch

In the previous example, this benefit was achieved simply as a result of the table and index rows being in the same sequence. DB2 for z/OS is in fact able to *create* skip-sequential access even when this is *not* the case; to do this, it has to access *all* the qualifying index rows and sort the pointers into table page sequence before accessing the table rows. Figures 2.6 and 2.7 contrast an access path that does *not* use list prefetch with one that does, the numbers indicating the sequence of events.

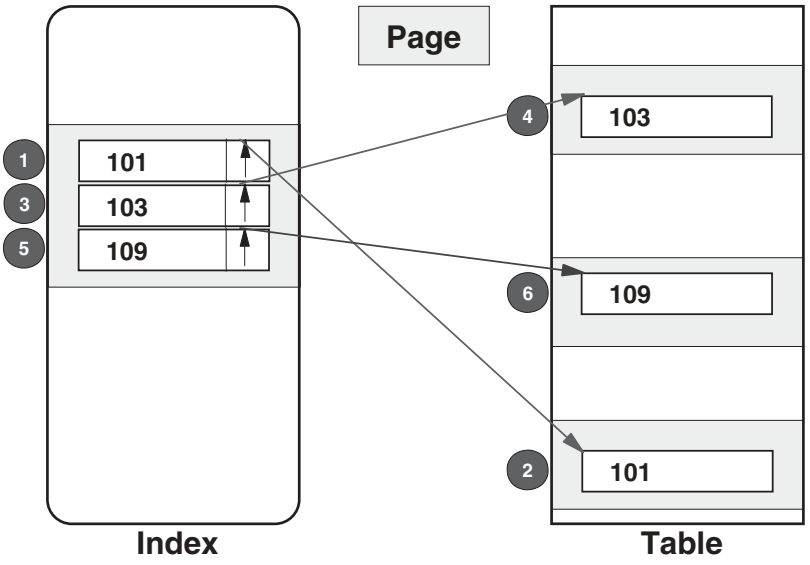


Figure 2.6 Traditional index scan.

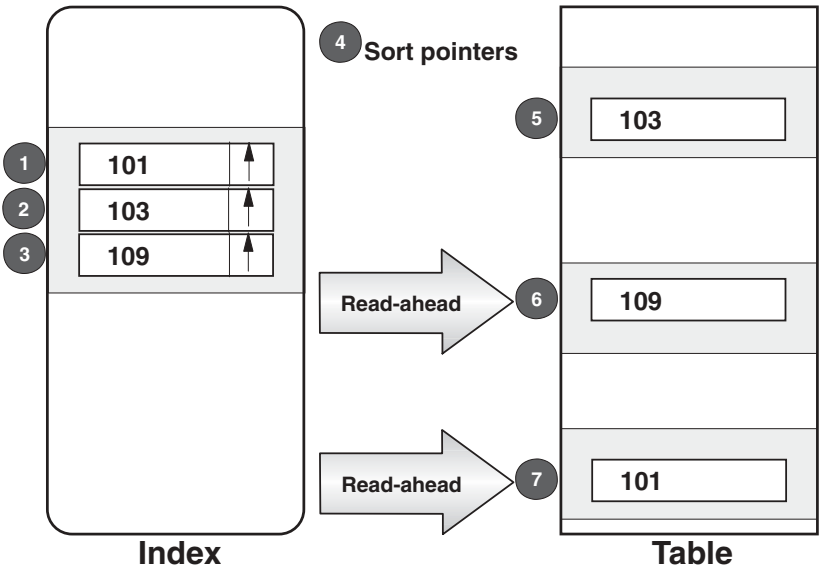


Figure 2.7 DB2 List Prefetch.

Data Block Prefetching

This feature is used by Oracle, again when the table rows being accessed are *not* in the same sequence as the index rows. In this case, however, as shown in Figure 2.8, the pointers are collected from the index slice and multiple random I/Os are started to read the table rows in parallel. If the table rows represented by steps 4, 5, and 6 reside on three different drives, all three random I/Os will be performed in parallel. As with list prefetch, we could use Figures 2.6 and 2.8 to contrast an access path that does *not* use data block prefetching with one that does.

Before we leave assisted random reads, it might be worth considering the order in which a result set is obtained. An index could provide the correct sequence automatically, whereas the above facilities could destroy this sequence before the table rows were accessed, thereby requiring a sort.

Comment

Throughout this book, we will refer to three types of read I/O operations: synchronous, sequential, and assisted random reads; in order to make the estimation process usable, initially only the first two types will be addressed, but Chapter 15 will discuss assisted random read estimation in some detail.

Note that SQL Server uses the term *Index Read-Ahead* and Oracle uses the term *Index Skip Scan*. The former refers to the reading-ahead of the next leaf pages following leaf page splits, while the latter refers to the reading of several index slices instead of doing a full index scan.

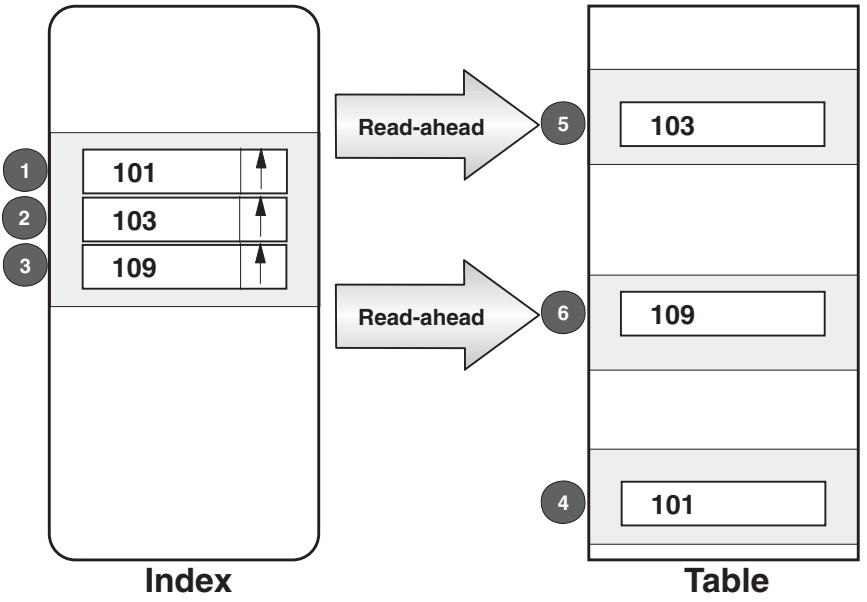


Figure 2.8 Oracle data block prefetching.

Assisted Sequential Reads

When a large table is to be scanned, the optimizer may decide to activate parallelism; for instance, it may split a cursor into several range-predicate cursors, each of which would scan one slice. When several processors and disk drives are available, the elapsed time will be reduced accordingly. Again we will put this to one side until we come to Chapter 15. Please note that the term *assisted sequential reads* is again not one that is used by any of the DBMSs.

Synchronous and Asynchronous I/Os

Having discussed these different access techniques, it will be appropriate now to ensure we fully appreciate one final consideration, synchronous and asynchronous I/Os as shown in Figure 2.9.

The term *synchronous* I/O infers that while the I/O is taking place, the DBMS is not able to continue any further; it is forced to wait until the I/O has completed. With a synchronous read, for example, we have to identify the row required (shown as "C" to represent the first portion of CPU time in the figure), access the page and process the row (shown as the second portion of CPU time), each stage waiting until the previous stage completes.

Asynchronous reads on the other hand are being performed in advance while a *previous* set of pages are being processed; there may be a considerable overlap between the processing and I/O time; ideally the asynchronous I/O will complete

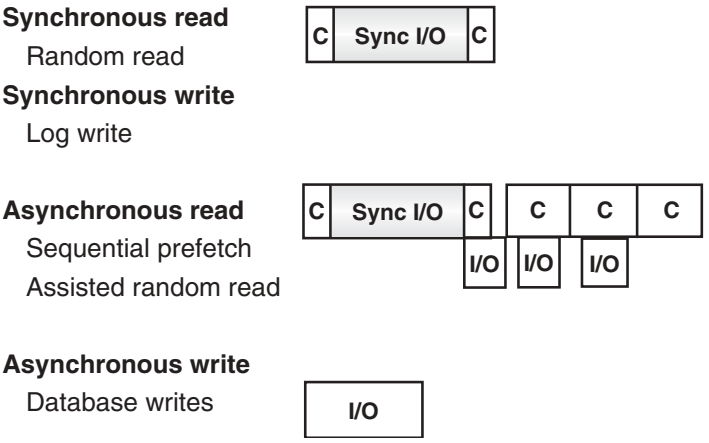


Figure 2.9 Synchronous and asynchronous I/O.

before the pages are actually required for processing. Each group of pages being prefetched and processed in this way is shown in Figure 2.9; note that a synchronous read kick-starts the whole prefetch activity before the first group of pages is prefetched to minimize the first wait.

When the DBMS requests a page, the disk system may read the next few pages as well into a disk cache (anticipating that these may soon be requested); this could be the rest of the stripe, the rest of the track, or even several stripes (striping is described shortly). We call this *Disk Read Ahead.*

Most database writes are performed asynchronously such that they should have little effect on performance. The main impact they do have is to increase the load on the disk environment, which in turn may affect the performance of the read I/Os.

HARDWARE SPECIFICS

At the time of writing, the disk drives used in database servers do not vary much with regard to their performance characteristics. They run at 10,000 or 15,000 rotations per minute and the average seek time is 3 or 4 ms. Our suggested estimate for an average random read from a disk drive (10 ms)—including drive queuing and the transfer time from the server cache to the pool—is applicable for *all* current disk systems.

The time for a sequential read, on the other hand, varies according to the configuration. It depends not only on the bandwidth of the connection (and eventual contention), but also on the degree of parallelism that takes place. RAID striping provides potential for parallel read ahead for a single thread. It is *strongly* recommended that the sequential read speed in an environment is measured before using our suggested figure of 0.1 ms per 4K page (refer to Chapter 6).

In addition to the I/O time estimates, the cost of disk space and memory influences index design.

Local disk drives provide physical data storage without the additional function provided by disk servers (such as fault tolerance, read and write cache, striping, and so forth), for a very low price.

Disk servers are computers with several processors and a large amount of memory. The most advanced disk servers are fault tolerant: All essential components are duplicated, and the software supports a fast transfer of operations to a spare unit. A high-performance fault tolerant disk server with a few terabytes may cost $2 million. The cost per gigabyte, then, is in the order of U.S.$500 (purchase price) or U.S.$50 per month (outsourced hardware).

Both local disks and disk servers employ industry-standard disk drives. The largest drives lead to the lowest cost per gigabyte; for example, a 145-GB drive costs much less than eight 18-GB drives. Unfortunately, they also imply much longer queuing times than smaller drives with a given access density (I/Os per gigabyte per second).

The cost of memory has been reduced dramatically over the last few years as well. A gigabyte of random access memory (RAM) for Intel servers (Windows and Linux) now costs about $500 while the price for RISC (proprietary UNIX and Linux) and mainframe servers (z/OS and Linux) is on the order of U.S.$10,000 per gigabyte. With 32-bit addressing, the maximum size of a database buffer pool might be a gigabyte (with Windows servers, for example), and a few gigabytes for mainframes that have several address spaces for multiple buffer pools. Over the next few years, 64-bit addressing, which permits *much* larger buffer pools, will probably become the norm. If the price for memory (RAM) keeps falling, database buffer pools of 100 gigabytes or more will then be common.

The price for the read cache of disk servers is comparable to that of RISC server memory. The main reason for buying a 64-GB read cache instead of 64 GB of server memory is the inability of 32-bit software to exploit 64 GB for buffer pools.

Throughout this book, we will use the following cost assumptions:

CPU time	$1000 per hour, based on 250 mips per processor
Memory	$1000 per gigabyte per month
Disk space	$50 per gigabyte per month

These are the possible current values for outsourced mainframe installations. Each designer should, of course, ascertain his or her own values, which may be *very much lower* than the above.

DBMS SPECIFICS

Pages

The size of the table pages sets an upper limit to the length of table rows. Normally, a table row must fit in one table page; an index row must fit in one

leaf page. If the average length of the rows in a table is more than one third of the page size, space utilization suffers. Only one row with 2100 bytes fits in a 4K page, for instance. The problem of unusable space is more pronounced with indexes. As new index rows *must* be placed in a leaf page according to the index key value, the leaf pages of many indexes should have free space for a few index rows, after load and reorganization. Therefore, index rows that are longer than 20% of the leaf page may result in poor space utilization and frequent leaf page splits. We have much more to say about this in Chapter 11.

With current disks, one rotation takes 4 ms (15,000 rpm) or 6 ms (10,000 rpm). As the capacity of a track is normally greater than 100 kilobytes (kb), the time for a random read is roughly the same for 2K, 4K, and 8K pages. It is essential, however, that the stripe size on RAID disks is large enough for one page; otherwise, more than one disk drive may have to be accessed to read a single page.

In most environments today, sequential processing brings several pages into the buffer pool with one I/O operation—several pages may be transferred with one rotation, for instance. The page size does not then make a big difference in the performance of sequential reads.

SQL Server 2000 uses a single page size for both tables and indexes: 8K. The maximum length of an index row is 900 bytes.

Oracle uses the term *block* instead of *page*. The allowed values for BLOCK_SIZE are 2K, 4K, 8K, 16K, 32K, and 64K, but some operating systems may limit this choice. The maximum length of an index row is 40% of BLOCK_SIZE. In the interests of simplicity, we trust Oracle readers will forgive us if we use the term *page* throughout this book.

DB2 for z/OS supports 4K, 8K, 16K, and 32K pages for tables but only 4K pages for indexes. The maximum length for index rows is 255 bytes in V7, but this becomes 2000 bytes in V8.

DB2 for LUW allows page sizes of 4K, 8K, 16K, and 32K for both tables and indexes. The upper limit for the index row length is 1024 bytes.

Table Clustering

Normally a table page contains rows from a single table only. Oracle provides an option to interleave rows from several related tables; this is similar to storing a hierarchical IMS database record with several segment types. An insurance policy, for instance, may have rows in five tables. The policy number would be the primary key in one table and a foreign key in the other four tables. When all the rows relating to a policy are interleaved in one table, they might all fit in one page; the number of table I/Os required to read all the data for one policy will then be only one, whereas it would otherwise have been five. On the other hand, as older readers may remember, interleaving rows from many tables may create problems in other areas.

Index Rows

The maximum number of columns in an index varies across the current DBMSs: SQL Server 16, Oracle 32, DB2 for z/OS 64, and DB2 for LUW 16 (refer to Chapter 12 for more detail).

Indexing variable-length columns have limitations in some products. If only fixed-length index rows are supported, the DBMS may pad an index column to the maximum length. As variable-length columns are becoming more common (because of JAVA, for instance)—even in environments in which they were rarely used in the past—support for variable-length index columns (and index rows) is now the norm in the latest releases. DB2 for z/OS, for instance, has full support for variable-length index columns in V8.

Normally, all columns copied to an index form the index key, which determines the order of the index entries. In unique indexes, an index entry is the same as an index row. With nonunique indexes, there is an entry for each distinct value of the index key together with a pointer for each of the duplicate table rows; this pointer chain is normally ordered by the address of the table row. DB2 for LUW, for instance, allows nonkey columns at the end of an index row. In addition to the above, each index entry requires a certain amount of control information, used, for example, to chain the entries in key sequence; throughout this book, this control information will be assumed, for the purpose of determining the number of index rows per page, to be about 10 bytes in length.

Table Rows

We have already seen that some DBMSs, for instance, DB2 for z/OS, DB2 for LUW, Informix, and Ingres, support a clustering index, which affects the placement of inserted table rows. The objective is to keep the order of the table rows as close as possible to the order of the rows in the clustering index. If there is no clustering index, the inserted table rows are placed in the last page of the table or to any table page that has enough free space.

Some DBMSs, for example, Oracle and SQL Server, do not support a clustering index that influences the choice of table page for an inserted table row. However, with any DBMS, the table rows can be maintained in the required order by reorganizing the table frequently; by reading the rows via a particular index (the index that determines the required order) before the reload or by sorting the unloaded rows before the reload.

Oracle and SQL Server provide an option for storing the table rows in the index as shown in the next section. More information is provided in Chapter 12.

Index-Only Tables

If the rows in a table are not too long, it may be desirable to copy *all* the columns into an index to make SELECTs faster. The table is then somewhat redundant.

Some DBMSs have the option of avoiding the need for the table. The leaf pages of one of the indexes then effectively contain the table rows.

In Oracle, this option is called an index-organized table, and the index containing the table rows is called the primary index. In SQL Server, the table rows are stored in an index created with the option CLUSTERED. In both cases, the other indexes (called secondary indexes in Oracle and unclustered indexes in SQL Server) point to the index that contains the table rows.

The obvious advantage of index-only tables is a saving in disk space. In addition, INSERTs, UPDATEs, and DELETEs are a little faster because there is one less page to modify.

There are, however, disadvantages relating to the other indexes. If these point to the table row using a direct pointer (containing the leaf page number), a leaf page split in the primary (clustered) index causes a large number of disk I/Os for the other indexes. Any update to the primary index key that moves the index row, forces the DBMS to update the index rows pointing to the displaced index row. This is why SQL Server, for instance, now uses the *key* of the primary index as the pointer to the clustered index. This eliminates the leaf page split overhead, but the unclustered indexes become larger if the clustered index has a long key itself. Furthermore, any access via a nonclustered index goes through two sets of nonleaf pages; first, those of the unclustered index and then those of the clustered index. This overhead is not a major issue as long as the nonleaf pages stay in the buffer pool.

The techniques presented in this book apply equally well to index-only tables, although the diagrams always show the presence of the table. *If index-only tables are being used, the primary (clustered) table should be considered as a clustering index that is fat for all SELECTs*. This last statement may not become clear until Chapter 4 has been considered. The order of the index rows is determined by the index key. The other columns are nonkey columns.

Note that in SQL Server the clustered index does not have to be the *primary key index*. However, to reduce pointer maintenance, it is a common practice to choose an index whose key is not updated, such as a primary or a candidate key index. In most indexes (the nonkey column option will be discussed later), all index columns make up the key, so it may be difficult to find other indexes in which *no* key column is updated.

Page Adjacency

Are the logically adjacent pages (such as leaf page 1 and leaf page 2) physically adjacent on disk? Sequential read would be very fast if they are (level 2 in Fig. 2.10).

In some older DBMSs, such as SQL/DS and the early versions of SQL Server, the pages of an index or table could be spread all over a large file. The only difference in the performance of random and sequential read was then due to the fact that a number of logically adjacent *rows* resided in the same page (level 1 in Fig. 2.10). Reading the next page required a random I/O. If there are

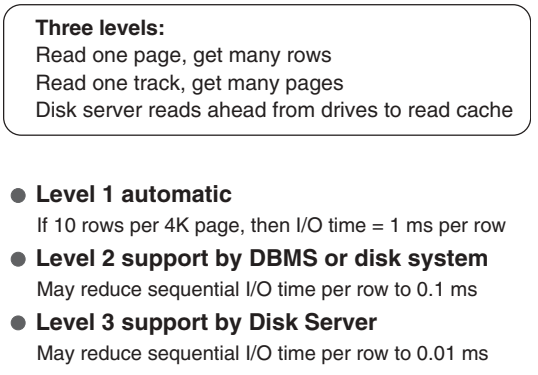


Figure 2.10 Page adjacency.

10 rows per page and a random I/O takes 10 ms, the I/O time for a sequential read is then 1 ms per row.

SQL Server allocates space for indexes and tables in chunks of eight 8K pages. DB2 for z/OS allocates space in *extents*; an extent may consist of many megabytes, all pages of a medium-size index or table often residing in one extent. The logically adjacent pages are then physically next to each other. In Oracle (and several other systems) the placement of pages depends on file options chosen.

Many databases are now stored on RAID 5 or RAID 10 disks. RAID5 provides *striping* with *redundancy*. RAID 10, actually RAID 1 + RAID 0, provides *striping* with *mirroring*.

The terms *redundancy* and *mirroring* are defined in the glossary. RAID *striping* means storing the first stripe of a table or index (e.g., 32K) on drive 1, the second stripe on drive 2, and so on. This obviously balances the load on a set of drives, but how does it affect sequential performance? Surprisingly, the effect may be positive.

Let us consider a full table scan where the table pages are striped over seven drives. The disk server may now *read ahead from seven drives in parallel*. When the DBMS asks for the next set of pages, they are likely to be already in the read cache of the disk server. This combination of prefetch activity may bring the I/O time down to 0.1 ms per 4K page (level 3 in Fig. 2.10). The 0.1-ms figure is achievable with fast channels and a disk server that is able to detect that a file is being processed sequentially.

Alternatives to B-tree Indexes

Bitmap Indexes

Bitmap indexes consist of a bitmap (bit vector) for each distinct column value. Each bitmap has one bit for every row in the table. The bit is on if the related row has the value represented by the bitmap.

Bitmap indexes make it feasible to perform queries with *complex and unpredictable* compound predicates against a large table. This is because ANDing and

ORing (covered in Chapters 6 and 10) bitmap indexes is very fast, even when there are hundreds of millions of table rows. The corresponding operation with B-tree indexes requires collecting a large number of pointers and sorting large pointer sets.

On the other hand a B-tree index, containing the appropriate columns, eliminates table access. This is important because random I/Os to a large table are very slow (about 10 ms). With a bitmap index, the table rows *must* be accessed unless the SELECT list contains only COUNTs. Therefore, the total execution time using a bitmap index may be *much longer* than with a tailored, (fat) B-tree index.

Bitmap indexes should be used when the following conditions are true:

1. The number of possible predicate combinations is so large that designing adequate B-tree indexes is not feasible.
2. The simple predicates have a high filter factor (considered in Chapter 3), but the compound predicate (WHERE clause) has a low filter factor—or the SELECT list contains COUNTs only.
3. The updates are batched (no lock contention).

Hashing

Hashing—or randomizing—is a fast way to retrieve a single table row whose primary key value is known. When the row is stored, the table page is chosen by using a randomizer, which converts the primary key value into a page number between 1 and N. If that page is already full, the row is placed in another page, chained to the home page. When a SELECT ... WHERE PK = :PK is issued, the randomizer is used again to determine the home page number. The row is either found in that page or by following the chain that starts on that page.

Randomizers were commonly used in nonrelational DBMSs such as IMS and IDMS. When the area size (N) was right—corresponding to about 70% space utilization, the number of I/Os to retrieve a record could have been as low as 1.1, which was very low compared to an index (a three-level index at that time could require two I/Os—plus a third to access the record itself). However, the space utilizations required constant monitoring and adjustments. When many records were added, the overflow chains grew and the number of I/Os increased dramatically. Furthermore, range predicates were not supported. Oracle provides an option for the conversion of a primary key value to a database page number by hashing.

Many Meanings of Cluster

Cluster is a term that is widely used throughout relational literature. It is also a source of much confusion because its meaning varies from product to product.

In DB2 (z/OS, LUW, VM, and VSE), a clustering index refers to an index that defines the home page for a table row being inserted. An index is clustered if there is a high correlation between the order of the index rows and the table rows.

A table can have only one clustering index but, at a given point in time, *several* indexes may be clustered. The CLUSTERRATIO of an index is a measure of the correlation between the order of the index rows and the table rows. It is used by the optimizer to estimate I/O times.

DB2 tables normally have a clustering index.

In SQL Server, the index that stores the table rows is called *clustered*; a clustered index is only defined if an index-only table is required. The other indexes (SQL Server term: *nonclustered indexes*) point to the *clustered index*.

In Oracle, the word *cluster* is used for the option to interleave table rows (clustered tables). *It has nothing to do with the clustering index* that we have taken to determine the sequence of the table rows.

DB2 for LUW V8 has an option called *multidimensional clustering*; this enables *related rows* to be grouped together. Refer to Chapter 13 for more details.

Important

In the diagrams throughout this book, C is used to *mark the index that defines the home page for a table row that is being inserted*. In our calculations, *the table rows are assumed to be in that same order*. For a product that does not support a clustering index in this sense, the order of the table rows is determined when reorganizing and reloading the table.

Chapter 3

SQL Processing

- How indexes and tables are used to process SQL statements
- SQL processing concepts
- Predicates
- Optimizers and access paths
- Index slices, matching index scans, and matching columns, index screening, and screening columns
- When the access path is chosen
- Monitoring the optimizer
- Helping the optimizer with statistics and the number of FETCH calls required
- Concepts of filter factor, selectivity, and cardinality, together with their impact on index design
- The materialization of the result table and its implications

INTRODUCTION

We now have some understanding of the structure of tables and indexes; we also understand how these objects relate to buffer pools, disks, and disk servers and how the latter are used to make the data available to the SQL process. We are now in a position to consider the processing of the SQL calls.

As before, much will depend on the individual relational DBMS being used; here again, there are many similarities, but there are also a number of differences. We will describe the underlying processes first in general terms in order to avoid confusing the basic issues with DBMS specifics; the latter will be addressed as appropriate. Considerably more detail of these processes will be provided throughout the book, at a time when it is more convenient to do so. Remember that the glossary provided at the end of the book summarizes all the terms used throughout this chapter.

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

PREDICATES

A WHERE clause consists of one or more *predicates* (search arguments). Three *simple predicates* are shown in SQL 3.1. These are:

SQL 3.1

```
WHERE   SEX = 'M'
        AND
        (WEIGHT > 90
        OR
        HEIGHT > 190)
```

- `SEX = 'M'`
- `WEIGHT > 90`
- `HEIGHT > 190`

They can also be considered as two *compound predicates*:

- `WEIGHT > 90 OR HEIGHT > 190`
- `SEX = 'M' AND (WEIGHT > 90 OR HEIGHT > 190)`

Predicates are the primary starting points for index design. When an index supports *all* the predicates of a SELECT statement, the optimizer is likely to build an efficient access path.

Comment

In his numerous books, Chris Date uses the term *predicate* reluctantly (4):

> *We follow conventional database usage here in referring to conditions as predicates. Strictly speaking, however, this usage is incorrect. A more accurate term would be conditional expression or truth-valued expression. (page 79)*

OPTIMIZERS AND ACCESS PATHS

One of the long-standing advantages of relational databases has been that data is requested with little or no thought for the way in which the data is to be accessed. This decision is made by a component of the DBMS called the *optimizer*. These have varied widely across different relational systems and probably always will, but they all try to access the data in the most effective way possible, using statistics stored by the system collected on the data. The optimizer is, of course, at the heart of SQL processing and is also central to this book as well.

Before an SQL statement can be executed, the optimizer must first decide how to access the data; which index, if any, should be used; how the index should be used; should assisted random read be used; and so forth. All of this information

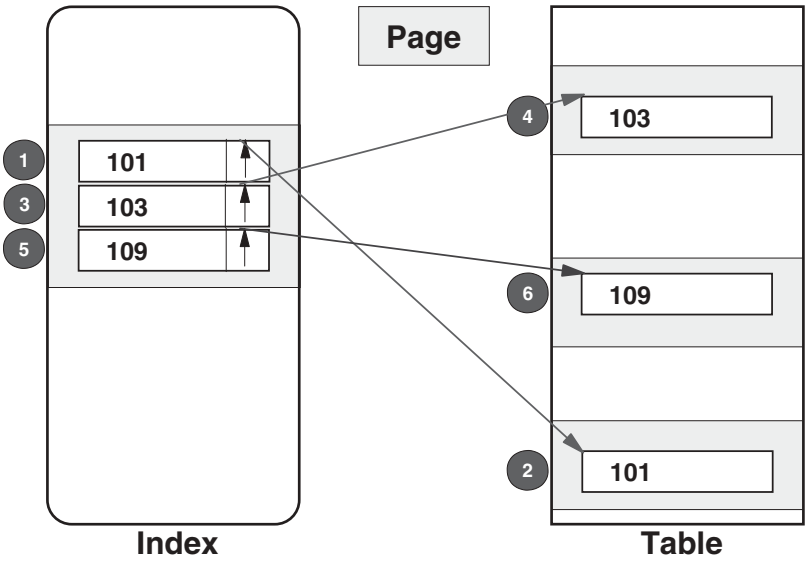


Figure 3.1 Traditional index scan.

is embodied in the *access path*. For the example we used in Chapter 2 and shown again in Figure 3.1, the access path would define a sequential scan of a portion of the index together with a synchronous read for each table page required.

Index Slices and Matching Columns

Thus a *thin slice* of the index, depicted in Figure 3.1, will be sequentially scanned, and for each index row having a value between 100 and 110, the corresponding row will be accessed from the table using a synchronous read unless the page is already in the buffer pool. The cost of the access path is clearly going to depend on the thickness of the slice, which in turn will depend on the range predicate; a thicker slice will, of course, require more index pages to be read sequentially and more index rows will have to be processed. The real increase in cost though is going to come from an increase in the number of *synchronous reads* to the table at a cost of perhaps 10 ms per page. Similarly, a thinner slice will certainly reduce the index costs, but again the major saving will be due to fewer synchronous reads to the table. The size of the index slice may be very important for this reason.

The term *index slice* is *not* one that is used by any of the relational database systems; these all use their own terminology, but we feel that an *index slice* is a much more descriptive term and one that we can use regardless of DBMS. Another way that is often used to describe an index slice is to define the number of *matching columns* used in the processing of the index. In our example (*having a value between 100 and 110*), we have only a single matching column. This

single matching column in fact defines the thickness of our slice. If there had been a second column, both in the WHERE clause and in the index, such that the two columns together were able to define an even thinner slice of the index, we would have two matching columns. Less index processing would be required, but of even greater importance, fewer synchronous reads to the table would be required.

Index Screening and Screening Columns

Sometimes a column may indeed be in both the WHERE clause and in the index, but for one reason or another it is not able to *further* define the index slice—this is a very complex area and one that will have to be addressed continually throughout this book; suffice it to say at this time that not all columns in the index are able to define the index slice. Such columns, however, may still be able to reduce the number of synchronous reads to the table and so play the more important part. We will call these columns *screening columns*, as indeed do some relational database systems, because this is exactly what they do. They avoid the necessity of accessing the table rows because they are able to determine that it is not necessary to do so by their very presence in the index.

Without going into too much detail at this stage, we can at least provide an initial simple understanding of how one can determine whether predicates may participate in this matching and screening process.

Figure 3.2 shows a WHERE clause consisting of three predicates, the columns of each being part of the index shown; column D, the last column in the index, is not present in a predicate.

We must take the first index column A. This column is present in an equal predicate, the simplest predicate of all. This is obviously a matching column and will be used to define the index slice.

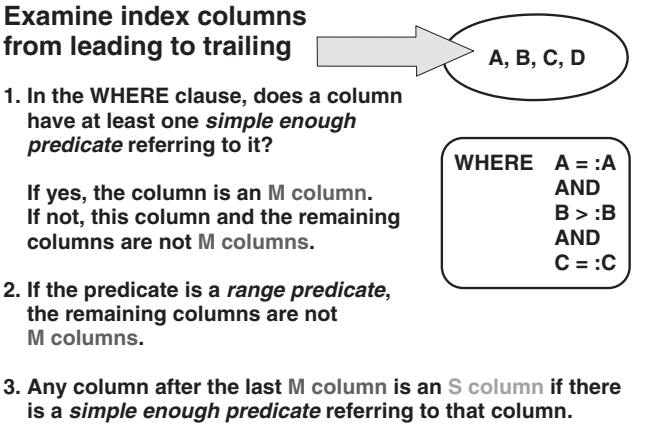


Figure 3.2 Predicting matching and screening columns.

Next we take the second index column, B. This too, like the BETWEEN predicate we saw in Figure 3.1, is simple enough to be a matching column and again be used to define the index slice.

Item 2 in Figure 3.2 now indicates that because B is a *range* predicate, the remaining column C cannot participate in the matching process—it *cannot* further define the index slice. Item 3 goes on to say, however, that column C *can* avoid unnecessary table accesses because it can participate in the screening process. In effect column C is *almost* as important as columns A and B; the index slice scanned will just be a little thicker.

The most difficult issue is what constitutes a simple or a difficult predicate; this depends very much on the DBMS, and we can safely leave this until Chapter 6.

To summarize, the WHERE clause shown in Figure 3.2 has two matching columns, A and B, which define the index slice used. In addition it has one other column, C, which will be used for screening; thus the table will only be accessed when it is *known* that a row satisfies *all three* predicates. If the predicate on column B had been omitted, we would have a thicker index slice with only one matching column, A; but column C would still be used for screening. If the predicate on column B had been an equal predicate, *all three columns* would be used for matching, resulting in a very thin index slice. Finally, if the predicate on column A had been omitted, the index slice used would be the *whole index*, with columns B and C both used for screening.

Access Path Terminology

Unfortunately, the terminology used to describe access paths is far from standardized—even the term *access path* itself, another term that is often used being the *execution plan*. In this book we will use *access path* when referring to the way the data is actually accessed; we will use *execution plan* when referring to the output provided by the DBMS by means of an EXPLAIN facility (described below). This is a subtle distinction, and it is really of no consequence if the terms are used interchangeably. Other, much more important issues are now described.

Matching predicates are sometimes called *range delimiting* predicates. If a predicate is simple enough for an optimizer in the sense that it is a *matching* predicate when a suitable index is available, it is called *indexable* (DB2 for z/OS) or *sargable* (SQL Server, DB2 for VM and VSE). The opposite term is *nonindexable* or *nonsargable*. Some Oracle books use the term *index suppression* when they discuss predicates that are too difficult for matching.

SQL Server uses the term *table lookup* for an access path that uses an index but also reads table rows. This is the opposite of index only. The obvious way to eliminate the table accesses is to add the missing columns to the index. Many SQL Server books call an index a *covering index* when it makes an index-only access path possible for a SELECT call. SELECT statements that use a covering index are sometimes called *covering SELECTs*.

In DB2 for z/OS the predicates that are too complex for the optimizer in the sense that index *screening* is not enabled are called stage 2 predicates. The opposite term is stage 1. Many product guides do not discuss index screening at all. Hopefully, this means that the products do index screening whenever it is logically possible; before making this assumption, however, it could well be worthwhile performing a simple experiment such as the following:

A table having four columns is created with 100,000 rows; the first three columns, A, B, and C, each have 100 different values. Column D is added to ensure that the table is accessed; it should be long enough to allow only one table row per page. An index (A, B, C) is created and SELECT D FROM TABLE WHERE A = :A AND C = :C is run to determine the number of *table rows* accessed. We are assuming that *all* the relevant rows are FETCHed and that the index is in fact used; with only one row per page and consequently having 100,000 pages in the table, it is unlikely that it wouldn't be. If the observed number of table pages accessed is close to 10 ($0.01 \times 0.01 \times 100{,}000$; the first 0.01 for matching, the second for screening), index screening must have been used for predicate C = :C. If it is closer to 1000 ($0.01 \times 100{,}000$; only the first 0.01, used for matching), then it hasn't. This experiment can then be repeated at any time with complex predicates to determine whether they are too difficult for the optimizer.

Chapter 6 discusses in some detail difficult and very difficult predicates, together with the implications regarding matching and screening.

Monitoring the Optimizer

When a slow SQL call is identified, the first suspect is often the optimizer; perhaps it chose the wrong access path. A facility is available in relational DBMSs to explain the decision made by the optimizer; this is called EXPLAIN, SHOW PLAN, or EXPLAIN PLAN and is discussed in Chapter 7. All references to EXPLAIN from now on should be taken to apply to all three terms.

Helping the Optimizer (Statistics)

The optimizer may have made a wrong choice because the statistics it uses as a basis for making cost estimates were inadequate; perhaps the options chosen when the statistics were gathered were too limited or the statistics are out-of-date. These statistics are usually gathered on request, for example, by running a special program called RUNSTATS in DB2 for z/OS.

The default statistics gathered normally include basic information, such as the number of table rows and pages per table, the number of leaf pages, the cluster ratio per index, and the number of distinct values for some columns or column combinations (termed cardinality) as well as the highest and lowest values (or the second highest and the second lowest) for some columns. Other optional statistics provide more information about the value distribution of columns and column

combinations, such as the N most common values together with the number of rows for each value; many products (e.g., Oracle, SQL Server, and DB2 for LUW) may also allow value distributions to be collected as histograms (N% of rows occur in a user-defined number of key ranges).

Helping the Optimizer (Number of FETCH Calls)

When cost-based optimizers estimate the cost of alternative access paths, they assume that *all* the result rows are required (FETCHed), unless they are informed otherwise. If the whole result set is *not* going to be required, we can indicate that we are only interested in the first N result rows. This is done in SQL Server by adding:

```
OPTIONS (FAST n)
```

at the end of the SELECT statement.

With Oracle, an access path hint is used:

```
SELECT/*+ FIRST_ROWS(n)*/
```

[the (n) option is only available with Oracle 9i and above].

The syntax for DB2 for z/OS is

```
OPTIMIZE FOR n ROWS
```

Examples are shown in SQL 3.2 S, O and D below.

SQL 3.2S

```
DECLARE LASTINV CURSOR FOR
SELECT     INO, IDATE, IEUR
FROM       INVOICE
WHERE      CNO = :CNO
ORDER BY   INO DESC
OPTIONS (FAST 1)
```

SQL 3.2O

```
DECLARE LASTINV CURSOR FOR
SELECT /*+ FIRST_ROWS(1)*/
           INO, IDATE, IEUR
FROM       INVOICE
WHERE      CNO = :CNO
ORDER BY   INO DESC
```

SQL 3.2D

```
DECLARE LASTINV CURSOR FOR
SELECT     INO, IDATE, IEUR
FROM       INVOICE
WHERE      CNO = :CNO
ORDER BY   INO DESC
OPTIMIZE FOR 1 ROW
```

Please note that these options do *not* restrict the number of FETCH calls that may be issued. SELECT TOP (n) with SQL Server and FETCH FIRST n ROW(S) ONLY with DB2 for z/OS are provided for that purpose.

When the Access Path Is Chosen

The only additional optimizer-based question we need to understand at this stage is the extremely important one posed in Figure 3.3. It should be clear that choosing the access path *every time* an SQL statement is executed will consume much more processing power than doing it perhaps only once, when *the application is developed*; the cost of the access path selection process is not insignificant for cost-based optimizers.

Far less obvious, although sometimes far more important, is that choosing the access path *every time* an SQL statement is executed, may give the optimizer a much better chance of choosing the *best* access path. This is because actual values, rather than program or host variables, are then used. WHERE SALARY >:SALARY is not nearly as transparent as WHERE SALARY >1,000,000.

The items shown in Figure 3.3 indicate how optimizers allow this choice to be made; these again are discussed later.

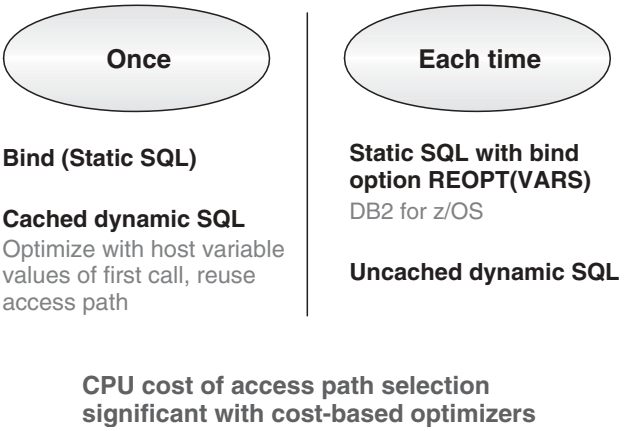


Figure 3.3 When does the optimizer select the access path.

FILTER FACTORS

The *filter factor* specifies the selectivity of a predicate—what proportion of the source table rows satisfy the condition expressed by the predicate. It is dependent on the distribution of the column values. Thus, the filter factor of predicate SEX = 'F' increases whenever a female customer is added to the customer table.

The predicate CITY = :CITY has an *average filter factor* (1/the number of distinct values of CITY) as well as a *value specific filter factor* (CITY = 'HELSINKI'). The difference between these two is critical, both for index design and for access path selection. Figure 3.4 shows an example where a filter factor of 0.2% for the CITY column in a 1-million-row customer table predicts the size of the result set will be 2000 rows.

When evaluating the adequacy of an index, *worst-case filter factors* are more important than average filter factors; worst case relates to the *worst input*, that is, the input that results in the longest elapsed time with a given index.

Filter Factors for Compound Predicates

The filter factor for a *compound predicate* can be derived from the filter factors of the simple predicates, only if the values of the predicate columns are not statistically correlated. Consider, for example, the filter factor for CITY = :CITY AND LNAME = :LNAME. If there is no statistical correlation between the columns CITY and LNAME, the filter factor of the compound predicate will be equal to the *product* of the filter factors for CITY = :CITY and LNAME = :LNAME. If column CITY has 500 distinct values and column LNAME 10,000 distinct values, the filter factor for the compound predicate will be $1/500 \times 1/10{,}000$ or 1/5,000,000. This implies that the column combination CITY, LNAME has 5,000,000 distinct values. In most CUSTOMER tables, however, the two columns *are* correlated. The proportion of Andersens is much higher in Copenhagen than in London, and there could even be cities in England that don't have a single customer with a common Danish last name. Therefore, the filter factor of this

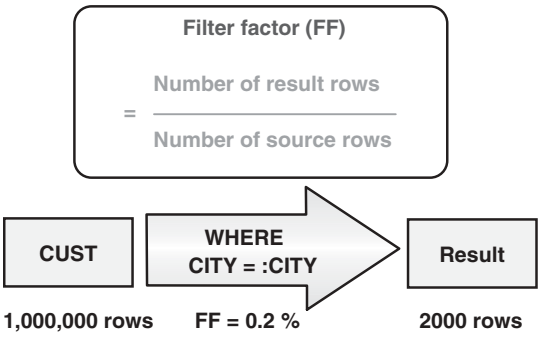


Figure 3.4 Filter factor is a property of a predicate.

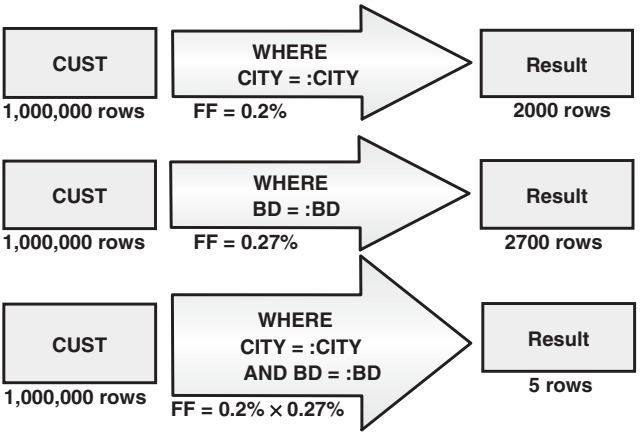


Figure 3.5 No correlation.

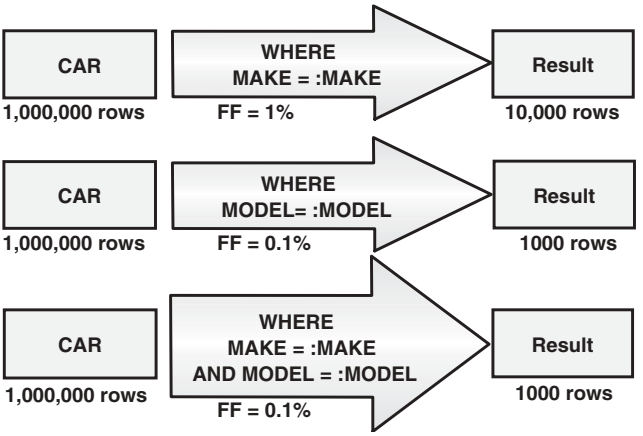


Figure 3.6 Strong correlation.

compound predicate is probably much lower than the product of the two simple predicates.

Columns CITY and BD (birthday, mmdd) are probably not related. Therefore, the filter factor of the compound predicate can be estimated by a process of multiplication as shown in Figure 3.5.

An extreme example of column correlation is the table CAR, shown in Figure 3.6, with columns MAKE and MODEL. The filter factors for predicates MAKE = :MAKE and MODEL = :MODEL may be 1/100 and 1/1000, respectively. The filter factor for MAKE = :MAKE AND MODEL = :MODEL is much less than 1/100,000, perhaps as low as 1/1000 because, for example, to the best of our knowledge, only Ford made Model Ts.

When designing indexes, one should *estimate* the filter factor for a compound predicate *as a whole* rather than basing it on zero correlation. Fortunately, the estimation of the worst-case filter factor is normally straightforward. If the *largest* filter factors for CITY = :CITY and LNAME = :LNAME are 10 and 1%, respectively, the *largest* filter factor for CITY = :CITY AND LNAME = :LNAME is indeed $0.1 \times 0.01 = 0.001 = 0.1\%$ because it relates to the LNAME distribution of a *specific* CITY.

Optimizers also have to estimate the filter factors before they can estimate the costs for alternative access paths. When they were less sophisticated than they are today, an incorrect filter factor estimate for compound predicates was one of the most common reasons for inappropriate access path choices. Many current optimizers have options for counting or sampling the *cardinality* (the number of distinct values) of *index column combinations.*

Comment

A similar idea, sometimes discussed in relational literature, is the *selectivity ratio*. Vipul Minocha (3) defines this in the following way:

> Selectivity ratio =100 × (Total number of rows uniquely identified by the key)/Total number of rows in the table.

If table CUST has 1 million rows and 200,000 of them have the value HELSINKI in column CITY, the selectivity ratio of index CITY is $100 \times 200{,}000/1{,}000{,}000 = 20\%$ for HELSINKI, equal to the filter factor of the predicate CITY = 'HELSINKI'. The index would therefore have poor selectivity, especially for the value HELSINKI, so CITY would not be a good index.

Selectivity is a somewhat confusing concept because it is a property of an index (and the whole index key); but the picture becomes even more confusing as a quotation from Ref. 1 nicely warns:

> *Various definitions of selectivity from vendor manuals or other texts are imprecise or confusing: "the number of rows in the table divided by the number of distinct values" (Oracle); "the ratio of duplicate key values in an index" (Sybase). The phrase "high selectivity" means either "a large selectivity number" or "a low selectivity number" depending on who is saying it. (page 552)*

We do not wish to add to this confusion; the *filter factor* is a more useful concept for index design, and an essential one for performance prediction. It will be widely used throughout this book.

Impact of Filter Factors on Index Design

The thickness of the *index slice* that must be scanned is of importance to the performance of an access path. With current hardware, the most important measure of thickness is the number of index rows that must be scanned; the *total* number of index rows multiplied by the filter factor of the *matching* compound predicate.

A *matching predicate*, by definition, participates in defining the slice; where to enter, where to exit. The number of index rows is the same as the number of table rows, with the exception that Oracle does not create index rows for NULL values. Consider the following query (SQL 3.3):

SQL 3.3

```
SELECT      PRICE, COLOR, DEALERNO
FROM        CAR
WHERE       MAKE = :MAKE
            AND
            MODEL = :MODEL
ORDER BY    PRICE
```

Both simple predicates will be matching predicates if the index in Figure 3.7 is used. If the filter factor of the compound predicate is 0.1%, the index slice accessed will be 0.1% of the whole index. Columns MAKE and MODEL are *matching columns*. The index in Figure 3.7 appears to be fairly appropriate for the SELECT shown in SQL 3.3, although it is far from being the *best possible* index; at least the index slice to be scanned is rather thin.

The index is not as good, however, for the SELECT shown in SQL 3.4 because we have only *one* matching column. This is shown in Figure 3.8.

If index SEX, HEIGHT, WEIGHT, CNO is chosen to search for large men using the SELECT shown in SQL 3.5, there will be *only one* matching predicate, SEX, the filter factor of which is normally 0.5. The reason why only one predicate participates in the matching process will be discussed later.

Consequently, if there are 1,000,000 rows in table CUST, the index slice to be scanned will have $0.5 \times 1{,}000{,}000$ rows $= 500{,}000$ rows, a very thick slice!

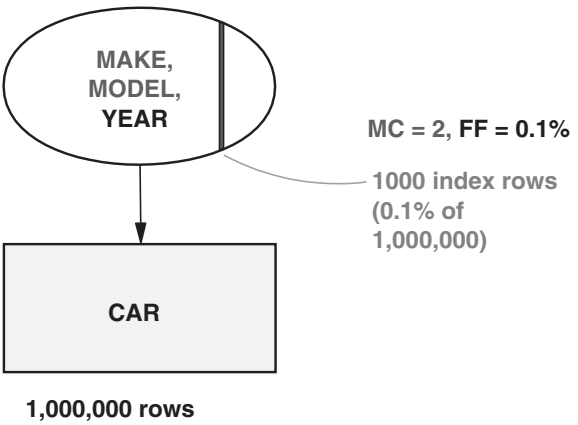


Figure 3.7 Two matching columns—a thin slice.

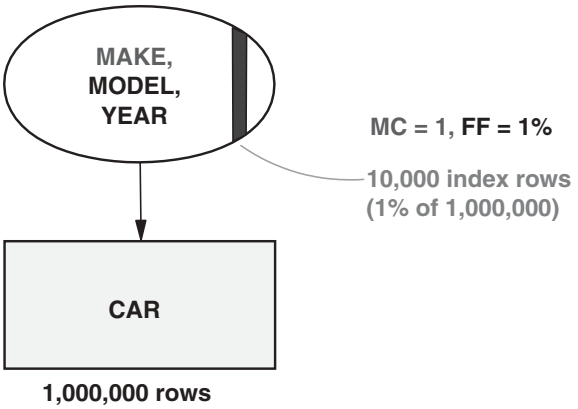


Figure 3.8 One matching column—a thicker slice.

SQL 3.4

```
SELECT   PRICE, COLOR, DEALERNO
FROM     AUTO
WHERE    MAKE = :MAKE
         AND
         YEAR = :YEAR
```

Some textbooks recommend that index columns should be ordered according to *descending cardinality* (the number of distinct values). Taken literally, this advice would lead to absurd indexes, such as CNO, HEIGHT, WEIGHT, SEX, but in certain situations—assuming the order of the columns doesn't adversely affect the performance of SELECT statements [e.g., with WHERE A = :A AND B = :B, indexes (A, B) and (B, A) would be equivalent] nor that of updates—this is a reasonable recommendation. It increases the probability that the index will be useful for *other* SELECTs.

SQL 3.5

```
SELECT     LNAME, FNAME, CNO
FROM       CUST
WHERE      SEX = 'M'
           AND
           (WEIGHT > 90
           OR
           HEIGHT > 190)
ORDER BY   LNAME, FNAME
```

MATERIALIZING THE RESULT ROWS

Materializing the result rows means performing the *database accesses* required to *build the result set*. In the best case, this simply requires a row to be moved from the database buffer pool to the program. In the worst case, the DBMS will request a large number of disk reads.

When a single row is retrieved with a *singleton* SELECT, there is no choice but to materialize the result row when the SELECT call is executed. On the other hand, when a *cursor* is used, which is necessary if there may be several result rows, there are *two* alternatives as shown in Figure 3.9:

1. The DBMS materializes the *whole* result table at OPEN CURSOR (or at least at the first FETCH).
2. Each FETCH materializes one result row.

We will discuss each of these possibilities in turn.

Cursor Review

A FETCH call moves one row of the result table defined by the DECLARE CURSOR statement to the application program. When DECLARE CURSOR contains *host variables*, at execution time the application program moves the actual values into the host variables before the OPEN CURSOR call. If the application program creates several result tables using the same cursor, a CLOSE CURSOR call must first be issued. New values may be moved into the host variables and the cursor is reopened with an OPEN CURSOR call.

The CLOSE CURSOR call releases any locks that may have been taken by the last FETCH call, assuming the customary isolation level (ANSI SQL-92: READ COMMITTED, Level 1, e.g., CURSOR STABILITY with DB2). If

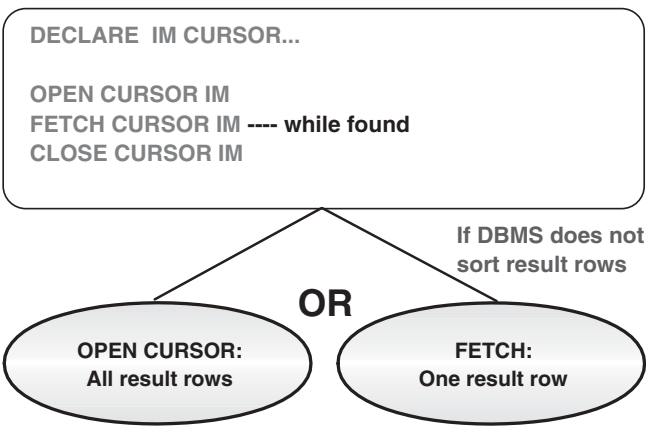


Figure 3.9 Result row materialization.

the program does not issue CLOSE CURSOR or if a stronger isolation level is specified, the DBMS releases the locks at commit point.

Query tools generate DECLARE CURSOR, OPEN CURSOR, a FETCH loop, and CLOSE CURSOR from the SELECT statement.

SQL application programming courses have been known to claim that OPEN CURSOR *always* creates the result table, defined by DECLARE CURSOR with the values moved into the host variables, at the time of the OPEN CURSOR call. Fortunately, this statement is *not true*. To avoid unnecessary work, the DBMS materializes result rows *as late as possible*. The chosen materialization time may also affect the *contents* of the result: if the DBMS chooses *early materialization*, the FETCH calls retrieve result rows from a temporary table; the DBMS does not update this temporary table when the database is updated.

To illustrate the significance of the time of materialization, we will use the program shown in SQL 3.6, which reads a customer's *last* invoice.

The response to the query *is always* a *single* row. The program will issue only one FETCH call, hence the request to use the most efficient access path for 1 row. The actual syntax is DBMS specific as we saw earlier; whenever we want to use this function throughout this book, we will use this "generic form" to avoid the necessity of providing multiple variations of the code. Despite making the one row request, what does the DBMS actually read?

1. The *whole* result table as defined by the cursor (*all* the customer's invoices)?
2. A *single* row relating to the required invoice?

SQL 3.6

```
DECLARE LASTINV CURSOR FOR
SELECT     INO, IDATE, IEUR
FROM       INVOICE
WHERE      CNO = :CNO
ORDER BY   INO DESC
WE WANT 1 ROW PLEASE

OPEN CURSOR LASTINV
   FETCH CURSOR LASTINV
CLOSE CURSOR LASTINV
```

From the performance point of view, this may be a *critical question*. We will consider the two possible alternatives in turn.

Alternative 1: FETCH Call Materializes One Result Row

The DBMS chooses this desirable alternative if

- There *is no* sequence requirement (ORDER BY, GROUP BY, etc.) *or*

- There *is* a sequence requirement but the following conditions are *both* true:
  - There is an index that is able to provide the *result rows* in the ORDER BY sequence, for example (CNO, IDATE DESC).
  - The optimizer decides to use this index in the traditional way, that is, the first qualifying index row and its table row are accessed, followed by the second qualifying index row and its table row, and so on.

If the optimizer assumes that the program will fetch *the whole result set*, it may select the wrong access path—perhaps a full table scan.

Alternative 2: Early Materialization

By far the most common reason for this choice is that a sort of the result rows is necessary; this will be reported by EXPLAIN or its equivalent. There are a few special cases, partly DBMS specific, in which the whole result is materialized even though the result rows are *not* sorted. These cases may also be detected by the EXPLAIN output.

In our example, the DBMS would have to sort the result rows if there isn't an index starting with either the columns CNO and IDATE or with just the column IDATE. If the DBMS cannot read an index backwards, the index column IDATE would have to be specified as descending: IDATE DESC.

If early materialization is chosen, some DBMSs materialize the result table at OPEN CURSOR, others at the first FETCH. It is only necessary to know this if reading an SQL trace. Otherwise it does not make a significant difference; it seems unlikely that a program would issue an OPEN CURSOR without issuing any FETCH calls to it.

What Every Database Designer Should Remember

Sorting results rows implies that the whole result table is materialized even if only the *first* result row is fetched.

EXERCISES

3.1. Design the best possible indexing for the following query:

SQL 3.7

```
SELECT     LNAME, FNAME, CNO
FROM       CUST
WHERE      SEX = 'M'
           AND
           HEIGHT > 190
ORDER BY   LNAME, FNAME
```

3.2. Design the best possible indexing for the following query:

SQL 3.8

```
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
            AND
            (WEIGHT > 90
            OR
            HEIGHT > 190)
ORDER BY    LNAME, FNAME
```

Chapter 4

Deriving the Ideal Index for a SELECT

- Guidelines for the major performance factors concerning table and index scans
- Random and sequential read times and CPU costs
- Assigning stars to an index for a SELECT statement, according to the three most important requirements
- The design of a three-star index—the ideal index for the statement
- Fat indexes
- Algorithm to design the best index for a SELECT statement
- Consideration of the existing indexes to determine the most practical index, taking into account CPU time, disk read time, and elapsed time
- Implications of any suggested changes to the current indexes with regard to the maintenance overheads
- Response time, drive load, and disk costs
- Recommendations

INTRODUCTION

Many DBAs appear to be satisfied if all the SQL calls in a program use one or more index. Everything looks normal; "the EXPLAIN is clean." An inappropriate index, however, may lead to worse performance than a scan of the whole table.

Mark Gurry (2) agrees:

> *I am astounded at how many tuners, albeit inexperienced, believe that if an SQL statement uses an index, it must be well tuned. You should always ask, "Is it the best available index?" or "Could an additional index be added to improve the responsiveness?" or "Would a full table scan produce the result faster?" (page 47)*

In this chapter we are going to consider these extremely important questions in considerable detail, but first we will need to define the assumptions on which our analysis will be based.

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

BASIC ASSUMPTIONS FOR DISK AND CPU TIMES

We have already discussed disk I/Os in considerable detail in Chapter 2; from that information, we will now state some basic assumptions that we will use throughout this book. More information about CPU times will be found in Chapter 15.

```
Basic Assumptions for Disk and CPU Times
                  I/O Time
Random Read         10 ms (4K or 8K page)
Sequential Read     40 MB/s
       CPU Time for Sequential Scan
Examine a Row       5 µs
FETCH               100 µs
```

INADEQUATE INDEX

The only reasonable access path alternatives for the following simple query (SQL 4.1 and Figure 4.1) are:

1. Index scan with (LNAME, FNAME)
2. Full table scan

SQL 4.1

```
DECLARE CURSOR41 CURSOR FOR
SELECT     CNO, FNAME
FROM       CUST
WHERE      LNAME = :LNAME
           AND
           CITY = :CITY
ORDER BY   FNAME
```

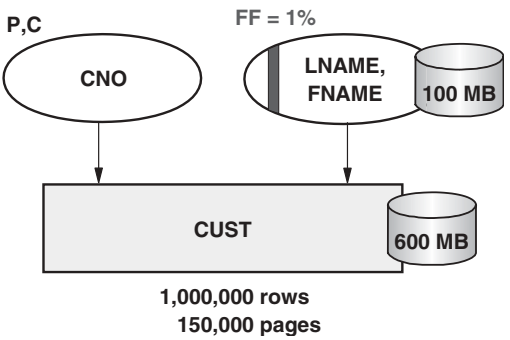


Figure 4.1 Full table scan or an inappropriate index?

Would one of these provide acceptable response times, even with the most common last name (filter factor 1%)?

With the first alternative, the DBMS would scan the index slice defined by predicate LNAME = :LNAME. For each index row in the slice, the DBMS must check the CITY value in the table. As the table rows are clustered by CNO and not by LNAME, this would probably require one random read from the disk. With the most common LNAME, obtaining the whole result would mean examining 10,000 index rows and 10,000 table rows, regardless of the filter factor for CITY. How long would this take?

Let us assume that the size of index (LNAME, FNAME)—including system data and distributed free space—is 1,000,000 × 100 bytes = 100 MB. Let us also assume that sequential read speed is 40 MB/s. Reading a 1% slice, 1 MB, then takes

$$10 \text{ ms} + 1 \text{ MB}/40 \text{ MB/s} = 35 \text{ ms}$$

This is obviously not a problem, but the 10,000 random reads would take

$$10{,}000 \times 10 \text{ ms} = 100 \text{ s}$$

making this alternative far too slow.

With the second alternative, only the first page requires a random read. If the size of the table—including distributed free space—is 1,000,000 × 600 bytes = 600 MB, the I/O time would be

$$10 \text{ ms} + 600 \text{ MB}/40 \text{ MB/s} = 15 \text{ s, still far too long}$$

The CPU time would be considerably higher with the second alternative because the DBMS must examine 1,000,000 rows instead of 20,000, and the result rows must be sorted. On the other hand, the CPU time and the I/O time overlap, thanks to sequential reads. A full table scan would be faster than the inadequate index in this case, but it is not fast enough; a better index would be required.

THREE-STAR INDEX—THE IDEAL INDEX FOR A SELECT

After discussing a very inadequate index for CURSOR41 in the previous section, let's go to the other extreme, a *three-star index*, which is by definition the *best possible index* for a given SELECT. A single table SELECT using a three-star index, such as the one shown in Figure 4.2, normally needs only one random disk drive read and a scan of a thin slice of an index. Therefore the response times are often a *couple of orders of magnitude shorter* than those with a mediocre index.

The elapsed time for CURSOR41, shown again in SQL 4.2, is a fraction of a second, even if the result table has 1000 rows. How is that possible? Figure 4.2 depicts the lowest level of the index, the leaf pages. The upper levels (nonleaf pages) will be discussed later.

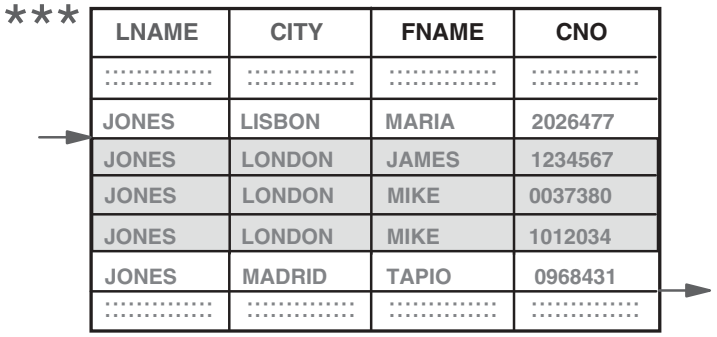


Figure 4.2 Three-star index for CURSOR41.

SQL 4.2

```
DECLARE CURSOR41 CURSOR FOR
SELECT     CNO, FNAME
FROM       CUST
WHERE      LNAME = :LNAME
           AND
           CITY = :CITY
ORDER BY   FNAME
```

If there are 1000 result rows, the filter factor for the compound predicate LNAME = :LNAME AND CITY = :CITY is 0.1%. The scanned index slice now consists of only 1000 rows because there are two matching predicates: columns LNAME and CITY define the index slice (this is why the slice is shaded in Fig. 4.2). Confirmation of the matching process will be shown by EXPLAIN: two matching columns (MC = 2). This will take

$$1 \times 10 \text{ ms} + 1000 \times 0.1 \text{ ms} = 0.1 \text{ s}$$

to read this index slice. The table is not accessed at all because all the required columns have been copied into the index.

How the Stars Are Assigned

An index deserves the first star if the index rows relevant to the SELECT are next to each other—as in Figure 4.2—or at least as close to each other as possible. This *minimizes the thickness of the index slice* that must be scanned.

The second star is given if the index rows are in the right order for the SELECT—as in Figure 4.2. This *eliminates sorting*.

If the index rows contain all the columns referred to by the SELECT—as in Figure 4.2—the index is given the third star. This eliminates table access: The access path is *index only*.

Of these three stars, the third is often the most important one. Leaving a column out of an index may lead to many slow random reads from the disk drive. An index that has *at least* the third star is a *fat index* for the given SELECT.

A Fat Index

A fat index is an index that has at least the third star. It contains all the table columns referred to by the SELECT and so enables index only access.

In simple cases, like CURSOR41, the structure of a three-star index is quite straightforward.

To Qualify for the First Star

Pick the columns from all equal predicates (WHERE COL = ...). Make these the first columns of the index—in any order. For CURSOR41, the three-star index will begin with columns LNAME, CITY or CITY, LNAME. In both cases the index slice that must be scanned will be as thin as possible.

To Qualify for the Second Star

Add the ORDER BY columns. Do not change the order of these columns, but ignore columns that were already picked in step 1. For example, if CURSOR41 had redundant columns in the ORDER BY, say ORDER BY LNAME, FNAME or ORDER BY FNAME, CITY, only FNAME would be added in this step. When FNAME is the third index column, the result table will be in the right order without sorting. The first FETCH call will return the row with the smallest FNAME value.

To Qualify for the Third Star

Add all the remaining columns from the SELECT statement. The order of the columns added in this step has no impact on the performance of the SELECT, but the cost of updates should be reduced by placing volatile columns at the end. Now the index contains all the columns required for an index-only access path.

The resulting three-star index is:

```
(LNAME, CITY, FNAME, CNO) or (CITY, LNAME, FNAME, CNO)
```

CURSOR41 is trivial in three respects:

- The WHERE clause does not contain range predicates (BETWEEN, >, >=, etc.).
- The FROM clause refers to a single table only.
- None of the predicates seems too difficult for the optimizer.

The third of these items will be addressed in Chapter 6; the second in Chapter 8; the first will be considered at a simple level now and in more depth in Chapter 6.

Range Predicates and a Three-Star Index

This example, SQL 4.3, requires the same information as before, but the required customers are now within a *range* of values.

SQL 4.3

```
DECLARE CURSOR43 CURSOR FOR
SELECT   CNO, FNAME
FROM     CUST
WHERE    LNAME BETWEEN :LNAME1 AND :LNAME2
         AND
         CITY = :CITY
         ORDER BY FNAME
```

Let's try to design a three-star index for this CURSOR. Much of the reasoning is the same as that for CURSOR41, but the substitution of the BETWEEN predicate for the = predicate will have significant implications. We will consider the three stars in reverse order, which arguably reflects the level of difficulty of understanding.

First the easy (albeit very important) star, the third star. As before, the index will qualify for the third star by ensuring that all the columns specified in the SELECT are in the index. No access to the table will be required and so synchronous reads will not cause a problem.

The index will qualify for the second star by adding the ORDER BY column, but this will only be true if it is added *before* the BETWEEN predicate column LNAME, for example, with index (CITY, FNAME, LNAME). There is only a single CITY value (= predicate), and so the result set will be provided in FNAME sequence simply by using this index without the need for a sort. But if the ORDER BY column is added *after* the BETWEEN predicate column LNAME, for example, with index (CITY, LNAME, FNAME), the index rows will *not* be in FNAME sequence and so a sort will be required; this can be seen in Figure 4.3. Therefore to qualify for the second star, the FNAME *must be*

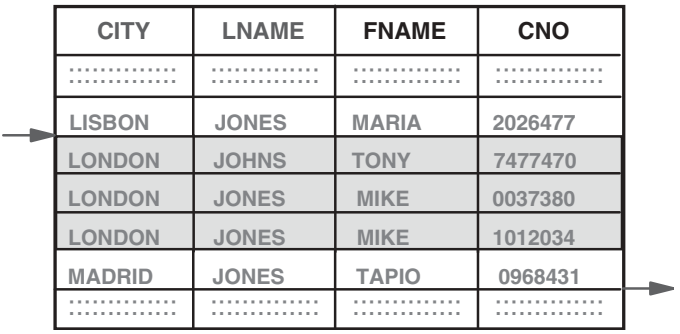


Figure 4.3 Effect of a range predicate on the sequence.

before the BETWEEN predicate column LNAME as in index (FNAME, . . .) or index (CITY, FNAME, . . .).

Regarding the first star, if CITY is the first column of the index, we will have a relatively thin slice to scan (MC = 1), depending on the CITY filter factor. But the index slice will be even thinner with the index (CITY, LNAME, . . .); now with two matching columns we only touch the index rows we really need. But to do this and so benefit from a very thin index slice, no other column (such as FNAME) can come between them.

So how many stars will our *ideal* index have? It can certainly have the third star, but, as we have just seen, we can have *either the first star or the second star, but not both!* In other words, we can either:

- Avoid a sort—by having the second star.

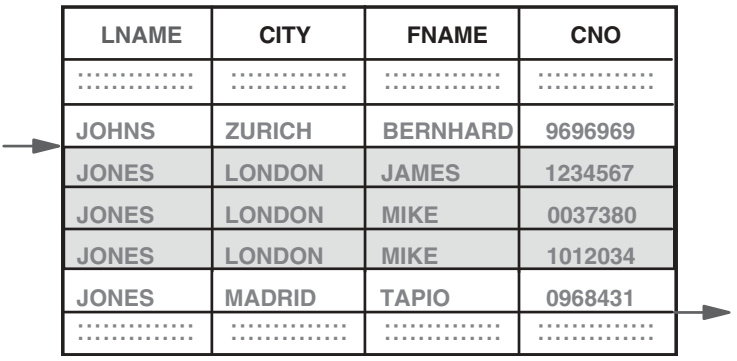
or

- Have the thinnest possible index slice, which will minimize not only the number of index entries to be processed but also any subsequent processing, in particular synchronous reads for the table rows—by having the first star.

The presence of the BETWEEN predicate in our example, or indeed any other range predicate, means *we cannot have both*; we cannot have a three-star index.

This implies therefore, that we have to make a choice between the first and second stars. Usually this isn't a difficult choice since, as we will see in Chapter 6, the first star tends to be the much more important star, although this is not always the case as we will see at that time.

Let's consider for a moment an index (LNAME, CITY, . . .) as shown in Figure 4.4. LNAME is a range predicate, which means it is the last column that can participate in the index matching process, as we saw in Chapter 3. The equal predicate CITY would *not be used* in the matching process. The result of this would be only *one* matching column—a much thicker index slice than we would have with the index (CITY, LNAME, . . .).



LNAME	CITY	FNAME	CNO
.....	.....	.....	.....
JOHNS	ZURICH	BERNHARD	9696969
JONES	LONDON	JAMES	1234567
JONES	LONDON	MIKE	0037380
JONES	LONDON	MIKE	1012034
JONES	MADRID	TAPIO	0968431
.....	.....	.....	.....

Figure 4.4 Only the third star for CURSOR43 with MC = 1.

ALGORITHM TO DERIVE THE BEST INDEX FOR A SELECT

The *ideal* index will be a three-star index for the reasons discussed above. We have seen, however, that with a range predicate this may not be possible; we may have to sacrifice (probably) the second star in order to achieve a thin index slice (the first star). The *best* index will then only have two stars. This is why we are careful to distinguish between *ideal* and *best*. The ideal in this case would just not be possible. Taking this into account, we can formulate the rules for creating the *best* (perhaps not *ideal*) index under all conditions. The result might have three stars or two stars.

First a fat index (third star) is designed, where the scanned index slice is as thin as possible (first star). If this index does not imply a sort (second star), it is a three-star index. Otherwise it will only be a two-star index, having sacrificed the second star. Another candidate should then be derived that eliminates sorting, thereby having the second star but having sacrificed the first star. One of the resulting two star indexes will then be the best possible index for the given SELECT.

The algorithm to derive the *best* index for a SELECT follows.

Candidate A

1. Pick the equal predicate columns that are *not too difficult* for the optimizer (discussed in Chapter 6). These are the first index columns—in any order.
2. The next index column is the column in the most selective range predicate, if any. Most selective means the lowest filter factor with the worst input. Only consider the range predicates that are not *too difficult* for the optimizer (discussed in Chapter 6).
3. Add the ORDER BY columns in the correct order (and with DESC if an ORDER BY column has DESC). Ignore columns that were already picked in step 1 or 2.

4. Add all remaining columns referred to by the SELECT statement, in any order (but start with the nonvolatile columns).

Example: CURSOR43

Candidate A will be (CITY, LNAME, FNAME, CNO).

Candidate A causes a sort for CURSOR43 because FNAME comes *after* the range predicate column, LNAME.

Candidate B

If candidate A leads to a sort with the given SELECT, candidate B is designed. By definition, the second star is more important than the first star for candidate B.

1. Pick the equal predicate columns that are not too difficult for the optimizer. These are the first index columns—in any order
2. Add the ORDER BY columns in the correct order (and with DESC if an ORDER BY column has DESC). Ignore columns that were already picked in step 1.
3. Add all remaining columns referred to by the SELECT statement, in any order (but start with the nonvolatile columns).

Example: CURSOR43

Candidate B will be (CITY, FNAME, LNAME, CNO).

We now have two candidates for the best index, one with the first star, one with the second star. To determine which will be the best index, we could analyze the performance of each index as we did at the beginning of this chapter. This would be quite time consuming, however, and Chapter 5 presents a simpler method (the QUBE) for estimating which candidate will provide the faster access path for the SELECT.

It is important to realize that all we have done so far is to design the *ideal* or *best* index. Whether this would be *practical* or not, at this stage we are not in a position to say.

Sorting Is Fast Today—Why Do We Need Candidate B?

Sorting has become much faster over recent years. These days much of the sort process takes place in memory, such that with the fastest current processors the CPU time per sorted row is roughly 10 μs. Thus, the elapsed time for sorting 50,000 records could be as short as 0.5 s, possibly acceptable as a response time component of an operational transaction but quite high as a contributor to the CPU time.

Because sorting is so fast with current hardware, candidate A will be as fast or faster than candidate B if a program fetches *all* the result rows. For the programmer this is the most convenient solution. Many environments provide flexible commands for browsing the result table.

Candidate B, however, may be much faster than candidate A if a program fetches only sufficient rows to fill a single screen, such as CURSOR44 in SQL 4.4. As discussed in Chapter 3, if there is no sort in the access path, the DBMS materializes the result rows FETCH by FETCH. This is why it is sometimes *very* important to *avoid* sorting (by using candidate B). *Producing the first screen with a two-star candidate A index* (requiring a sort) *may take far too long if the result table is large*. We should always keep in mind that the result table may be *extremely* large if the end user makes a typing error, for example.

The program using CURSOR44 will be very fast *if there is no sort* in the access path (assuming the columns LNAME and CITY are the first two columns of the index—in either order), even if the result table contains millions of rows. Each transaction *never* makes the DBMS materialize more than 20 result rows. We will later discuss how to implement the transaction that finds the *next* 20 result rows efficiently.

SQL 4.4

```
DECLARE CURSOR44 CURSOR FOR
SELECT     CNO, FNAME
FROM       CUST
WHERE      LNAME = :LNAME
           AND
           CITY = :CITY
ORDER BY   FNAME
WE WANT 20 ROWS PLEASE

OPEN CURSOR CURSOR4
FETCH CURSOR CURSOR4 ----- max 20 times
CLOSE CURSOR CURSOR4
```

IDEAL INDEX FOR EVERY SELECT?

Index design is quite straightforward if the best possible index is created for each SELECT unless an identical index already exists. This process, essentially the two-candidate algorithm described earlier, is mechanical and can be performed automatically, given the following input:

1. The SELECT statement
2. Database statistics (number of rows, number of pages, column value distribution, etc.)
3. The worst-case filter factors for each simple predicate and certain compound predicates
4. The current indexes

In the simplest approach, the current indexes are checked merely to avoid identical twins. In one AS/400 system, which used this approach, the DBMS

had created 60 indexes on one table. Many of these could have been dropped (and were indeed dropped) to make inserts faster, without making any query noticeably slower.

It is important to take a close look at the existing indexes after designing the best possible index (or indexes) for a new SELECT. It is possible that one of the existing indexes would be almost as good as the ideal index, particularly if a few columns were to be added *after* the existing index columns.

When analyzing how useful an existing index would be for a new SELECT, it should be borne in mind that superfluous indexes fall into three categories: totally superfluous indexes, practically superfluous indexes, and possibly superfluous indexes.

Totally Superfluous Indexes

AS/400 users noticed many years ago that if one query contained WHERE A = :A AND B = :B while another query had WHERE B = :B AND A = :A, the DBMS created two indexes: (A, B) and (B, A). One of these would be totally superfluous assuming no query contained range predicates for column A or column B. There is no need for two phone books, one sorted by LNAME, FNAME, the other by FNAME, LNAME if both the last name and the first name are always known. AS/400 users soon learned to standardize their WHERE clauses, but the problem reappeared with program generators.

Practically Superfluous Indexes

Assume index (LNAME, CITY, FNAME, CNO) already exists. The ideal index for a new SELECT consists of 14 columns, starting with the four columns of the existing index. Should the old index be dropped after creating the 14-column index? Some DBAs would hesitate to do so because the existing index is unique. However, it is *not* a primary key index or a candidate key index. It just happens to contain column CNO, the primary key. There is no integrity exposure when additional columns are added to this index. If the DBMS supports nonkey index columns or constraints to enforce uniqueness, columns may even be added to a primary key index or any index whose key values must be unique. So, this is purely a performance issue: Will a SELECT that currently uses the 4-column index run noticeably slower with the new 14-column index?

How much longer does it take to scan a 10,000-row slice to pick up 1000 index entries if the index row grows, say, from 50 to 200 bytes? The CPU time increases by an insignificant amount, but the I/O time is proportional to the number of *pages* accessed.

CPU time = 1000×0.1 ms + $10{,}000 \times 0.005$ ms = 150 ms
(1000 FETCH calls and 10,000 index rows in both cases)

Number of 4K leaf pages (4 columns): $1.5 \times 10{,}000 \times 50/4000 = 200$
(1.5 for free space)

Number of 4K leaf pages (14 columns) $1.5 \times 10{,}000 \times 200/4{,}000 = 800$

Sequential read time (4 columns) $= 200 \times 0.1$ ms $= 20$ ms

Sequential read time (14 columns) $= 800 \times 0.1$ ms $= 80$ ms

As sequential processing remains *CPU bound*, there is no noticeable difference in the elapsed time. The current 4-column index becomes superfluous when the new 14-column index is created. Chapter 15 discusses CPU estimation in considerable detail.

Possibly Superfluous Indexes

A common situation: The ideal index for a new SELECT is (A, B, C, D, E, F). The table already has index (A, B, F, C). Is the new index superfluous if the existing index is replaced by index (A, B, F, C, D, E)? In other words is the existing index adequate for the new SELECT, if columns D and E are added to it to make the access path index only?

The *proposed* ideal index may be *better* than index (A, B, F, C, D, E) in two respects:

1. There may be more matching columns.
2. It may prevent a sort.

Both advantages are influenced by the number of rows that must be scanned in the index slice. This difference can be converted to milliseconds, as we have done in this chapter, or more easily by the Quick Upper-Bound Estimate (QUBE) discussed in Chapter 5. More often that not, the estimates will indicate that a new index is *not* needed; an existing index with the new columns added at the end, will be adequate for the new SELECT.

COST OF AN ADDITIONAL INDEX

If 100 different SELECT statements access a table and the best possible index is designed for each SELECT, we may end up with dozens of indexes on the table even if no identical twins are created. INSERTs, UPDATEs, and DELETEs to the table may then become too slow.

Response Time

When the DBMS adds a table row, it must add a corresponding row to *each* index. With current hardware, adding a row to an index typically adds 10 ms to the elapsed time of the INSERT call because a leaf page must be read from a disk drive. When a transaction inserts a row to a table with 10 indexes, index maintenance may add 10×10 ms $= 100$ ms to the response time, which is probably acceptable. However, if one transaction adds 20 rows to a table with 10 indexes, index maintenance may require 181 random reads, 1.8 s. This

estimate assumes that the new index rows go to the same leaf page in one index (an ever-increasing key) and to 20 different leaf pages in the 9 other indexes. From the *response time* point of view, a large table with many inserts (or deletes) *per transaction* may not tolerate 10 indexes. Furthermore, from the *disk drive load* point of view, a large table with more than 10 inserted rows *per second* may not tolerate 10 indexes. Let us take a closer look at the second issue.

Drive Load

Sooner or later, the modified leaf pages must be written to disk drives. Because database writes are asynchronous, they do not affect the response time of the transaction. However, they do increase the drive load. RAID 5 increases the impact as each random update of a page results in access to *two* drives. Each access may take 12 ms because a whole track must be read and written back: one seek (4 ms) and two rotations (2×4 ms). Thus, the total increase in *drive busy* caused by writing a modified page to the disk drives is 24 ms. The corresponding value for RAID 10 (striping and mirroring) is 2×6 ms $= 12$ ms.

If the INSERT rate to a table is high, drive load may be the primary problem, which sets a limit to the number of indexes on a table. DELETEs cause the same disk load as INSERTs, so massive delete jobs are another serious concern. UPDATEs affect only the indexes whose columns have been changed.

Let us assume a RAID 5 disk server has 128 drives: 112 active, 16 spares. The database (tables and indexes or their partitions) is striped over the active drives. Read cache is 64 GB, write cache 2 GB.

In table TRANS, shown in Figure 4.5, new rows go the end of the table and its clustering index; they do not cause a large number of disk reads and writes because many rows are inserted to a page before the page is written to disk. The

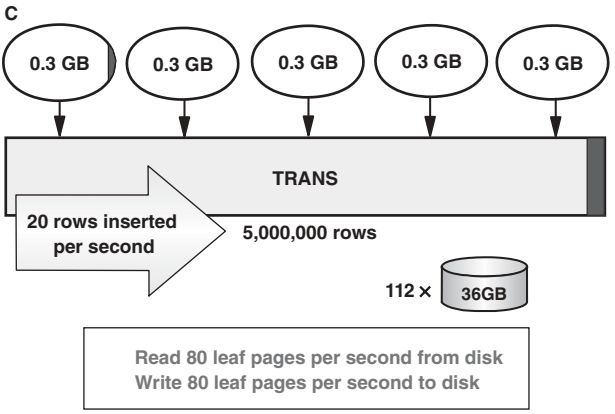


Figure 4.5 Stressful for the disk server.

problem arises from the random inserts to the four indexes; every new index row may cause a disk read and a disk write, 80 random reads and 80 random writes per second in total (4×20).

Let us first ignore both the read cache and the write cache. In this worst-case scenario, the drive load caused by the four indexes is

$$80 \times (6 \text{ ms} + 24 \text{ ms}) = 2400 \text{ ms/s} = 2.4 = 240\%$$

If the four indexes are striped over 112 drives, their contribution to the average drive load is $240\%/112 = 2\%$. This value should be compared against the level at which a drive is considered to be overloaded. Ideally, the average drive load should be as low as 25%. Then, according to queuing theory, the average drive queuing time would be 3 ms. Thus adding 2% to the average drive busy is probably tolerable but not insignificant.

How much of this drive load would be eliminated by the read cache and the write cache?

The read cache, 64 GB, seems large compared to the total size of the four indexes (1.2 GB), but if the access pattern is truly random, the time between references to any one of the 75,000 leaf pages in each index is not far from the average, which is

$$75{,}000 \times 50 \text{ ms} = 3750 \text{ s} = 1 \text{ h}$$

when the insert rate is 20 rows per second. If the average *read cache* residency time is 30 min, not many disk drive reads will be saved by the read cache. If the *write cache* holding time is shorter, 10 min, for instance, its effect will be even less significant: a write to drive would be saved only if a leaf page is updated more than once in 10 min. The actual average drive busy caused by the four indexes may thus be almost 2%. The caches bring much bigger drive load savings if the access pattern is not random. Leaf pages that are updated more than ten times per hour could stay in both caches for a long time.

RAID 10, mirroring and striping but no parity bits, would reduce the drive busy per modified page from 24 to 12 ms, but the number of drives would increase. The 256 drives with RAID 5 would have roughly the same effect.

A rule of thumb can be derived from this example. Indicator L predicts the contribution of the indexes on a table to the *average drive load* with RAID5:

$$L = N \times I/D$$

where N = the number of indexes with random inserts
I = insert rate (table rows per second)
D = the number of disk drives (excluding spares)

If $L < 1$, the increase in drive load is not an issue; it is probably less than 2%.

If L is between 1 and 10, the increase in drive load may be noticeable.

If $L > 10$, drive load is likely to be a problem, unless cache hit ratios are high.

In the example above,

$$L = 4 \times 20/112 = 0.7$$

From the drive-load point of view, a table with a low insert rate, such as a typical CUST table, would tolerate dozens of indexes; the limiting factor would be the response time of insert transactions.

If drive load is a problem, the obvious solution would be to try to combine indexes. One index with 10 columns causes much less drive load than 2 indexes, each with 6 columns.

Disk Space

If there are more than 10 million rows in a table, the cost of disk space for the indexes may become an issue. The charge for outsourced hardware depends mainly on two factors: consumed CPU time and allocated disk space. At the time of writing, the monthly rent for disk space in a high-performance, fault-tolerant disk server may be U.S.$50 per gigabyte. Installations that own their own hardware may have an internal billing rate per gigabyte of disk space. Database designers should be aware of the current cost of disk space. All too often, a request to add a 100-byte column to an index, for instance, is rejected simply because it *feels* like wasting disk space.

This is what database guru Jim Gray (Microsoft Research) said in a recent interview (5):

> *I actually tracked the price of disks pretty carefully, and the price of disks has gone down by a factor of a hundred since the vacuum cleaner idea failed (1996). The price of disk arms has gone down by about a factor of ten in that time. These ratios absolutely change things... (page 5)*

To take an example, a new index with 400 bytes of user data is proposed for a table with 10 million rows. Should we be concerned about the disk space?

The proposed index (without RAID overhead) requires about $1.5 \times 10{,}000{,}000 \times 400$ bytes $= 6$ GB of disk space. The cost may be perhaps $300 per month. This is probably not a showstopper if adding columns to the *current* indexes is unable to provide acceptable response times.

Buffer pools or disk caches should be increased as indexes become larger, otherwise the number of nonleaf page I/Os will increase. With current hardware prices, the memory cost for the nonleaf pages would be about $300 per month in our example.

The *overhead coefficient* of fat indexes (which tend to be unique or almost unique) depends primarily on the amount of distributed free space. A value of 1.5 is probably high enough to allow for 25% of free space per leaf page. With an 8K leaf page, this means that five 400 byte index rows can be added to each leaf before a split will occur. Chapter 11 recommends free space specifications and reorganization frequencies for indexes according to row length and insert patterns.

The length of the pointer to the table row depends on the DBMS. Direct pointers are short, normally less than 10 bytes. Symbolic pointers may be longer; these are used in indexes that point to an index holding the table rows.

RECOMMENDATION

Even with the current low cost of disk space, it will be unwise to mechanically implement the best possible index for every SELECT because index maintenance may make some programs too slow or overload the disk drives (which will affect *every* program). The best possible index (derived with the two-candidate method or by an index design tool) is a good starting point, but the three types of superfluous indexes should be considered before deciding to create the ideal index for a new SELECT.

Even if it were possible to derive the best index for every new SELECT, it is more likely that in the real world this would be done only for the SELECT statements that are found to be too slow, either by estimation or by measurement, because of inappropriate indexing; but how are those statements to be found? Chapter 5 recommends two simple methods for detecting slow access paths at an early stage, when the SQL statements are being written or generated. Chapter 7 discusses an approach to exception monitoring, which will reveal significant access path problems as well as other performance problems in a production system.

EXERCISES

4.1. Derive index candidates A and B for the SELECT statement in SQL 4.5.

4.2. For each alternative, count the number of index rows that must be accessed by one transaction with the worst input. The ORDERITEM table has 100,000,000 rows.

SQL 4.5

```
SELECT     A, B, D, E
FROM       ORDERITEM
WHERE      B BETWEEN :B1 AND :B2   (FF = 1...10%)
           AND
           C = 1                   (FF = 2%)
           AND
           E > 0                   (FF = 50%)
           AND
           F = :F                  (FF = 0.1...1%)
ORDER BY   A, B, C, F
WE WANT 20 ROWS PLEASE
```

Chapter 5

Proactive Index Design

- Introduction to the two fast and easy-to-use techniques that will be used extensively throughout this book: the Basic Question (BQ) and the Quick Upper-Bound Estimate (QUBE)
- Application of the principles discussed in Chapter 4 to the SQL application design process
- Use of these techniques to determine the merits and demerits of potential index designs and to determine improvements necessary to provide adequate SQL performance
- Assessment of the suitability of any improvements considered
- Discussion of when the QUBE should be used

DETECTION OF INADEQUATE INDEXING

As soon as the specifications for a program are complete, we should determine whether the current indexes are adequate for the new program. To do this, we will now consider two simple, fast, and practical approaches, namely

1. Basic Question (BQ) (8)
2. Quick Upper-Bound Estimate (QUBE)

BASIC QUESTION (BQ)

Even the busiest programmer has time to do this evaluation. For each SELECT statement the answer to the following question must be considered as shown in the following steps:

Is there an existing or planned index that contains all the columns referenced by the WHERE clause (a semifat index)?

- If the answer is no, we should first consider adding the *missing predicate columns* to an *existing* index. This will produce a semifat index where,

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

even if the matching index process is inadequate (the first star), index screening will take place to ensure table access will only occur when it is *known* that all the selection criteria have been met.

- If this does not result in adequate performance, the next alternative is to add enough columns to the index to make the access path index only. This will produce a fat index that will eliminate *all* table access.
- If the SELECT is still too slow, a new index should be designed using the two-candidate algorithm introduced in Chapter 4. This, by definition, will be the *best* index possible.

How do we determine whether the first alternative (semifat index) or the second one (fat index) makes the SELECT fast enough even with the worst input?

If access to the production database or a production-like test database is available, one index may be created at a time and the response time measured with the worst input. To ensure that the measured response times are close to those in normal production, the buffer pools must be considered and the buffer pool hit ratios of each transaction observed. The first test transaction will probably find no pages in the buffer pool, so the number of disk reads will be higher than in a normal environment. Furthermore the first transaction accessing a new index must wait for the files to be opened. On the other hand, the second transaction accessing the same index may have a buffer pool hit ratio of 100% (no disk reads) if the same input is repeated. To obtain representative response times, each index alternative should be tested after warm-up transactions have been submitted; a few typical inputs (but not the worst case) are entered to open the files and to bring many of the nonleaf index pages into the database buffer pool. Then the response time of the transaction with the worst input will probably have a representative response time, at least as far as the CPU and disk read service times are concerned.

It is actually far less tedious to employ the second technique, which will be addressed shortly, the QUBE to evaluate the alternatives.

If neither the measurement nor the QUBE approach is undertaken, the fat index alternative should be adopted, followed by the use of exception reports immediately after production cutover to detect the cases where even a fat index is not adequate. Then, if necessary, the best possible (new) index will need to be designed for the slow SELECTs.

Warning

An affirmative answer to the BQ *does not guarantee* adequate performance. Remember that the objective of the BQ is to simply ensure that we can at least do index screening in order to minimize the access to the table—no more.

For example, assume a SELECT with WHERE B = :B AND C = :C and the only useful index for the SELECT being (A, B, C). The index does indeed contain all the predicate columns for the SELECT, and so the SELECT does not raise an alarm with the BQ. The access path, however, will be a full index

scan—much too slow if there are say more than 100,000 rows in the table. Index screening in itself does not imply that the index has *any* of the three stars.

Nevertheless, according to our experience, a significant proportion of index problems found after cutover would have been detected early by considering BQ.

Applying BQ to single-table selects is straightforward. A join on the other hand must be mentally broken down into several single-table cursors before the BQ can be applied. This is a much more complex process, one that we shall consider in great detail in Chapter 8.

QUICK UPPER-BOUND ESTIMATE (QUBE)

In the initial evaluation phase (will this SELECT be fast enough with the current indexes?), the QUBE is more time consuming than BQ, but it reveals *all* performance problems that relate to index or table design—assuming that the worst-case filter factors used for each predicate are reasonably close to the actual worst-case filter factors. By definition the QUBE is pessimistic (upper-bound); it sometimes makes false alarms, unlike BQ which isn't able to detect some problems.

The objective of the QUBE is to reveal potentially slow access paths at a very early stage, as the program is being designed, written, or generated. In order to be usable in real projects, any statement level prediction formula must be so simple that the additional effort of the estimation process is at an acceptable level. A few years ago, a Finnish company used a former version of the QUBE to evaluate all the transactions and batch programs for a new application. According to its records, the estimation process added only 5% to the implementation effort; however, numerous index improvements and indeed a few program design changes were identified and implemented as a result. Post-cutover tuning was reduced by 90% compared to similar projects designed without the QUBE. The version of the QUBE discussed in this book is also somewhat less time consuming than that used in this company's evaluation.

The output of the quick estimate is the local response time (LRT), that is, the elapsed time within the database server. The LRT of a traditional transaction in a single-tier environment (where the program issuing the SQL calls resides in the same machine as the database) is the response time for one interaction between the user and the database server, excluding all network delays. In multitier environments (client/server), the communication time between tiers is also excluded. The LRT of a batch job means the elapsed time of the job execution. Any queuing time in the job queue is also excluded.

Figure 5.1 shows the components of the LRT together with the subcomponents of interest in the QUBE process.

Service Time

In simple cases (those for which I/O time and CPU time do not overlap), service time is the sum of the CPU time and the random read time without drive queuing. If there is no contention, the local response time will be equal to the service time.

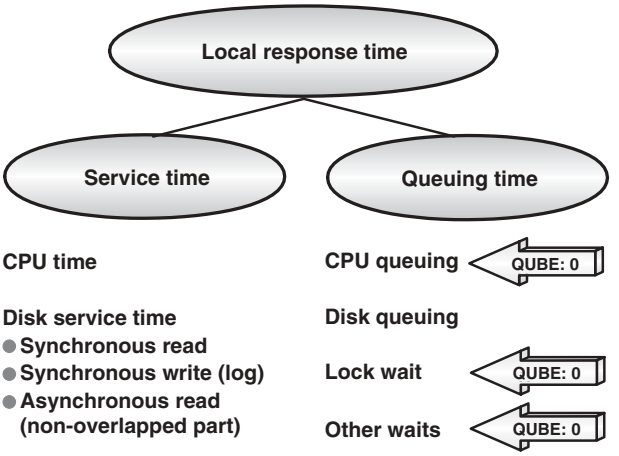


Figure 5.1 Local response time is the sum of the service time and the queuing time.

Queuing Time

In a normal multiuser environment, concurrent programs cause many kinds of contention for the resources required; they will therefore have to queue for these resources. The following resources are within the domain of the LRT:

- CPU queuing time (all processors are busy with high-priority tasks).
- Disk drive queuing (the drive containing the requested page is busy).
- Lock waits (the requested table page or row is locked at an incompatible level).
- Multiprogramming levels, number of threads, or other throttles have reached their limits (these are system values designed to prevent resource overload situations; it is not necessary to understand these for the purpose of this book).

With the exception of disk drive queuing, the QUBE ignores queuing in order to provide a simple estimation procedure for the areas that will normally have far greater performance implications. We will discuss this area, together with some other simplifying assumptions behind the QUBE formula in more detail later.

The result is a very simple estimation process with merely two input variables, TR and TS, with a third variable, F, when necessary. These are able to take into consideration the processing and I/O costs of the SQL, and are what primarily influence the index design process. They are shown in Figure 5.2 together with the values used in the QUBE. If the QUBE is being used to compare the performance of alternative access paths (Comparison, in Figure 5.2), the number of FETCH calls will be constant and therefore can be ignored. If it is being used to determine the response time (Absolute), then they would be included. For the

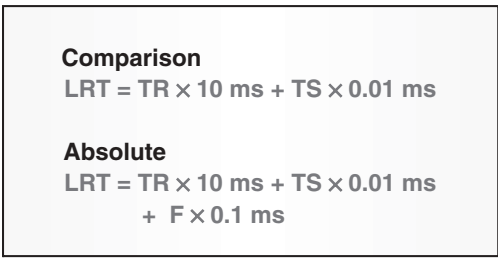


Figure 5.2 Quick Upper-Bound Estimate (QUBE).

sake of completeness, the FETCH cost will normally be included in the examples shown in this book.

Essential Concept: Touch

When the DBMS reads one *index row* or one *table row* the cost is, by definition, one *touch*: index touch or table touch. If the DBMS scans a slice of an index or table (the rows being read are physically next to each other), reading the first row infers a *random touch*. Reading the next rows take one *sequential touch* per row. With current hardware, sequential touches are much cheaper than random touches. The cost of an *index touch* is essentially the same as the cost of a *table touch*.

We will now consider the index and table sequential touches in more detail.

Reading a Set of Sequential Index Rows

What does *physically next to each other* mean? All the index rows in an index are chained together by pointers. The order is strictly defined by the key of the index. When several index rows have the same key value, they are chained according to the pointer value. In a traditional (and from one point of view, ideal) setup, the chain starts in LP1 (leaf page 1), then continues through LP2 and so on. Furthermore, (assuming 12 LPs per track—current devices usually have many more), the leaf pages form a contiguous file; LP1–LP12 could be on the first track of a disk cylinder, LP13–LP24 on the next cylinder, and so forth. When the first cylinder is full, the next LPs are placed on the first track of the next cylinder and so on. In other words, there are no other pages between the leaf pages.

Reading a set of sequential index rows—an index slice, or the rows that correspond to a single key or range of key values—is now very fast. One disk rotation reads several leaf pages into memory and a short seek is required only when the position moves to the next cylinder.

There are at least three factors that can destroy this perfect order:

1. If there is not enough space in a leaf page for an index row being inserted, a leaf page split must be performed. Afterwards the chain will still connect the index rows in the correct sequence, but that sequence no longer corresponds to the physical sequence; some "supposedly" sequential touches then become random touches. A reorganization of the index will restore perfect order once more.
2. Unexpected growth may fill the original continuous area (extent or equivalent). The operating system will then find another continuous area that is linked to the first one. This is not a huge consideration but jumping to the next extent will mean a random touch.
3. RAID 5 striping stores the first few leaf pages on one drive and then continues on the next drive. This causes additional random reads, but in fact the benefits of doing this are more important; with an intelligent disk server the next leaf pages can be brought into the disk cache in parallel from several drives, and so the I/O time per leaf page is much reduced. Furthermore, a very active index is less likely to overload a disk drive because the I/O requests are spread across several drives in a RAID 5 array.

Regardless of the above, two index rows are considered to be *physically next to each other* (in the QUBE) if they are next to each other in the pointer chain (or in a nonunique index, if the pointers representing the index rows are next to each other). This simply means that the QUBE assumes all the indexes are in perfect order; this will be discussed further in Chapter 11.

Reading a Set of Sequential Table Rows

Reading a set of sequential table rows comes in two flavors.

1. A *full table scan* starts with TP1 (table page 1), reads all the table rows there, and then continues with TP2. No particular order is followed; the table rows are simply read as they were placed in each table page.
2. A *clustered index scan* reads the first index row of an index slice and then retrieves the corresponding table row. This is then repeated with the second index row and so forth. If the index rows and the table rows are in exactly the same sequence (cluster ratio 100%), all the table row touches except the first one will be sequential. The table rows are not chained together in the same way as the index rows. The order of the table rows in a table page is irrelevant as long as the next table row to be touched is on the same or the next physical table page.

As with indexes, the traditional way to store a table is to keep all the table pages together in one continuous area. The factors that cause disorganization or fragmentation are similar to those we saw with indexes, but with two differences:

1. If a table row being inserted does not fit on its *home page* as defined by the clustering index, the existing rows in that page are not normally moved. Instead, the new row is placed on another page as close as possible to the home page. Additional random I/Os to this second page may make clustering index scans slower, although these will be avoided if the displaced table row is close to its home page because sequential prefetch reads a number of pages at a time into the database buffer pool. Even if sequential prefetch is not used, additional random I/Os will only take place if the second page has been overwritten in the database buffer pool.
2. A row may be updated such that the row becomes so long that it no longer fits in the same page. The DBMS must then move the extended row to another page, leaving an anchor point on the original page, which points to the new page. This creates additional random touches when the extended row is accessed.

Table reorganization will restore the order of a table thereby minimizing unnecessary random touches. Consequently, table rows are considered to be *physically next to each other* in the QUBE if they reside on the same or consecutive table pages. In other words, all tables, as with indexes, are assumed to be in perfect order.

It has been necessary to make several worst-case assumptions to make the QUBE simple enough to be used quickly and easily. A few optimistic assumptions were necessary as well. According to the QUBE, scanning one index or table slice will require *one random touch only*, despite the issues raised above. The database specialists should monitor the need for reorganization so that the optimistic assumptions we make should be justified.

Counting Touches

As the *touch* is so important to the QUBE, we will now explain how to determine the number of index and table touches, both random and sequential.

Random Touches

We will first consider the difference between a *disk read* and a *touch*. A disk read accesses a page while a touch accesses a row. One random disk read brings a whole page, often many rows, into the database buffer pool, but by definition, two consecutive random touches are unlikely to touch the same page. Therefore, the QUBE elapsed time for a random touch is the same as the average time for a random read from disk, 10 ms. Random reads are synchronous, but with current fast processors the CPU time required for a random touch can be ignored when estimating the elapsed time; it is normally less than 0.1 ms.

Sequential Touches

A *sequential touch* reads the next row in physical sequence, which should either be on the same page or the next page. As the CPU time and the I/O time overlap with a sequential read (both the DBMS and the disk controller normally read a number of pages ahead), the elapsed time for a sequential touch will be the *larger* of these two components. The QUBE elapsed time for a sequential touch is 0.01 ms.

When counting touches, we will apply the following rules, again in the quest for simplicity.

1. Ignore index nonleaf pages because they are assumed to stay in the database buffer pool or at least in the read cache of the disk server.
2. Assume that the DBMS goes directly to the first index row of an index slice (ignore the time for binary searches or other techniques that may be used to locate the index row in the leaf page).
3. Ignore any savings due to skip-sequential read. This can be a *very* pessimistic assumption and so will be discussed later.
4. Assume that all the indexes and tables are in perfect order. As described above, this can be an optimistic assumption unless the state of the indexes and tables is adequately monitored.

When counting *index touches*, it usually helps to visualize an index as a minitable that has as many rows as the table to which it points, in perfect order by the index key.

When counting *table touches*, we should assume that the table rows are perfectly sorted in the table pages. This sequence depends on how the table is reorganized. A full table scan (with N rows) can be assumed to require one random touch and $N - 1$ sequential touches.

The figure indicated in Figure 5.2 for sequential touches (0.01 ms) *refers to the processing necessary to be able to decide whether the row will be rejected or accepted*—all rows being examined, either in the index or the table, must at least go through to this stage. Rejected rows will need no further processing.

FETCH Processing

Accepted rows, the number of which will be determined by the number of FETCH calls issued (unless multirow FETCH is available), will require a great deal more processing. Remember that TS has *excluded* this additional processing; the third input variable, the F component of the LRT, now takes it into account. As shown in Figure 5.2, this cost is an order of magnitude *greater* than the cost of TS; on the other hand, it is very much *smaller* than the cost of TR.

If the QUBE is being used to compare alternative access paths, a table scan compared to one using a particular index, for example, the F parameter is irrelevant because it would be the same in each case; all that needs to be considered are TR and TS as described above. If the QUBE is being used to

determine the LRT, however, the F parameter may be significant, depending on the number of FETCH calls.

One of the activities that *may* have to take place for an accepted row is that of sorting; the overall cost of this will usually be proportional to the number of FETCHed rows. In most cases, this will be very small compared to the other processing that takes place for the accepted rows and so will be "hidden" within the F cost. There are occasions, however, where this will not be true, as we will see in Chapter 8. Apart from these exceptions, we will assume the sort cost to be included in the F parameter.

To avoid any confusion that might arise with regard to counting the number of FETCH calls, please note that in our examples we will ignore the one that causes "no more qualified rows" to be returned; this is purely to simplify the process, so, for example, a filter factor of 1% on a table containing 10,000 rows would expect to retrieve 100 rows—we would assume F = 100, not 101.

QUBE Examples for the Main Access Types

Example 5.1: Primary Key Index Access

Index Columns

We have seen how important index *matching* and index *screening* are with regard to performance. We have also discussed the use of an index to avoid the necessity of a sort. In order to fully appreciate how index columns are being used in these three ways in this chapter, ensure that you identify most carefully:

Matching columns
Screening columns
Sort avoidance columns

The local response times will be shown for each SQL example to enable comparisons to be easily made.

Although this is the simplest of examples, Figure 5.3 shows very clearly the touches taking place to both the index and the table. The arrows on both sides of the index show the index slice that has to be scanned (in this case a single index row).

SQL 5.1

```
SELECT   CNO, LNAME, FNAME
FROM     CUST
WHERE    CNO = :CNO
```

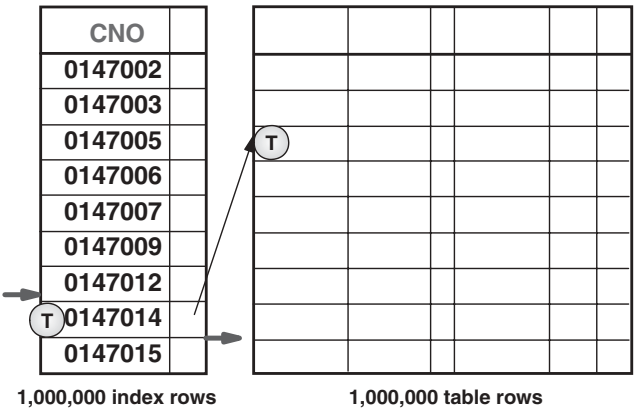


Figure 5.3 Primary key index access.

Reading a table row through the primary key index requires one random touch to the index and one random touch to the table.

```
Index  CNO            TR = 1
Table  CUST           TR = 1
Fetch  1 × 0.1 ms

LRT                   TR = 2
                      2 × 10 ms + 0.1 ms
                      = 20 ms
```

Example 5.2: Clustering Index Access

SQL 5.2

```
DECLARE CURSOR52 CURSOR FOR
SELECT     CNO, LNAME, FNAME
FROM       CUST
WHERE      ZIP = :ZIP
           AND
           LNAME = :LNAME
ORDER BY   FNAME
```

In Figure 5.4 let us assume we have 1000 Joneses in the area code (ZIP) 30103. This two-star index for CURSOR52 accesses a thin index slice with two matching columns; no sort is required as the index provides the required sequence; index only is of course not possible.

How long will it take to scan the index slice of 1000 rows? It takes a random touch to find the first index row in the slice and then the DBMS reads forward

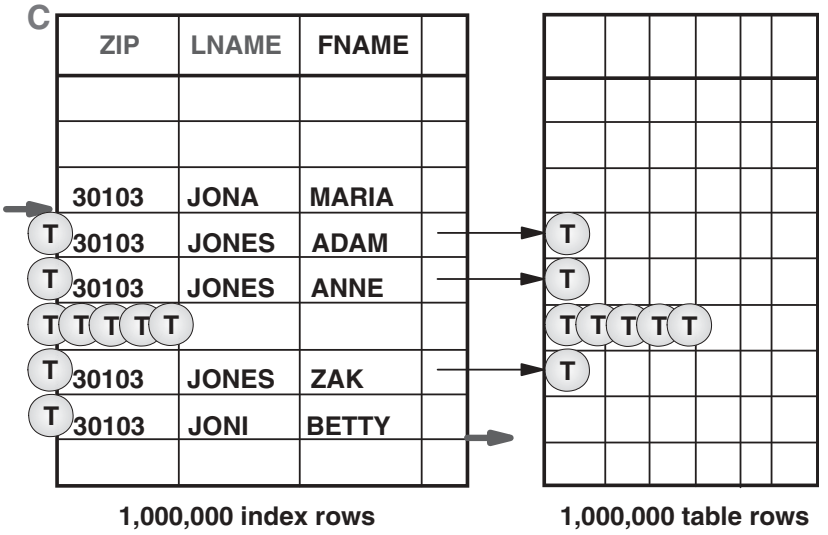


Figure 5.4 Clustering index access.

until it finds an index row that does not have the right values for the columns ZIP and LNAME. Including the last touch to a nonqualifying index row, the number of sequential touches to the index is 1000. Thus, scanning the index slice takes, according to the QUBE, 1×10 ms $+ 1000 \times 0.01$ ms $= 20$ ms.

Because column CNO is not in the index, it must be read from the table. As the index is clustering and assuming the 1000 table rows are adjacent, the cost of scanning the table slice will be 1×10 ms $+ 999 \times 0.01$ ms $= 20$ ms (we described how to count touches earlier in this chapter). In addition, there will be the cost of the FETCH calls, making the total elapsed time, according to the QUBE, of 140 ms. The cost of the table access is very low because sequential touches are very cheap. The largest component is the processing associated with the FETCH calls.

```
Index  ZIP, LNAME, FNAME   TR = 1      TS = 1000
Table  CUST                TR = 1      TS = 999
Fetch  1000 × 0.1 ms

LRT                        TR = 2      TS = 1999
                           2 × 10 ms   1999 × 0.01 ms
                           20 ms + 20 ms + 100 ms = 140 ms
```

The number of sequential touches to the table as shown above gives a misleading impression of the accuracy of the QUBE figures. We have written the TS figure as 999 simply to show how we can count the touches in an example such as this. In future, we will not be quite so precise; it will be sufficient to show the figure rounded up to 1000.

Example 5.3: Nonclustering Index Access

SQL 5.3

```
DECLARE CURSOR53 CURSOR FOR
SELECT     CNO, LNAME, FNAME
FROM       CUST
WHERE      ZIP = :ZIP
           AND
           LNAME = :LNAME
ORDER BY   FNAME
```

If the table rows are not accessed via the clustering index as in Figure 5.5, the 1000 table touches are now random.

```
Index  ZIP, LNAME, FNAME   TR = 1          TS = 1000
Table  CUST                TR = 1000       TS = 0
Fetch  1000 × 0.1 ms

LRT                        TR = 1001       TS = 1000
                           1001 × 10 ms    1000 × 0.01 ms
                           10 s + 10 ms + 100 ms = 10 s
```

Adding CNO to the index to make it a three-star index is now an excellent idea because it would reduce the local response time from 10 to 0.1 s.

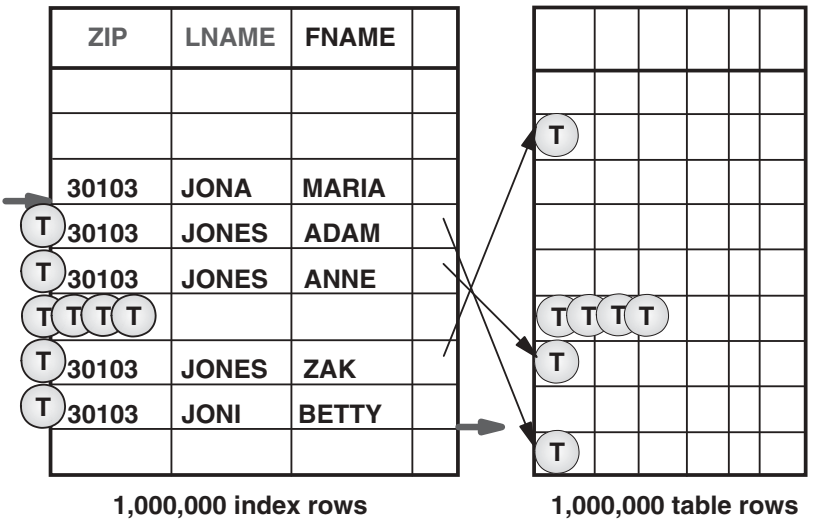


Figure 5.5 Nonclustering index access.

```
Index ZIP, LNAME, FNAME, CNO  TR = 1     TS = 1000
Table CUST                    TR = 0     TS = 0
Fetch 1000 × 0.1 ms

LRT                           TR = 1     TS = 1000
                              1 × 10 ms  1000 × 0.01 ms
                              10 ms + 10 ms + 100 ms = 120 ms
```

Example 5.3 shows just how dangerous a nonclustering index access can be. It also shows how important the third star for the index can be.

Example 5.4: Clustering Index with Skip-Sequential Table Access

SQL 5.4

```
SELECT     STREET, NUMBER, ZIP, BORN
FROM       CUST
WHERE      LNAME = 'JONES'
           AND
           FNAME = 'ADAM'
           AND
           CITY = 'LONDON'
ORDER BY   BORN
```

In Example 5.4 the query will use index (LNAME, FNAME, BORN, CITY), which is the clustering index. How many random and sequential touches will be required to access all the Adam Joneses who live in London?

Two matching columns will be used in the index scan. According to the data in Figure 5.6, there will only be two touches to the table instead of four because the CITY column will be used for screening. These two table touches will be *skip-sequential* (if indeed we can use this term for a mere two table touches), because the table rows will be in the clustering index sequence. The two rows will not be contiguous because of the column BORN before CITY in the index. Being skip-sequential, they are counted as random reads in the QUBE. We will discuss the impact of ignoring the benefit of skip-sequential read in Chapter 15.

```
Index LNAME, FNAME, BORN, CITY   TR = 1      TS = 4
Table CUST                       TR = 2      TS = 0
Fetch 2 × 0.1 ms

LRT                              TR = 3      TS = 4
                                 3 × 10 ms   4 × 0.01 ms
                                 30 ms + 0.04 ms + 0.2 ms = 30 ms
```

CHEAPEST ADEQUATE INDEX OR BEST POSSIBLE INDEX: EXAMPLE 1

Having looked at a few examples of how the QUBE can be used to very easily estimate some of the more important access methods, we will now show how we

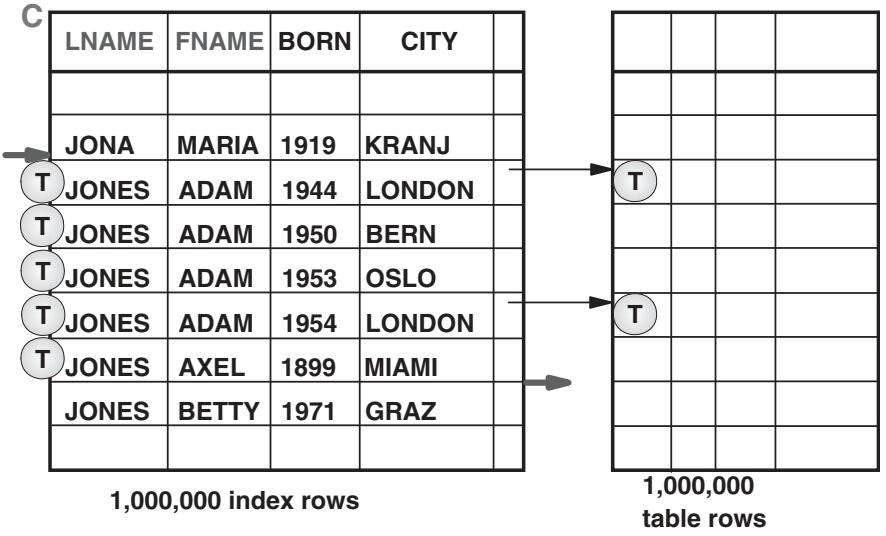


Figure 5.6 Clustering index with skip-sequential table access.

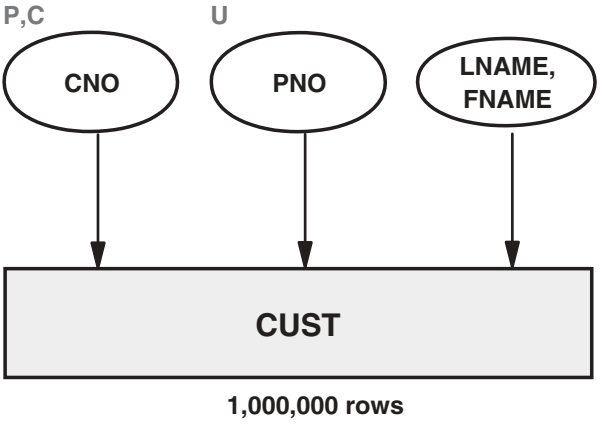


Figure 5.7 Cheapest adequate index or best possible index: Example 1A.

may use the two techniques (BQ and QUBE) in the whole index design approach discussed in Chapter 4.

Figure 5.7 shows the current indexes for the customer table. The user chooses both the last name (LNAME) and the city name (CITY) from pop-up windows before starting the transaction, which then opens CURSOR55 (refer to SQL 5.5).

The only index that appears to be useful is the index (LNAME, FNAME). We will analyze the use of this index with the cursor to determine whether performance will be adequate.

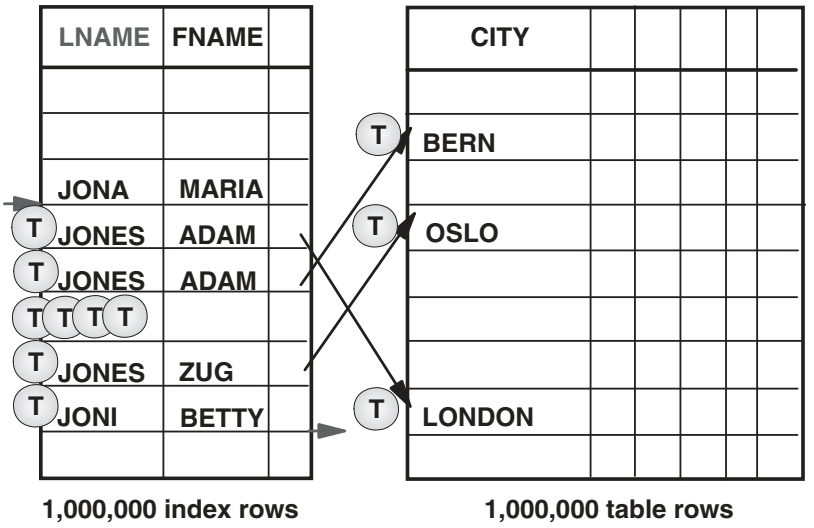


Figure 5.8 Cheapest adequate index or best possible index: Example 1B.

We will use the BQ to determine whether the index (LNAME, FNAME) will provide adequate performance for CURSOR55 (SQL 5.5) (Fig. 5.8). If necessary, we will then estimate the local response time for a transaction that FETCHes all the result rows. The assumptions are:

1. There are one million table rows.
2. The only suitable index is (LNAME, FNAME).
3. The maximum filter factor for predicate LNAME =: LNAME is 1%.
4. The maximum filter factor for predicate CITY =: CITY is 10%.
5. The maximum size of the result table is 1000 rows (10% of 1% of 1,000,000).
6. The table rows are stored in CNO order [primary key index (CNO) is the clustering index or the table is frequently unloaded and reloaded after a sort by CNO].

SQL 5.5

```
DECLARE CURSOR55 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME = :LNAME
            AND
            CITY = :CITY
ORDER BY    FNAME
```

Basic Question for the Transaction

The BQ asks the question:

> Is there an existing or planned index that contains all the columns referenced by the WHERE clause?

In other words do we, or will we, have a *semifat index*? The answer is clearly *no*, at least as far as the current indexes are concerned. This doesn't necessarily mean we have a problem; performance might still be satisfactory. But a warning signal has at least been raised. We should determine if we *do* have a problem; if we have, how serious would it be if we decided to go ahead anyway? should we add the missing predicate columns to the index? This would provide a positive answer to BQ, but we should bear in mind this still does not guarantee good performance.

Quick Upper-Bound Estimate for the Transaction

The result rows are obtained in the requested order (ORDER BY FNAME), when the DBMS chooses index (LNAME, FNAME); no sort is needed. Therefore, *the DBMS will not do early materialization*: OPEN CURSOR generates no touches but each FETCH generates touches to both the index and the table, some of which will return a result row to the FETCH and some (the majority) that won't. A matching index scan would take place with *one* matching column. The filter factor of this column is 1%. According to the QUBE, the local response time of the transaction that performs the following SQL

```
OPEN CURSOR   1001 × FETCH   CLOSE CURSOR
```

will be:

```
Index LNAME, FNAME    TR = 1           TS = 1% × 1,000,000
Table CUST            TR = 10,000      TS = 0
Fetch 1000 × 0.1 ms

LRT                   TR = 10,001      TS = 10,000
                      10,001 × 10 ms   10,000 × 0.01 ms
                      100 s + 0.1 s + 0.1 s = 100 s
```

The huge problem here, of course, is that the DBMS must read 10,000 nonadjacent table rows. This takes a long time, *nearly 2 min*.

The QUBE is an upper-bound formula; the resulting 100 s actually means that the LRT may be *up to* 100 s. The actual LRT may be less, particularly if there is a lot of memory and read cache compared to the database size. Many random touches will not then cause a read from the disk drive, and the average time per random touch may be less than 10 ms. In any case, this is *not* a false alarm. The number of random touches to the table *is much too high.* This issue is discussed at length in Chapter 15.

Comment

As the QUBE is a very rough formula, it is misleading to show more than one significant digit in the result, although we do it in this book for the sake of clarity. In a real report, the conclusion should be formulated such as: According to a quick estimate, the proposed index reduces the worst-case response time from 2 min to 1 s.

Cheapest Adequate Index or Best Possible Index

Indexing is not simply about minimizing the total number of disk I/Os; it is to try to make all programs *fast enough* without using an excessive amount of disk storage, memory and read cache, and without overloading the disk. All these issues were discussed in some detail in Chapter 4.

It is even possible to express the decision in numerical terms, such as to ask: "Is it worth paying U.S.$10 per month to decrease the response time of a transaction that is run 50,000 times a month, from 2 to 10 s to 0.1 to 0.5 s, and to save 1000 CPU seconds per month?"

Having determined, by applying the BQ and/or the QUBE (or some other method), that CURSOR55 is too slow because of inadequate indexing, we face a difficult question: Should we go for the best possible index for the cursor or choose the cheapest index improvement that results in adequate performance? Or perhaps something between the two? We don't have a rigid formula with which to answer this question, but the ideas we discussed in Chapter 4 should enable us to design a suitable index.

Best Index for the Transaction

By applying the algorithm described in Chapter 4, we can determine that there are *two* possible three-star indexes for candidate A:

```
(LNAME, CITY, FNAME, CNO)  or
(CITY, LNAME, FNAME, CNO).
```

These both have three stars because they would scan a very thin slice of the index (2 MC); the ORDER BY column follows the matching columns (both used with equal predicates) in the index, thereby negating the need for a sort; and the query would be index only.

As no sort is required, there is no need to consider candidate B.

```
Index LNAME, CITY, FNAME, CNO TR = 1    TS = 1% × 10% × 1,000,000
or CITY, LNAME, FNAME, CNO
Fetch 1000 × 0.1 ms

LRT                              TR = 1    TS = 1000
                                 1 × 10 ms 1000 × 0.01 ms
                                 10 ms + 10 ms + 100 ms = 120 ms
```

There is no difference between the two indexes with regard to performance—they both result in an enormous reduction in cost. Unfortunately, a three-star index would imply an additional index because adding a column (CITY) between LNAME and FNAME in the current index could adversely affect *existing* programs. We have already discussed this problem in Chapter 4. This is why we need to consider the cheaper alternatives first.

Semifat Index (Maximum Index Screening)

Adding the predicate column CITY at the end of the existing index (LNAME, FNAME) eliminates most of the table touches because index screening will take place. Table rows will only be accessed for index entries *known* to contain the required CITY.

This index (LNAME, FNAME, CITY) passes BQ. *All* the predicate columns are in the index. However, it has only one star; the required index rows are not close to each other (we only have one MC) and, of course, the index is not fat. But would this cheap solution be adequate?

```
Index LNAME, FNAME, CITY   TR = 1              TS = 1% × 1,000,000
Table CUST                 TR = 10% × 10,000   TS = 0
Fetch 1000 × 0.1 ms

LRT                        TR = 1001           TS = 10,000
                           1001 × 10 ms        10,000 × 0.01 ms
                           10 s + 0.1 s + 0.1 s = 10 s
```

The original LRT has been reduced *from nearly 2 minutes to 10 seconds* with this semifat index, a huge improvement, but not enough (remember the best index was only 120 ms). We still have too many expensive TRs; we will need a fat index to obtain decent performance.

Fat Index (Index Only)

Adding one more column to the semifat index makes the index fat: (LNAME, FNAME, CITY, CNO). Now we have two stars, the second and third with an LRT of 1 s.

```
Index LNAME, FNAME, CITY, CNO   TR = 1      TS = 1% × 1,000,000
Table CUST                      TR = 0      TS = 0
Fetch 1000 × 0.1 ms

LRT                             TR = 1      TS = 10,000
                                1 × 10 ms   10,000 × 0.01 ms
                                10 ms + 0.1 s + 0.1 s = 0.2 s
```

Table 5.1 provides a comparison of the performance of the various indexes; the last column shows the additional maintenance costs that would be incurred. Note that the three-star index would be a *new* index. A reminder of these costs follows:

Warning

Changing the order of the index columns in an existing index is as dangerous as adding a new column between existing index columns. In both cases an existing SELECT may become dramatically slower because the number of matching columns may be reduced or a sort (causing early result materialization) introduced.

CHEAPEST ADEQUATE INDEX OR BEST POSSIBLE INDEX: EXAMPLE 2

We will once more show how we may use the two techniques in the index design process, but this time for a slightly more difficult SELECT.

INDEX CITY—SQL 5.6A

```
DECLARE CURSOR56 CURSOR FOR
SELECT     CNO, FNAME
FROM       CUST
WHERE      CITY = :CITY
           AND
           LNAME BETWEEN :LNAME1 AND :LNAME2
ORDER BY   FNAME
```

INDEX LNAME, FNAME—SQL 5.6B

```
DECLARE CURSOR56 CURSOR FOR
SELECT     CNO, FNAME
FROM       CUST
WHERE      CITY = :CITY
           AND
           LNAME BETWEEN :LNAME1 AND :LNAME2
ORDER BY   FNAME
```

Basic Question and QUBE for the Range Transaction

For this example we will assume a maximum filter factor for CITY of 10% as before, but now that the LNAME column is in a range predicate, we will assume a larger filter factor for it than before, say 10% also. The maximum size of the result table will now be assumed to be 10% of 10% of 1,000,000 which is 10,000.

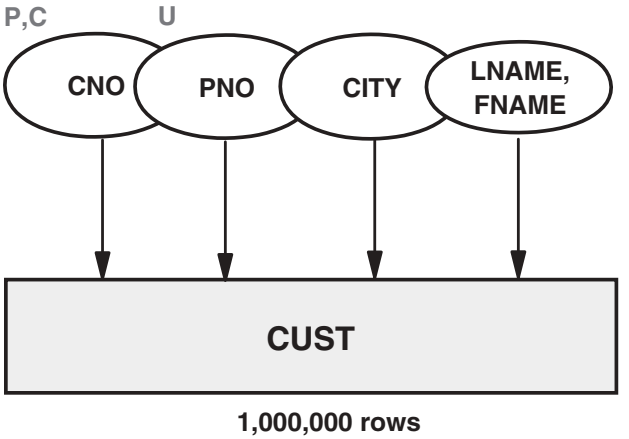


Figure 5.9 Cheapest adequate index or best possible index: Example 2.

Figure 5.9 shows the current indexes for the customer table. There are now two possible indexes that should be considered, (CITY) and (LNAME, FNAME) and so CURSOR56 is shown twice (SQL 5.6A and B), once for each index; note that neither cursor is able to avoid a sort.

Neither index is semifat and so both will fail BQ. The QUBE shows very poor performance; the result is identical regardless of which index is used. Each can only use 1 MC in a matching index scan, each requires a sort, and neither are index only. No stars at all, with correspondingly dreadful performance.

```
Index LNAME, FNAME      TR = 1              TS = 10% × 1,000,000
or CITY
Table CUST              TR = 100,000        TS = 0
Fetch 10,000 × 0.1 ms

LRT                     TR = 100,001        TS = 100,000
                        100,001 × 10 ms     100,000 × 0.01 ms
                        1000 s + 1 s + 1 s = 17 min
```

With these indexes, many optimizers would actually choose multiple index access (as discussed in Chapter 10); two index slices would be read, (LNAME, FNAME) and (CITY) before accessing the table. This would take much less than 17 min, but it would still take far too long.

Best Index for the Transaction

The *best index* is easy to derive. Candidate A is:

```
(CITY, LNAME, FNAME, CNO)
```

This index only has *two* stars because a sort is required (the ORDER BY column follows a range predicate column). Therefore candidate B must be:

```
(CITY, FNAME, LNAME, CNO)
```

This also has two stars, but it has the second star instead of the first. The QUBE will easily determine which one is the best.

```
Index CITY, LNAME, FNAME, CNO   TR = 1      TS = 10% × 10% × 1,000,000
Fetch 10,000 × 0.1 ms

LRT for candidate A             TR = 1      TS = 10,000
                                1 × 10 ms   10,000 × 0.01 ms
                                10 ms + 0.1 s + 1 s = 1 s

Index CITY, FNAME, LNAME, CNO   TR = 1      TS = 10% × 1,000,000
Fetch 10,000 × 0.1 ms

LRT for candidate B             TR = 1      TS = 100,000
                                1 × 10 ms   100,000 × 0.01 ms
                                10 ms + 1 s + 1 s = 2 s
```

The best index is clearly candidate A (CITY, LNAME, FNAME, CNO) with an LRT of 1 s; even the *best* index is now a borderline case.

We suggested earlier that for the purpose of *comparing* alternative access paths, the FETCH processing may be ignored because it will be the same in all cases. In doing this, however, one has to be a little careful when drawing conclusions about the need for changes to the indexes. For example, in this case, ignoring the FETCH processing would suggest that candidate A (LRT 0.1 s) would be *10 times* cheaper than candidate B (LRT 1 s), whereas the true advantage is proportionally far smaller (2 to 1).

The cost of the sort required by candidate A is trivial *in this example* compared to the 1-s saving over candidate B; in fact the sort cost has really been absorbed in the FETCH cost as described earlier. The problem with candidate B is the large number for TS because we are using a thick index slice (10%).

Semifat Index (Maximum Index Screening)

Adding the missing predicate column at the end of the two existing indexes would eliminate many of the huge number of TRs to the table because index screening would take place. Table rows will only be accessed for index entries that are known to contain *both* the required CITY and the LNAME columns.

The two semifat indexes we could use with the existing indexes would be:

```
(CITY, LNAME)      or      (LNAME, FNAME, CITY)
```

The QUBE will again determine which one is the best.

```
Index CITY, LNAME          TR = 1              TS = 10% × 10% × 1,000,000
Table CUST                 TR = 10,000         TS = 0
Fetch 10,000 × 0.1 ms

LRT                        TR = 10,001         TS = 10,000
                           10,001 × 10 ms      10,000 × 0.01 ms
                           100 s + 0.1 s + 1 s = 101 s

Index LNAME, FNAME, CITY   TR = 1              TS = 10% × 1,000,000
Table CUST                 TR = 10% × 100,000  TS = 0
Fetch 10,000 × 0.1 ms

LRT                        TR = 10,001         TS = 100,000
                           10,001 × 10 ms      100,000 × 0.01 ms
                           100 s + 1 s + 1 s = 102 s
```

Although we have a thicker slice with the second index, this factor is heavily outweighed by the huge (albeit much reduced because of the screening effect) number of table accesses. It looks as though we will need a fat index though to eliminate the 10,000 TRs to the table.

Fat Index (Index Only)

Adding two more columns to the first semifat index evaluated above and one more to the second will make *both* indexes fat: (CITY, LNAME, FNAME, CNO) or (LNAME, FNAME, CITY, CNO).

```
Index CITY, LNAME, FNAME, CNO   TR = 1      TS = 10% × 10% × 1,000,000
Fetch 10,000 × 0.1 ms

LRT                             TR = 1      TS = 10,000
                                1 × 10 ms   10,000 × 0.01 ms
                                10 ms + 0.1 s + 1 s = 1 s

Index LNAME, FNAME, CITY, CNO   TR = 1      TS = 10% × 1,000,000
Fetch 10,000 × 0.1 ms

LRT                             TR = 1      TS = 100,000
                                1 × 10 ms   100,000 × 0.01 ms
                                10 ms + 1 s + 1 s = 2 s
```

Now that the table accesses have been eliminated, the difference between the LRTs of the two indexes is apparent. The first index has the first star, providing a thin index slice; the second does not and so uses a thicker index slice.

The first index is, of course, the *best* index we derived earlier for candidate A. This is because the original index on which it is based only contained CITY, the only equal predicate in the SELECT. So we were able to *build it up* into the best index, first by adding the BETWEEN predicate column (making it semifat, which was still not adequate), and then by adding the other columns from the SELECT (making it fat).

Table 5.2 provides a comparison of the performance of the various indexes.

Table 5.2 Comparison of Worst Input QUBEs for Example 2

Type	Index	LRT	Maintenance
Existing	LNAME, FNAME *or* CITY	17 m	—
Semifat	CITY, LNAME	101 s	U LNAME +10–20 ms
Semifat	LNAME, FNAME, CITY	102 s	U CITY +10–20 ms
Fat	CITY, LNAME, FNAME, CNO	1 s	U L & FNAME +10–20 ms
Fat	LNAME, FNAME, CITY, CNO	2 s	U CITY +10–20 ms
2 * A	CITY, LNAME, FNAME, CNO	1 s	U L & FNAME +10–20 ms
2 * B	CITY, FNAME, LNAME, CNO	2 s	U L & FNAME +10–20 ms

LRT is for the worst input QUBE; I = insert; D = delete; U = update

WHEN TO USE THE QUBE

Ideally, the QUBE should be applied during the design of a new program. If the worst input local response time with the current or planned indexes exceeds, say 2 s, index improvements should be considered.

The alarm limit for batch jobs is job specific, but it is very important to check the maximum time between commit points in all batch programs. This should probably be less than 1 or 2 s, otherwise the batch job may cause excessive lock waits in transactions and other batch jobs.

The QUBE could even be applied *before* designing the program. It is simply necessary to know how the database must be processed with the worst input; for example, read 10 nonadjacent rows from table A, add 2 adjacent rows to table B, and so on. In fact, it is wise not to design the program until the estimate is satisfactory. Sometimes it may be necessary to make the program structure more complex in order to achieve adequate performance.

Chapter 7 will discuss the use of the QUBE when problems arise *after* the program has gone into production.

Chapter 6

Factors Affecting the Index Design Process

- Important factors that affect the index design process considered in Chapters 4 and 5
- Verification of the basic estimates
- Multiple thin slices
- DBMS specifics
- Difficult predicates for optimizers
- Boolean predicates
- Filter factor problems and pitfalls

I/O TIME ESTIMATE VERIFICATION

In Chapter 4 we introduced the following I/O times:

Random Read	10 ms (4K or 8K page)
Sequential Read	40 MB/s

These numbers assume current hardware with a reasonable load. Some installations may be slower or overloaded, and so it might be useful to perform the following checks. To do this, a table can be created with a million rows having an average length of about 400 bytes. The assumed sequential I/O time per 400-byte row is then 400 bytes/40 MB/s = 0.01 ms.

The time taken to perform the following scans is then determined (the figures in parentheses are the predictions based on our estimates as used in Chapter 4):

- A full table scan with a singleton SELECT that rejects all rows (1×10 ms $+ 1{,}000{,}000 \times 0.01$ ms $= 10$ s)
- An index scan with 1000 FETCH calls, using an index such as (LNAME, FNAME), that causes an index slice scan of 1000 rows and 1000 synchronous reads to the table (1000×0.01 ms $+ 1000 \times 10$ ms $= 10$ ms $+ 10$ s $= 10$ s)

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

It is essential to set up the measurement in such a way that the required pages are not in memory or read cache at the start of the program. An interval of a couple of hours between measurements is probably sufficient if the system is fairly busy during the interval.

If the *measured* elapsed times are *much* longer than the *predicted* times, the primary causes should be determined:

Slow disk drives or paths?

Overloaded disk drives?

Regardless of whether these figures are optimistic or even pessimistic, they can be useful as *relative* performance indicators.

MULTIPLE THIN INDEX SLICES

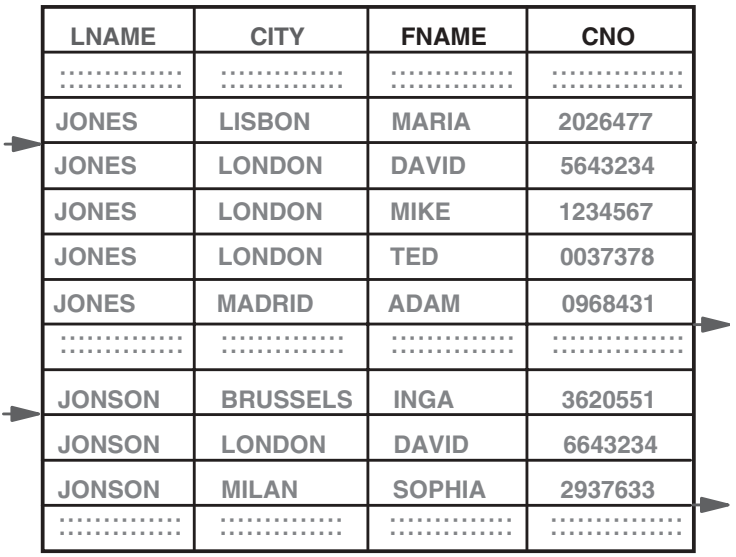
Normally, an index is derived from a SELECT (and not the other way round), but sometimes it may be desirable for an SQL statement to be rewritten to enable the DBMS to use an existing index in the most efficient manner. This is because the optimizers do not know everything!

SQL 6.1

```
DECLARE CURSOR61 CURSOR FOR
SELECT     CNO, FNAME
FROM       CUST
WHERE      LNAME BETWEEN :LNAME1 AND :LNAME2
           AND
           CITY BETWEEN :CITY1 AND :CITY2
ORDER BY   FNAME
```

A *perfect optimizer* would be able to construct an access path consisting of *several* thin slices when the WHERE clause contains *two* range predicates, as in CURSOR61 (refer to SQL 6.1 and Figure 6.1). In this example we assume that the input consists of abbreviated values for both LNAME and CITY, say JO and LO, from which the program derives the range values.

Because a range predicate column is the *last* matching column in the matching process, making the DBMS read *several* thin slices (defined by *both* LNAME and CITY) requires additional program logic. The predicate LNAME BETWEEN :LNAME1 AND :LNAME2 must be changed to LNAME = :LNAME. This can be done with an auxiliary table, maintained with a trigger, that contains all the distinct values of the column LNAME in the customer table. Then when the user enters the character string JO, the program reads one qualifying LNAME at a time from the auxiliary table and opens CURSOR62 (SQL 6.2) with each LNAME value.



LNAME	CITY	FNAME	CNO
.....	.....	.....	.....
JONES	LISBON	MARIA	2026477
JONES	LONDON	DAVID	5643234
JONES	LONDON	MIKE	1234567
JONES	LONDON	TED	0037378
JONES	MADRID	ADAM	0968431
.....	.....	.....	.....
JONSON	BRUSSELS	INGA	3620551
JONSON	LONDON	DAVID	6643234
JONSON	MILAN	SOPHIA	2937633
.....	.....	.....	.....

Figure 6.1 Multiple thin index slices (CURSOR62) if MC = 2.

SQL 6.2

```
DECLARE CURSOR62 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME =:LNAME
            AND
            CITY BETWEEN :CITY1 AND :CITY2
ORDER BY    FNAME
```

Now even a DBMS *without* a perfect optimizer will read several thin slices from the index (LNAME, CITY, FNAME, CNO). EXPLAIN will report MC = 2 for CURSOR62. The response time will depend on the number of slices and the number of rows in each slice.

Let's compare the two alternatives using the following assumptions:

- One million rows in the customer table
- Worst-case filter factor for LNAME = 1%
- Worst-case filter factor for CITY = 10%

The index slice scanned by CURSOR61 will consist of 10,000 rows (MC = 1; 1% of 1,000,000 = 10,000); only 10% of these will be for the required CITY (1000 rows).

Let us also assume there are 50 distinct LNAME values in that slice. The tuned program using CURSOR62 would read these 50 slices (with an average of 20 rows in each); each of the 1000 rows will be for the required CITY.

According to the QUBE (ignoring the FETCHes as we are simply making a comparison), the elapsed times for the two cursors would be:

CURSOR 61 $1 \times 10 \text{ ms} + 10{,}000 \times 0.01 \text{ ms} = 0.1 \text{ s}$
CURSOR 62 $50 \times (1 \times 10 \text{ ms} + 20 \times 0.01 \text{ ms}) = 0.5 \text{ s}$

This is somewhat surprising but logical; sequential processing is sometimes more efficient than skip-sequential processing because each skip may imply a random touch. With different numbers, the tuned program with CURSOR62 could be *much* faster than the simple program with CURSOR61.

To prove this last point, let's consider a case with a *bigger table* (100,000,000 rows) and a *higher filter* factor for the first index column as shown in SQL 6.3.

SQL 6.3

```
SELECT     CNO, FNAME
FROM       CUST
WHERE      LNAME BETWEEN :LNAME1 AND :LNAME2   FF max 10%
           AND
           CITY BETWEEN :CITY1 AND :CITY2      FF max 0.01%
ORDER BY   FNAME
```

The size of the result table will be $0.01\% \times 10\% \times 100{,}000{,}000 = 1000$ rows. Let us assume an index (LNAME, CITY, FNAME, CNO) and 50 distinct values of LNAME in the 1000 row result table. Now, the simple program (two BETWEENs) elapsed time estimate (one thick slice) is

$$1 \times 10 \text{ ms} + 10{,}000{,}000 \times 0.01 \text{ ms} = 100 \text{ s}$$

while the estimate for the tricky program (50 thin slices) is *still*

$$50 \times (1 \times 10 \text{ ms} + 20 \times 0.01 \text{ ms}) = 0.5 \text{ s}$$

Oracle 9i is able to generate a skip-sequential access path instead of a full index scan—*index skip scan*. In our example, this would mean scanning one slice for each LNAME value. These slices have two matching columns, LNAME and CITY.

Simple Is Beautiful (and Safe)

The solution using the auxiliary table for distinct LNAME values is a typical example of a compromise between data integrity and performance. In theory, there is no integrity risk if the auxiliary table is maintained with a trigger. However, triggers are application programs and programs are not perfect. Is

it absolutely certain that the trigger does not remove JONES from the auxiliary table when ALAN JONES is deleted from the customer table but another JONES remains? One should resort to tuning tricks like this *only* if a quantifiable estimate proves that performance would be unacceptable otherwise; and sometimes the trick may turn out to have a negative effect on performance as we saw earlier.

DIFFICULT PREDICATES

Definition: A predicate is too difficult for an optimizer if it cannot participate in defining the index slice—that is, it cannot be a matching predicate.

Consumer-friendly vendors of DBMSs provide a list of predicates that are too difficult for their optimizer in this respect. Sometimes such predicates are called *nonindexable*. Typical examples include column functions such as

```
COL1 CONCAT COL2 > :hv
```

and negations such as

```
COL1 NOT BETWEEN :hv1 AND :hv2
```

The DBMS-specific pitfall lists of difficult predicates tend to become shorter with each new version released; however, sometimes a pitfall may have been removed but perhaps not entirely. For instance, a function or arithmetic expression may have become easy enough for the optimizer, but perhaps not all variations. It is always advisable to check with EXPLAIN whether a suspicious predicate is too difficult for the optimizer being used.

The predicates that are discussed now are particularly important.

LIKE Predicate

In CURSOR61 and CURSOR62, LIKE might have served the same purpose as BETWEEN depending on the values used in the host variables. LIKE would have actually been more convenient, but when used with host variables it is too difficult for many optimizers; the correct result is obtained, of course, but the access path may be slow if the DBMS scans the whole index because the LIKE predicate doesn't participate in the matching process.

These issues are DBMS dependent. DB2 for z/OS, for instance, creates two access paths for COL LIKE :COL and selects one of them at execution time when it knows the value that has been moved into the host variable :COL. If the value is %XX, it will choose a full index scan; if the value is XX%, it will choose an index slice scan—unless there is a field procedure for the column LNAME. Field procedures affecting sort order are commonly used in countries suffering from "national characters."

The predicate COL LIKE :COL should only be used if the optimizer is known to cause no problems.

OR Operator and Boolean Predicates

Even simple predicates may be difficult for an optimizer if they are ORed with other predicates; the predicate may be able to participate in defining an index slice only with *multiple index access*, which will be discussed in Chapter 10.

The following examples (SQL 6.4A and B) should make this clear. Let us assume there is an index (A, B) on TABLE.

SQL 6.4A

```
SELECT   A, B, C
FROM     TABLE
WHERE    A > :A
         AND
         B > :B
```

SQL 6.4B

```
SELECT   A, B, C
FROM     TABLE
WHERE    A > :A
         OR
         B > :B
```

In SQL 6.4A, a single slice of the index would be scanned, defined by the predicate A > :A, the only matching column as it is a range predicate. Column B would be checked during this single scan due to index screening, the index entry being ignored if the predicate B > :B wasn't satisfied.

In SQL 6.4B, because of the OR operator, the DBMS cannot read *this slice alone* because even if the predicate A > :A isn't satisfied, the predicate B > :B may well be. The only feasible alternatives would be a full index scan, a full table scan, and, *if there is another index starting with column B*, multiple index access.

In this case we cannot blame the optimizer. Sometimes a cursor with a complex WHERE clause may have to be split into several cursors with simple WHERE clauses in order to avoid problems like this. In general, this type of predicate is *too difficult for the optimizer if a row cannot be rejected when that predicate is false without checking other predicates*. In SQL 6.4B, a row cannot be rejected just because predicate A > :A is false, nor because predicate B > :B is false. In SQL 6.4A a row *can be* rejected if predicate A > :A is false without needing to check predicate B > :B.

Such a predicate is called a *non-Boolean term* or *non-BT*. If there are only AND operators in a WHERE clause as in SQL 6.4A, all simple predicates are BT (Boolean term) and hence "good" predicates because a row can be rejected if any one of the simple predicates is evaluated false.

Summary

Only BT predicates can participate in defining an index slice, unless the access path is a multiple index scan.

Let us consider a WHERE clause that defines large men (SQL 6.5):

SQL 6.5

```
WHERE   SEX = 'M'
        AND
        (WEIGHT > 90
        OR
        HEIGHT > 190)
```

Only the first simple predicate SEX = 'M' is BT. If the WEIGHT predicate is false, the HEIGHT predicate may be true and vice versa. Neither can, by itself, cause the row to be rejected.

This is why we made the following statement: *Include only predicates that are simple enough for the optimizer* in Chapter 4 when using the ideal index algorithm for candidates A and B. For SQL 6.5, only column SEX could be used in step 1 of the algorithm.

IN Predicate

The predicate COL IN (:hv1, :hv2 . . .) may cause problems for optimizers. In DB2 for z/OS V7 for instance, *only* ***one*** *IN predicate can be matching* and so participate in defining the index slice, but the column in the matching IN predicate will not necessarily be the *last* matching column. Therefore, when the ideal index Candidate A is derived, the column of the *most selective* IN predicate should be included in step 1. SQL 6.6 provides an example.

SQL 6.6

```
DECLARE CURSOR66 CURSOR FOR
SELECT   A, B, C, D, E
FROM     TX
WHERE    A = :A
         AND
         B IN (:B1, :B2, :B3)   FF = 0...10%
         AND
         C > 1
         AND
         E IN (:E1, :E2, :E3)   FF = 0...1%
```

The numbers in the candidate A selection below refer to the steps described in Chapter 4.

Candidate A

1. Pick columns A and E.
2. Add column C.
3. No columns.
4. Add columns B and D.

Candidate A is (A, E, C, B, D) or any of the following variations:

- (A, E, C, D, B)
- (E, A, C, B, D)
- (E, A, C, D, B).

All these index candidates have MC = 3 with CURSOR66. The DBMS reads three index slices (A, :E1, C), (A, :E2, C), and (A, :E3, C) without any special tricks required. The B IN predicate would participate in screening. If four matching columns were required (nine index slices), it would probably be necessary to split the SELECT into three, each with only one IN list predicate. The first SELECT would include predicate B = :B1, the second B = :B2, and the third B = :B3. These three SELECTs could be combined into one cursor with UNION ALL. The EXPLAIN for this cursor should show MC = 4 three times.

Candidate B

IN predicate columns should *not* be chosen in step 1, otherwise the sequence requirement would be destroyed and a sort would become necessary.

FILTER FACTOR PITFALL

Throughout this book we have focused on the *worst-case scenarios* that arise when the input corresponds to the *maximum* filter factors. This isn't always the case, however, as this example (SQL 6.7) will show.

SQL 6.7

```
SELECT      B
FROM        TABLE
WHERE       A = :A (FF = 1%)
            AND
            C > :C (FF = 0...10%)
ORDER BY    B
WE WANT 20 ROWS PLEASE
```

In this SELECT statement, we aren't interested in the *whole result set* as we may deduce from the WE WANT n ROWS PLEASE clause. When 20 rows have been FETCHed, the screen is presented to the user regardless of the total size of the result set.

To explain the performance implications of this with regard to the filter factors involved, we will compare QUBEs with two filter factor extremes; the first provides the smallest result set possible, 0 rows; the second a fairly large result set (1000 rows). Figure 6.2 shows both cases, using index (A, B, C). Note that each case has the same filter factor for the matching index column A. The difference in the size of the result set is entirely determined by column C.

Let's first consider the *second* case, the large result set, which *normally* gives rise to the worst performance. A matching index scan (1 MC, index only) will access an index slice of 20 × 10 rows because to provide *each row* in the result set, 10 index rows need to be examined (column C FF 10%).

```
Index A, B, C          TR = 1      TS = 20 × 10
Fetch 20 × 0.1 ms

LRT Large Result Set   TR = 1      TS = 200
                       1 × 10 ms   200 × 0.01 ms
                       10 ms + 2 ms + 2 ms = 14 ms
```

In the *first* case, *no rows* satisfy the column C predicate and so the filter factor is 0%. Of course, the DBMS *doesn't know this* until *every single row* in the index slice, 1% of 1,000,000, is examined.

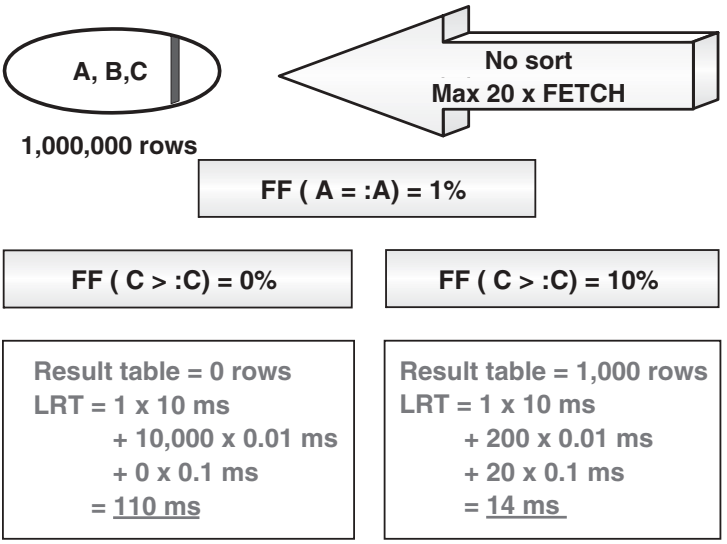


Figure 6.2 Filter factor pitfall.

```
Index A, B, C          TR = 1      TS = 1% × 1,000,000
Fetch 0 × 0.1 ms

LRT Empty Result Set   TR = 1      TS = 10,000
                       1 × 10 ms   10,000 × 0.01 ms
                       10 ms + 100 ms + 0 ms = 110 ms
```

Clearly, the *worst case* for this transaction is the *empty result set or any result set that implies reading the whole index slice*. Even worse, consider the implication of this case *not being index only* and the index *not being a clustering index*; an additional 10,000 TRs would be required, taking *nearly 2 min* to discover that not a single row could be found!

Consider also the implication of a *thicker* index slice being used in the matching scan process, a 20% filter factor for column A, for example, giving an LRT of 2 s (200,000 × 0.01 ms), even with an index only scan! When the result fits on one screen, the first star (minimize the thickness of the index slice to be scanned) is more important than the second star (avoid a sort).

This *filter factor pitfall* may arise when the following *three conditions* are all true:

1. There is *no sort* in the access path.
2. The transaction responds as soon as the *first screen is built*.
3. *All* the predicate columns *do not* participate in defining the index slice to be scanned—in other words, they are *not all* matching columns.

All three conditions apply to this example:

- There is no sort because column B (the Order By column) follows the equal predicate column A, which is the first column in the index.
- The transaction responds as soon as the first screen is built (20 rows).
- One predicate column (column C) does not participate in defining the index slice to be scanned.

The reasoning behind these three conditions should be fully understood:

- If conditions 1 and 2 are *not* true, the whole result table is always materialized either by the DBMS or by the application program (by issuing FETCHes).
- If condition 3 is *not* true, there is no index *screening* (because all the predicate columns are *matching*); every index row in the index slice qualifies. Even if the result table is empty, the DBMS makes only one touch to determine that this is so.

FILTER FACTOR PITFALL EXAMPLE

In this section we will once more show how we may use the two estimation techniques in the index design approach discussed in Chapters 4 and 5.

SQL 6.8

```
DECLARE CURSOR68 CURSOR FOR
SELECT     CNO, FNAME
FROM       CUST
WHERE      LNAME = :LNAME
           AND
           CITY = :CITY
ORDER BY   FNAME
WE WANT 20 ROWS PLEASE

OPEN CURSOR CURSOR610
FETCH CURSOR CURSOR610 max 20
CLOSE CURSOR CURSOR610
```

In this example, the user chooses both the last name (LNAME) and the city (CITY) from pop-up windows before starting the transaction, which then opens CURSOR68 (refer to SQL 6.8). When we analyzed this transaction earlier, the whole of the result set was required. In this transaction, however, *no more than 20 FETCH calls are issued by the program*, perhaps just enough to fill one screen. The next transaction displays the next 20 result rows and so forth. This technique complicates programming but improves the response time for the first screen when the result table is very large. As before, the maximum filter factors are assumed to be 1% (LNAME) and 10% (CITY).

Figure 6.3 shows the current indexes for the customer table. All of them fail the BQ test:

> Is there an existing or planned index that contains all the columns referenced by the WHERE clause (a semifat index)?

How serious would the problem be *now* if we decided to use index (LNAME, FNAME) anyway. The QUBE for the worst case will answer this question.

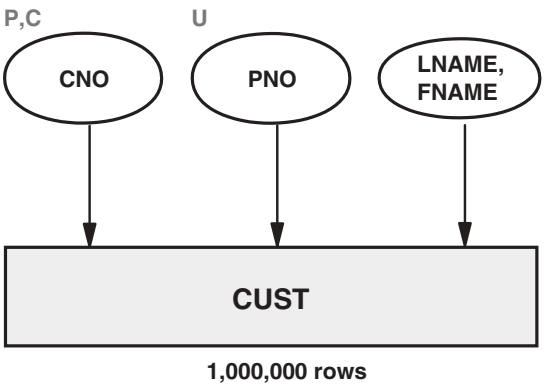


Figure 6.3 Filter factor pitfall example—indexes.

The result rows are obtained in the requested order (ORDER BY FNAME), when the DBMS chooses index (LNAME, FNAME); no sort is needed. Therefore, *the DBMS will not do early materialization*: OPEN CURSOR generates no touches, but each FETCH generates touches to both the index and the table, some of which will return a result row to the FETCH and some, the majority, that won't.

In Figure 6.4 the first two touches (one index, one table) produce a result row because the first Jones lives in London. The first FETCH is now satisfied, and the program issues a second FETCH. The next Jones does not live in London, and so does not provide a result row, so more touches to both the index and table take place until another London Jones is found to satisfy the FETCH. Because only 10% of the customers live in London, the average FETCH will need to check 10 Jones entries—10 index touches and 10 table touches—before it adds a row to the result table.

Thus, it takes on average $20 \times (10 + 10) = 400$ touches to build a full screen with 20 customers.

```
Index LNAME, FNAME      TR = 1          TS = 20 × 10
Table CUST              TR = 20 × 10    TS = 0
Fetch 20 × 0.1 ms

LRT Large Result Set    TR = 201        TS = 200
                        201 × 10 ms     200 × 0.01 ms
                        2 s + 2 ms + 2 ms = 2 s
```

The local response time according to the QUBE is perhaps almost acceptable, 2 s. But is this really *the worst input response time?* The assumed filter factors, 10% for CITY =: CITY and 1% for LNAME =: LNAME, were the *maximum values*. However, in this case the filter factors producing the largest result table

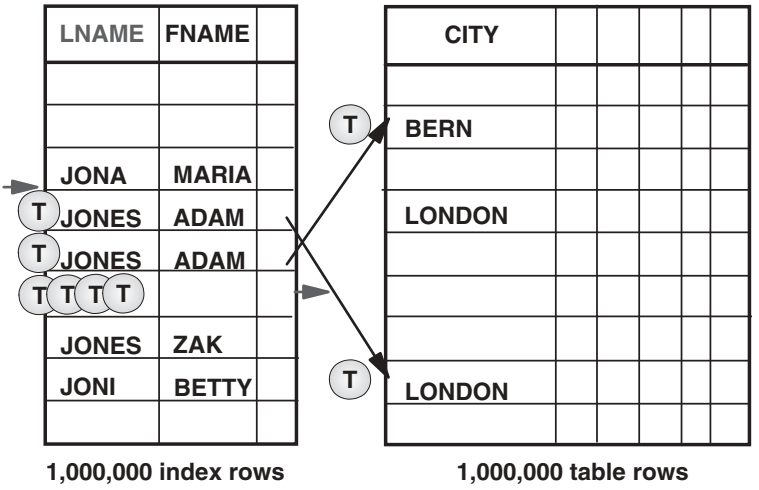


Figure 6.4 Filter factor pitfall example—index entries.

are not the worst case because this is another example of the filter factor pitfall. The three conditions for the filter factor pitfall are true for this Cursor with index (LNAME, FNAME):

- There is no sort in the access path.
- The transaction responds as soon as the first screen is built.
- All the predicate columns do not participate in defining the index slice to be scanned—in other words, they are *not* all matching columns.

Therefore, it may take longer to create the message "No qualifying customers" than the first screen of a large result set.

Let's consider the extreme case where the DBMS has to scan a maximum index slice but produces an empty result set—a city in which we have no customers with the most common last name; now the DBMS must still scan the whole index slice and check a table row for each index row.

```
Index LNAME, FNAME     TR = 1           TS = 1% × 1,000,000
Table CUST             TR = 10,000      TS = 0
Fetch 0 × 0.1 ms

LRT Empty Result Set   TR = 10,001      TS = 10,000
                       10,001 × 10 ms   10,000 × 0.01 ms
                       100 s + 100 ms + 0 ms = 100 s
```

Nearly *2 min instead of 2 s*! Indexing must be improved, and now we face the difficult question once more: the cheapest adequate improvement or the best index, or something in between?

Best Index for the Transaction

The ideal index is easy to derive. Candidate A is

```
(LNAME, CITY, FNAME, CNO)
```

Because this index has three stars, there is no need to derive candidate B. The QUBE with this index is

```
Index LNAME, CITY, FNAME, CNO   TR = 1      TS = 20
Fetch 20 × 0.1 ms

LRT                             TR = 1      TS = 20
                                1 × 10 ms   20 × 0.01 ms
                                10 ms + 0.2 ms + 2 ms = 12 ms
```

This relates to the *worst* case with candidate A: at least 20 result rows. With the ideal index, it takes 21 index touches to build a 20-row result table when the whole result does not fit on one screen. The difference between the worst input response times of indexes (LNAME, FNAME) and (LNAME, CITY, FNAME, CNO) is about *four orders of magnitude*.

Note that with this three-star index, if the result table was empty, this would be determined by *a single touch* to the index. The three-star index, by definition, fails the third requirement of the pitfall list.

Unfortunately, a three-star index would imply an additional index because adding a column (CITY) between LNAME and FNAME in the current index could adversely affect existing programs. We have already discussed this problem in Chapters 4 and 5. This is why we need to consider the cheaper alternatives first.

Semifat Index (Maximum Index Screening)

Adding the predicate column CITY *at the end* of the existing index (LNAME, FNAME) would eliminate most of the table touches because the optimizer should be able to do index screening in this simple case and read the table row only when CITY has the required value.

Index (LNAME, FNAME, CITY) passes BQ: All predicate columns are in the index. However, it only gets one star. The appropriate index rows are not close to each other (MC is only 1) and the index is not fat. So would this cheap solution bring acceptable response times?

Now, the cursor with this index does fall into the filter factor pitfall category. Consequently, we have to estimate the *extreme* filter factors with the index (LNAME, FNAME, CITY). One of these will be the worst case for this access path.

Largest Result Table (say 1000 rows)

```
FF (LNAME = :LNAME) = 1%
FF (CITY = :CITY) = 10%
FF (LNAME = :LNAME AND CITY = :CITY) = 0.1%
```

Every tenth index row satisfies the predicate CITY =: CITY with CITY FF = 10%, and so it takes, on average, 10 index touches to find one result row.

```
Index LNAME, FNAME, CITY  TR = 1      TS = 20 × 10
Table                     TR = 20     TS = 0
Fetch 20 × 0.1 ms

LRT 1000 result rows      TR = 21     TS = 200
                          21 × 10 ms  200 × 0.01 ms
                          210 ms + 2 ms + 2 ms = 214 ms
```

Empty Result Table

```
FF (LNAME = :LNAME) = 1%
FF (CITY = :CITY) = 0%
FF (LNAME = :LNAME AND CITY = :CITY) = 0%
```

With the empty result set *every* index row found in the 1% predicate LNAME =: LNAME index slice has to be examined, but the result of the CITY screening is clearly reflected in the table touch figures.

```
Index LNAME, FNAME, CITY  TR = 1     TS = 10,000
Table                     TR = 0     TS = 0
Fetch 0 × 0.1 ms

LRT empty result set      TR = 1     TS = 10,000
                          1 × 10 ms  10,000 × 0.01 ms
                          10 ms + 100 ms + 0 ms = 110 ms
```

Fat Index (Index Only)

Adding one more column to the semifat index evaluated above makes the index fat: (LNAME, FNAME, CITY, CNO).

```
Index LNAME, FNAME, CITY, CNO   TR = 1      TS = 20 × 10
Fetch 20 × 0.1 ms

LRT 1000 result rows            TR = 1      TS = 200
                                1 × 10 ms   200 × 0.01 ms
                                10 ms + 2 ms + 2 ms = 14 ms

Index LNAME, FNAME, CITY, CNO   TR = 1      TS = 10,000
Fetch 0 × 0.1 ms

LRT empty result set            TR = 1      TS = 10,000
                                1 × 10 ms   10,000 × 0.01 ms
                                10 ms + 100 ms + 0 ms = 110 ms
```

The local response time with the empty result set remains 0.1 s. Transactions that produce one screen of a multiscreen result are now very fast because the 20 TRs to the table are no longer required.

Summary

1. The worst case depends on the index used:
   - For the original index it is the empty result table (most common LNAME and uncommon CITY).
   - For the semifat index, although in our example the large result table is the worst case, it could be either result as determined by the relative cost of the table access and the index slice scan. 20 TRs (200 ms) is (according to the QUBE) equivalent in elapsed time to 20,000 TSs (200 ms). With fewer rows per screen or a thicker index slice, the empty result table could become the worst case.
   - For the fat index it is the empty result table.
   - For the three-star index it is the large result table (a single touch is required for the empty result table).

Table 6.1 Comparison of the Worst Input QUBE

Type	Index	LRT	Maintenance
Existing	LNAME, FNAME	100 s	—
Semifat	LNAME, FNAME, CITY	0.2 s	U CITY + 10–20 ms
Fat	LNAME, FNAME, CITY, CNO	0.1 s	U CITY + 10–20 ms
3*	LNAME, CITY, FNAME, CNO	0.01 s	I/D + 10 ms U + 10–20 ms

LRT is for the worst input QUBE; I = insert; D = delete; U = update

2. The original index would be a disaster, not with a large result table but with an empty one. This is due to the huge number of table accesses carried out to check the CITY.
3. Both the semifat index and the fat index provide adequate performance, even with the empty result table. Both indexes avoid the unnecessary table accesses, and the cost of scanning the index slice is relatively small. The local response time also grows relatively slowly as the slice defined by column LNAME grows. If the company obtains a lot of customers from South Korea, the LEE slice may one day consist of 100,000 index rows. Both indexes would still provide adequate response times with this large result set; the worst input response would now be 1 s (100,000 × 0.01 ms).
4. The advantage of using the fat index in preference to the semifat index would be to avoid accessing the table for the first 20 result rows.
5. Although there would be a further small advantage to be gained in using the three-star index, particularly for the empty result table case, this would not be significant and it would incur the overheads associated with a new index (but note point 6 below).
6. If the LEE slice becomes too thick, a three-star index may be justified; there will be a limit, perhaps of the order of 100,000 TSs, when scanning an index slice just takes too long and a new index, a three star in this case, should be created.

A summary of the various indexes we have considered during this analysis is shown in Table 6.1.

Before we leave this example, we should explain that we have been tacitly assuming that the empty result set may indeed be the worst case. We should appreciate, however, that an *almost* empty result would have been even worse, causing up to 20 table accesses in the case of a semifat index.

EXERCISES

6.1. The SELECT in Figure 6.5 takes up to 1 min with the current indexes. Design the best possible indexes: (1) without adding a third index and (2) with a third index.

```
SELECT  C, E, F
FROM    TABLE
WHERE  B = :B
         AND
       C > :C
         AND
       D = 1
ORDER BY C
```

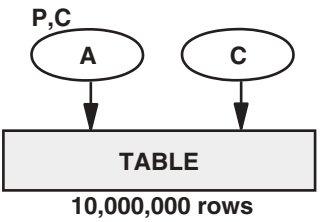


Figure 6.5 Exercises 6.1 and 6.2.

6.2. Assume that one second would be an acceptable elapsed time and that we are reluctant to add a new index. What would you need to know to predict the improvement provided by alternative 1?

Chapter 7

Reactive Index Design

- Role of the EXPLAIN function and how it may be used during the index design process to describe the selected access paths
- How to use LRT-level exception monitoring to identify transaction SQL problems by the use of spike reports that identify individual problems
- Use of bubble charts to identify and analyze culprits and victims
- Distinguishing between promising and unpromising culprits to focus on the major potential benefits to be obtained—the tuning potential
- How to use Call-level exception monitoring and a comparison with LRT-level
- Finding slow SQL calls
- DBMS-specific monitoring issues

INTRODUCTION

We have so far focused on relatively straightforward SQL statements. Before we address the more complex activities such as Joins, Subqueries, Unions, Star Joins, and Multiple Index Access, it will be useful to discuss how our SQL can be monitored and to consider the index design considerations from a reactive point of view.

Amy Anderson and Michael Cain (6) describe the *reactive query approach* vividly (page 26):

> *The reactive approach is very similar to the Wright Brothers' initial airplane flight experiences. Basically the query is put together, pushed off a cliff, and watched to see if it flies. In other words, build a prototype of the proposed application without any indexes and start running some queries. Or, build an initial set of indexes and start running the application to see what is used and what not. Even with a smaller database, the slow running queries will become obvious very quickly.*
>
> *The reactive tuning method is also used when trying to understand and tune an existing application that is not performing up to expectations.*

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

Postponing most of the index tuning for an application until after production cutover is indeed like pushing a large jet full of passengers off a cliff. It may have a happy ending, but the database crew must be prepared to take quick action. If nobody has paid much attention to indexing during the application development, some programs are likely to be painfully slow after production cutover.

Of course, we do not recommend the Wright brothers' approach. We believe the indexes for an application should be well tuned *before* cutover. However, even if quick estimates and a quick EXPLAIN review *are* done for all programs, some performance surprises will undoubtedly arise when production starts. Estimates are never perfect; it is easy to miss something when one estimates the worst input filter factors, for instance.

In this chapter we will discuss the performance tools and techniques that are essential during the first days of the life of an application in production. These same tools and techniques are also useful later for more relaxed tuning, to find performance problems before the users are aware of them, or at least before they become too painful; in this case we are actually being proactive rather than reactive.

EXPLAIN DESCRIBES THE SELECTED ACCESS PATHS

It is quite easy to identify suspicious access paths, particularly if the EXPLAIN output is stored in a table, thereby enabling easy accessibility. This is why the analysis of SQL statements, for which the optimizer has selected a suspicious access path, often starts the index improvement process. The following performance alarms that we have studied in some depth are quickly detected with EXPLAIN.

Full Table Scan or Full Index Scan

If the most suspicious access paths, scanning the whole index or the whole table, have not already been checked before cutover, now is the time to do it.

Sorting Result Rows

A sort of the result rows is the most useful alarm after that for full scans. There may be two reasons for the sort:

1. There is not an index that would make the sort unnecessary for the SELECT statement.
2. The access path chosen by the optimizer includes a redundant sort.

In the first case, the sort can be eliminated by an improvement of the index (refer to Chapters 4 and 5). The second case will be discussed in Chapter 14.

Often a sort does no harm. For example, an application may have a thousand different SELECT statements with hundreds of them having a sort in their access

path; 90% of these statements may have excellent performance with the sort contributing an insignificant amount to the response time. Thus, the large number of false alarms may make this review method somewhat unproductive.

There are many database consultants who consider the sort to be *the* enemy. We believe that those consultants who emphasize *death by random I/O* are more credible.

Cost Estimate

The EXPLAIN feature of several DBMSs display the optimizer's estimate for the local response time, or at least the CPU time, for the chosen access path. Some products, SQL Server 2000, for instance, show the estimated CPU time and I/O time for *each step* in the access path. In a simple case there might be several steps such as:

1. Retrieve index rows.
2. Sort pointers.
3. Retrieve table rows.
4. Sort result rows.

Experience has shown that it can be dangerous to place *too much* reliance on the optimizer's cost estimates. After all, the optimizer is only producing the cost estimate in order to choose the fastest alternative access path available. The estimates sometimes give false alarms, yet don't show *all* the alarms; but it is an easy and fast tool to use, and enables an *early* check to be made. Consequently SQL statements with unusually high cost estimates—perhaps the 20 statements that have the highest cost estimates belonging to a new application—should be checked. It is quite likely that inadequate indexing or optimizer problems will be discovered.

Unfortunately, two serious issues limit the value of using the cost estimate in this way:

1. The optimizer *estimate* for the local response time may be very different from the *actual* value (refer to Chapter 14).
2. Whenever a predicate uses host variables (which is very common, of course), the optimizer estimate for the filter factor is based on the *average* input, or even worse, a default value. To obtain the more valuable *worst-case* estimate, the host variables in the EXPLAIN must be replaced by the worst input literals. This is a tedious operation that requires application knowledge.

EXPLAIN is an indispensable tool for analyzing optimizer problems. This is its *raison d'être*, not index design. Airport security does not only check *suspicious* looking passengers at the airport!

DBMS-Specific EXPLAIN Options and Restrictions

The EXPLAIN features of many DBMSs have been enhanced significantly over the past 10 years. As well as a compact report producing a single line per execution step, graphs are available at the statement level. The *suspicious* access paths may even be marked in some EXPLAIN outputs. The statistics on which the optimizer's decision was based, together with a list of all the current indexes, and their characteristics, for each accessed table may be provided, which is extremely useful; without this, it would be necessary to access this information from the system tables separately. The best EXPLAINs have already integrated the next desirable step: to propose index improvements.

On the other hand, the current EXPLAINs still have a few irritating shortcomings. DB2 for z/OS V7 does not report index screening (rejecting a row by using column values copied to the index row). This shortcoming does not directly affect index reviews, but it does mean that some optimizer problems are only revealed by measuring the number of table touches. Oracle 9i does not report the number of index columns involved in the screening process; in fact it doesn't even report the number of matching columns either (the PLAN_TABLE column SEARCH_COLUMNS in Oracle 9i is not currently used). Oracle 9i does report the type of index scan: UNIQUE SCAN (one row), RANGE SCAN, FULL INDEX SCAN, or FAST FULL INDEX SCAN. Thus, the most suspicious access paths are revealed, but the lack of the MC value makes it more difficult to analyze optimizer problems caused by difficult predicates (sometimes called *index suppression* in the Oracle literature).

MONITORING REVEALS THE REALITY

Fortunately, most DBMSs now provide a trace that records the components of the local response time, or at least the number of events taking place, such as physical reads. The overhead of the required trace activity, at one time somewhat prohibitive, now tends to be quite acceptable: In DB2 for z/OS V7, for instance, it is a few microseconds of CPU time per SQL call. For a typical operational transaction this means an increase in CPU time of a few percent, and even less for the elapsed time. However, for a batch job that is mostly issuing simple FETCH calls and causing sequential scans, the elapsed time may go up by as much as 20%.

Compared to EXPLAIN, measurements are a superior starting point in the analysis of a slow program; this includes access path problems as well as all the other reasons there may be for long elapsed times. However, reading monitor reports may take a huge amount of time, particularly if a good monitoring strategy is not implemented.

Measuring the performance of a computer system is a complex task—it can be viewed from many different angles. The simple *spike report*, introduced later in this chapter, is the result of a long evolution. An understanding of its history will make it easier to appreciate the essence of the recommended approach.

Evolution of Performance Monitors

The Sixties

The first monitors had two main objectives: billing, on the basis of the consumed CPU time, and monitoring the hardware load. The monitors were usually a part of the operating system. The load factors such as CPU busy, channel busy, and volume busy were measured by sampling.

With regard to tuning, the most useful report was the one providing disk I/O response time, together with its components broken down by disk volume. Files were often moved from one volume to another to balance the load across volumes. In extreme cases, active files were even moved close to each other on the disk drive in order to reduce the average seek time.

The Seventies

The first DBMSs were enhanced with trace records that enabled the monitoring of elapsed time by program. It was now possible to follow trends in the average local response time and to identify slow programs. IMS, for instance, provided a monitor report that showed the elapsed time by DL/I call and return code. Although this report could be quite long, it was easy to find the calls that were exceptionally slow. The most common reason for this was incorrect sizing of the root addressable area, which contained root anchor points for the randomizer. Many programming errors were identified by the use of this report; for instance, a DL/I call might have been using the wrong index. There were no optimizers in the prerelational DBMSs.

The Eighties

When the move to relational databases began, tuning focused on the optimizer. EXPLAIN was the most important tool. Performance courses also emphasized detailed SQL trace reports, but users soon realized that they could solve almost all performance problems simply by checking the program counters and response time components using reports such as the DB2 Accounting Trace.

As the DBMS suppliers added more and more trace records, these performance reports became longer and started to look complicated. At the same time, applications grew more complex. Reading monitor reports became too time consuming. Both DBMS suppliers and third parties—companies specializing in performance monitors and utilities—invested a great deal in more user-friendly summaries, exception reports, and graphics. Real-time monitors became popular in many installations because they showed the reasons for gridlocks and other system-level problems. However, batch reports based on long measurement periods were found to be the only reliable way to find significant application performance problems. One of the most popular reports provided key average values by program name or transaction code. Although it could span several hundred

pages, the consistently slow programs were easy to find. If the CPU time or the number of disk reads was high, EXPLAIN showed the suspicious SELECT statements. Already in the eighties, inadequate indexing was the most common reason for poor performance, but optimizer problems were a close second. The optimizers were not quite as smart as they are now, but a more important reason for a bad access path choice (e.g., scanning the whole table although a perfectly good index was available) was the general lack of understanding of the optimizer. Optimizer-specific pitfall lists were not common in performance guides and course materials until the late nineties.

The Nineties

The applications grew even more complex and the tolerance level of the users became lower. Many users were no longer satisfied with a subsecond *average* response time. They demanded that *no* interactive transaction should take several seconds. To take just one example, a customer request for proposals for a seat reservation system insisted that, for a specified peak load, *every single transaction* must have a local response time of less than 5 s. This was verified in extensive stress tests and measurements after production cutover before the system was accepted.

Now, exception reports became necessary. The focus moved from averages by transaction code to profiles of individual slow transactions. Many performance monitors included this option in the nineties, the installation defining the exception criteria; but the profile for each exceptional transaction could span 10 pages and contain thousands of numbers.

The next natural step was to condense 10-page reports down to a summary containing a single line per transaction, choosing the most valuable numbers and ratios from the standard long report. In Finland, the first such report, called a *spike report*, was implemented in the mid-nineties. Now it was possible to increase the measurement period from minutes to hours with statistically more reliable results; one could say with confidence which problems were the most common ones. On the other hand, even infrequently executed slow programs were caught. Furthermore the analysis effort was significantly reduced. When the profile of a transaction fits on a single line, it becomes very easy to understand the information. A spike report sorted by *transaction code* and *end time* immediately reveals the transaction programs that are frequently slow. When the report is created daily, it is easy to see when a program started to become slow and when tuning action, such as helping the optimizer or adding columns to an index, resolved the problem. A spike report sorted by end time is valuable when looking for the cause of long lock waits or when analyzing the effect of disk write bursts on the duration of synchronous database reads.

In addition to spike reports, which show slow transactions, it is useful to see the slowest *individual* SQL calls. Monitors for this purpose came to the market in the nineties.

2000 to 2005

Spike reports have little value if nobody has time to analyze them and draw the right conclusions. In the last few years, perhaps encouraged by a serious worldwide shortage of DBAs, the demand for self-tuning DBMSs has become more pronounced. Among the first steps toward this ambitious goal are tools that propose table and index reorganizations or buffer pool sizing. Advisors for updating statistics for the optimizer, possibly including recommendations (e.g., histograms) per index and table, are expected in the near future. The next step for such a tool might be the reassessment of the access paths based on filter factors measured at run time, in order to start processing over again with an improved access path. This is, after all, how intelligent people behave in many situations.

The spike report, currently used in several DB2 installations, draws attention to transactions that appear to be access path culprits or lock victims. These indicators are valuable in enabling busy database specialists to quickly and easily detect new performance problems arising from changes relating to program maintenance, database growth, or changes in usage patterns.

LRT-LEVEL EXCEPTION MONITORING

Averages per Program Are Not Sufficient

A report showing the *average* local response time per program reveals the programs that are consistently slow. The usefulness of averages is limited, however, because one program often contains several different functions; even the duration of one function may vary a great deal depending on the input, such as whether it is for the average customer or the largest customer.

Monitoring *individual transactions* should be done as soon as possible, preferably during the test phase, in order to catch the worst input response times. Furthermore, it is much easier to analyze the profile of an individual transaction than the average profile of many different transactions using the same program; the latter is often as confusing as a comparison of the average characteristics of a few trucks and a lot of bicycles.

Exception Report Example: One Line per Spike

The information we recommend for an exception report that produces a single line for each exception transaction, a spike report, is shown below. We will use the term *spike* in this chapter to represent a single occurrence of a transaction that has been detected by the exception monitor. There may well be many other occurrences of the same transaction that are performing perfectly satisfactorily.

Identification

- Program name

- The name of the dominant module (contributing most to the LRT)
- The contribution of the dominant module to the LRT (%)
- Date
- End time (hh.mm.ss)

Transaction Profile

- Local response time (s) *LRT*
- Total elapsed time of the SQL calls (s) *SQL*
- CPU time of the SQL calls (s) *CPU time*
- Synchronous read time (s) *Sync read*
- Wait for prefetch time (s) *Wait for prefetch*
- Lock wait time (s) *Lock waits*
- Other wait time (s) *Other waits*
- Average duration of synchronous reads (ms)
- The number of synchronous reads (table pages)
- The number of synchronous reads (index pages)
- The number of executed SQL calls
- The number of table pages accessed
- The number of index pages accessed
- The number of sequential prefetch requests
- The number of commit points
- Quick diagnosis (derived from the above figures)

The terms highlighted to the right of the first seven items of the transaction profile (e.g., *LRT*) refer to the terms shown in the bubble charts used in this chapter.

An example of a transaction that would be identified during exception monitoring, showing these *transaction profile* components, is shown in Figure 7.1. Displaying the information from the exception report in this way makes it very easy to identify any problems.

A discussion on the meaning, interpretation, and any resultant tuning of these numbers will be found a little later.

Culprits and Victims

A spike report, covering perhaps the peak hours of the week, may contain thousands of spikes from more than a 100 different dominant modules. A database specialist may be able to analyze and fix, with a little help from application developers, perhaps 5 to 10 modules per week. The spike report for the following week may show new modules because of program changes or different user input. To productively use the limited time normally available, it is important

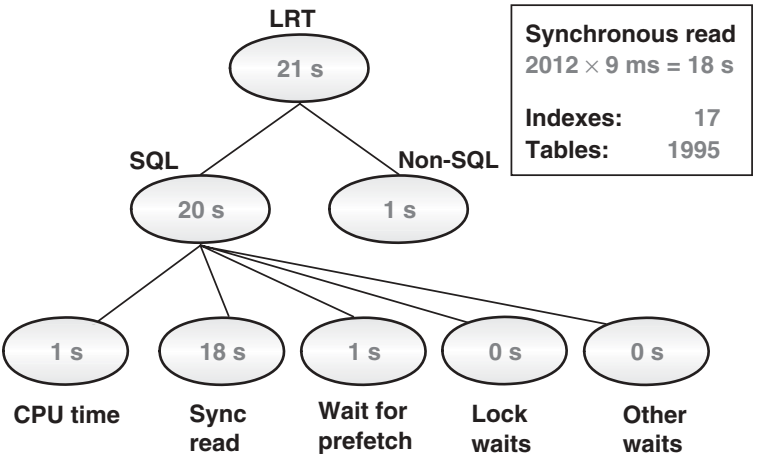


Figure 7.1 A spike example.

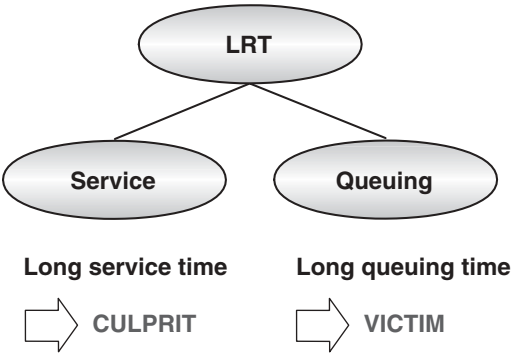


Figure 7.2 Culprits and victims.

to carefully prioritize the modules analyzed; the transaction response times and frequencies are not the only criteria.

The first distinction we should make is that between the *culprit* and the *victim*, as shown in Figure 7.2. If a transaction monopolizes resources, perhaps because of inadequate indexing, it will obviously have an adverse effect on *other* transactions, and so these will consequently appear on the exception report together with the culprit.

The resources used by the culprit will be included in the service time; the victim will be affected by having *to wait* for the resources, namely the queuing time.

It is logical to begin the tuning process by addressing the problems caused by the culprit rather than the victim.

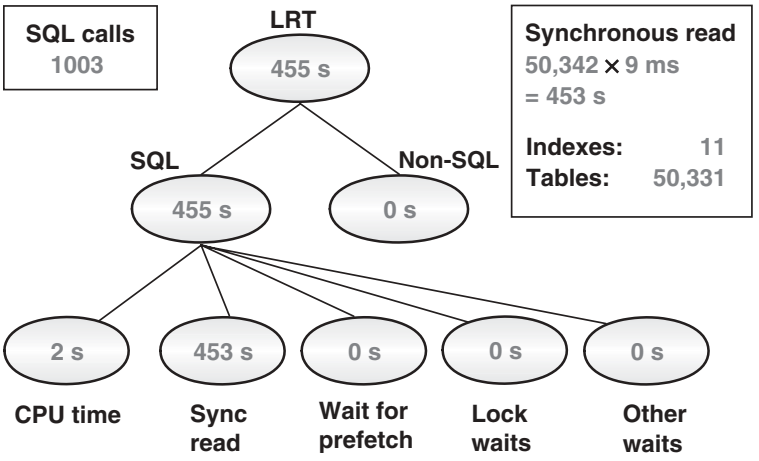


Figure 7.4 Promising culprit.

- Synchronous reads are the major cause (453 s) of the excessive local response time (455 s).
- The average time for a synchronous read is fairly short (9 ms); disk queuing is probably not significant.
- Almost all the synchronous reads access table pages (50,331).

The expected average time for a synchronous read depends on the read cache hit ratio. If the read cache is much larger than the database buffer pool, say 100:1, the expected value for the average of the cache reads and the drive reads could be as low as 3 ms. If that had been the case in this example (9 ms synchronous read time), the drive queuing time would be significant; reducing the number of synchronous reads would still, however, make a big difference. If a disk drive is seriously overloaded, perhaps due to a write burst, the average synchronous read time for a transaction could be hundreds of milliseconds. Then the transaction is clearly a victim.

Analyzing Promising Culprits

A promising culprit reads many table pages. We know that there are three ways to read table pages:

1. Traditional random read (synchronous read, SR)
2. Skip-sequential read
3. Sequential read (sequential prefetch)

We can deduce from the spike report which of these three methods was largely used by the promising culprit to read the table pages from disk.

Synchronous reads (type 1) do not overlap CPU time; the application program stops and waits for the arrival of the requested page from the disk server. The spike report shows the number of pages the transaction read synchronously together with the average wait time (the suspension time) for these synchronous reads.

Asynchronous reads (types 2 and 3) are prefetch reads; the I/O time does overlap the CPU time of the program; the DBMS asks for the next few pages from the disk server while the program is still processing pages in the database buffer pool. The program may or may not run out of pages to process. If it does, the wait time is recorded in the Wait for Prefetch counter. If the program never has to wait for prefetched pages, it is CPU bound. The Wait for Prefetch component in the spike report is then zero, and the elapsed time for the sequential scan is reported partly as SQL CPU Time and partly as Other Wait (for any CPU queuing).

Skip-sequential read (type 2) tends to be I/O bound with current hardware. Wait for Prefetch may be the largest component of the local response time if most of the skips are large, meaning that the pages containing the qualifying rows are far from each other.

Sequential read (type 3) is roughly balanced with current hardware; sometimes the CPU time per page is a bit longer than the I/O time per page, sometimes a bit shorter. The QUBE assumes that the larger of the I/O and the CPU times is up to 0.01 ms per row. If sequential read is I/O bound, the elapsed time of a sequential scan is equal to the disk I/O time, which in turn is the sum of the CPU Time and the Wait for Prefetch components of the spike report.

Many programs access table pages in more than one of these three ways. Interpreting the spike report is then not so straightforward, but the dominant method is normally not too hard to deduce. The total number of table pages accessed may help. This contains, in addition to the synchronous reads, pages read by prefetch (both types 2 and 3) as well as buffer pool hits (the table pages found in the database buffer pool).

Many DBMSs are nowadays able to perform sequential processing in parallel, for instance, by breaking a cursor into several internal cursors, each using one processor and creating its own prefetch stream. If this option (CPU and I/O parallelism) is enabled, it is normally the optimizer that determines the degree of parallelism. If there are enough processors, disk drives, and buffer pool space, the degree may be perhaps 10. The elapsed time may then, theoretically, be almost as low as 10% of the nonparallel elapsed time. This option is invaluable when scanning massive tables and their indexes, data warehouse fact tables, for instance, but most installations with traditional operational transactions and batch jobs disable parallelism because a program that exploits several processors and drives at the same time may cause unacceptable queuing times for other concurrent users.

Tuning Potential

Before spending a great deal of time analyzing an exception transaction, it is worthwhile estimating by how much the local response time may be reduced as

a result of planned index improvements. The tuning potential is the *upper* limit for the achievable reduction.

Random Reads

The tuning potential of read only transactions is the product of the number of table pages read synchronously and the average duration of a synchronous read. The local response time will be reduced by this amount if *all* table page reads are eliminated by means of fat indexes. Thus, in Figure 7.5, the tuning potential is $50{,}331 \times 9$ ms $= 453$ s if the program is read only; programs with INSERT, UPDATE, or DELETE calls have a smaller tuning potential because the table pages *must* be updated no matter how fat the indexes.

Skip-Sequential Reads

Some optimizers would choose skip-sequential read in the previous example. The DBMS would then first collect the pointers from all qualifying index rows and then sort the pointers by table page number. Now the DBMS has a sorted list of all the table pages that contain at least one qualifying row; only now will the table be accessed. Naturally, the I/O time per page will be shorter, especially if several qualifying rows happen to be on the same track. In addition, the I/O time will overlap the CPU time because the DBMS is able to read ahead. The saving compared to synchronous reads is very variable; it depends on the average distance between the pages to be read and on the striping implementation; the tuning potential is simply the current value for Wait for Prefetch (assuming the program is read only and all prefetch is skip-sequential, e.g., list prefetch in DB2 for z/OS). Thus, the tuning potential in Figure 7.6 is 102 s.

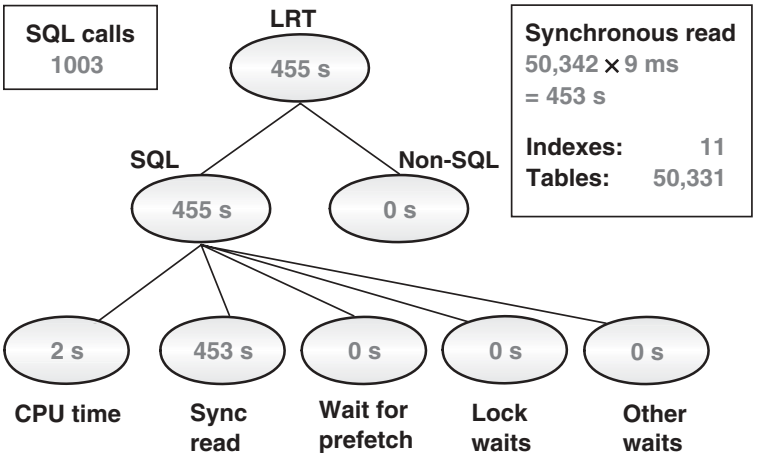


Figure 7.5 Promising culprit—random reads.

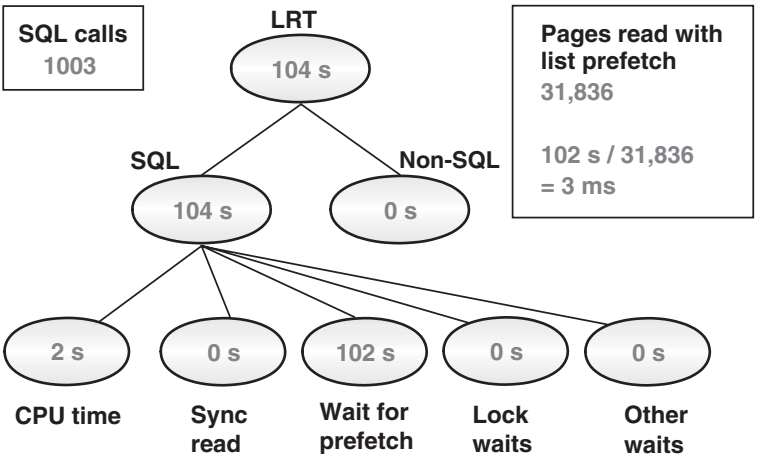


Figure 7.6 Promising culprit—skip-sequential reads.

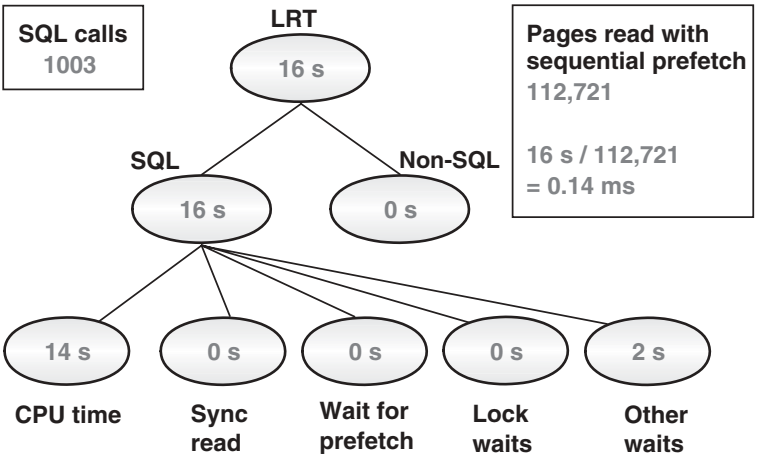


Figure 7.7 Promising culprit—sequential reads.

Sequential Reads

The spike report shown in Figure 7.7 shows an application that could be scanning an entire table (112,721 pages read using sequential prefetch). Because there is no I/O wait time, the program is CPU bound. The 2-s Other Waits figure might possibly be due to CPU queuing.

At first sight, the spike in Figure 7.7 may look like an unpromising culprit. It is easy to think of cases in which 1000 SQL calls take 16 s, even with good access paths; for instance, 1000 singleton SELECTs that cause, on average, two random reads. However, the biggest component is now CPU time, not I/O time.

This suggests that processing is mostly sequential. Sequential scans are often CPU bound. Random processing is CPU bound only if all the pages accessed are in memory. In addition, the report shows that about 100,000 pages *were* read with sequential prefetch. These pages can be either leaf pages or table pages, or both. It's possible, for instance, that a very thick index slice and a very thick table slice are read, the index and table rows being in the same order. Nevertheless, most rows are rejected. If we assume that a sequential touch takes roughly 10 μs (this will be fully discussed in Chapter 15), the total number of touches must be roughly

$$\frac{14{,}000{,}000\ \mu s}{10\ \mu s/\text{touch}} = 1{,}400{,}000\ \text{touches}$$

assuming that the number of random touches is much lower than the number of sequential touches.

From this, it should be quite easy to find the SELECT(s) that may generate massive sequential scans, more than 100,000 pages and probably more than 1,000,000 rows.

The tuning potential (assuming the 2-s *other wait* is caused by CPU queuing) is then almost 16 s. It should be possible to find an access path that reads much thinner slices.

If the slow access path contains a full index scan or a full table scan, would that not have been noticed already in the quick EXPLAIN review? Not necessarily. The access path may be using an index with one matching column for a range predicate. With the worst input the index slice to be scanned may, in fact, be the whole index. Figure 7.8 shows an example of such a case.

The optimizer may choose the index (IDATE, CNO) if it assumes a small filter factor for the predicate IDATE > :IDATE. As IDATE is the *only* matching column (CNO cannot participate in the matching process because the first column

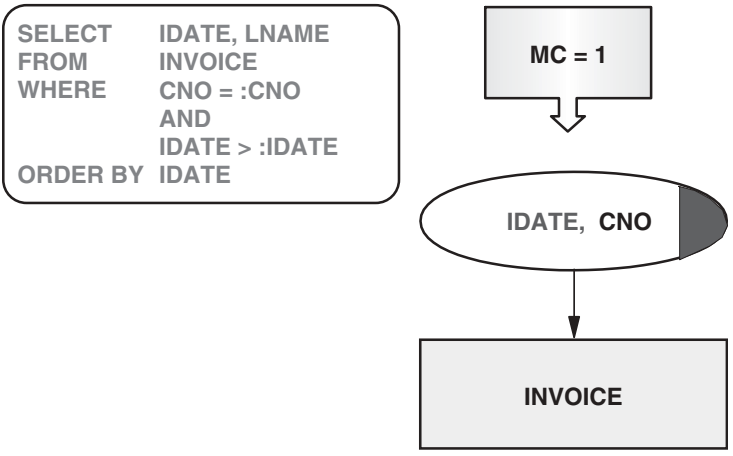


Figure 7.8 Whole index may be scanned even though MC = 1.

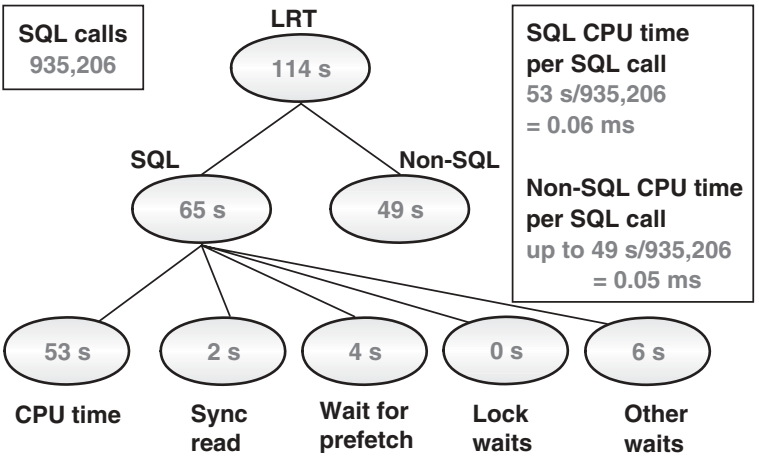


Figure 7.9 Unpromising culprit.

is a range predicate), the DBMS would scan the whole index if the user entered a very old date.

Unpromising Culprits

The spike depicted in Figure 7.9 probably indicates a predominantly CPU-bound sequential read (only 4-s Wait for Prefetch). The large number of SQL calls, 935,206, is bad news. Ideally, this number of SQL calls would imply that a similar number of rows should be examined, at a cost of 0.1 ms per row (no rows being rejected). The cost (absorbing the sequential touches within the cost of the FETCH processing) would then be

$$935{,}206 \times 0.1 \text{ ms} = 93 \text{ s}$$

Actually 93 s is higher than the measured CPU figure of 53 s; clearly the criteria in place for this query differ somewhat from those on which our 0.1-ms performance guideline figure is based. Nevertheless, this is the *minimum* cost of the SQL calls, and so we can't make them any faster. Consequently, the only way to reduce the SQL CPU time is to reduce the *number* of SQL calls. Sequential read may then become I/O bound, so the elapsed time may not decrease by 90% even if the number of SQL calls was reduced by 90%.

The high number of SQL calls also explains the large amount of non-SQL time (49 s). A part of each SQL call is reported as non-SQL time on many platforms. The 50 (49 s/935,206) μs between two successive SQL calls was mostly due to this overhead (sending an SQL call and receiving the response from the application program); the CPU time consumed by the application program, written in PL/I, was certainly much less in this case.

A spike like this was found to be a database integrity check run once a day by a user. According to the programmer, the number of SQL calls could have been reduced by adding predicates to a WHERE clause; this was not considered to be worth the effort because the long response time was not really a problem for the only user of the transaction. To avoid unnecessary queuing for other concurrent transactions, it was decided that the transaction should run early in the morning!

In general, transactions with a high number of SQL calls, say more than 10,000, are hard to improve if the CPU time (SQL CPU and non-SQL time) is the largest component of the local response time.

The tuning potential is the number of SQL calls eliminated multiplied by the base CPU time per SQL call (e.g., 0.1 ms).

Victims

A small table once caused spikes like the one shown in Figure 7.10. The table contained a few counters that were used to collect statistics. The rows were updated by many transactions. These updates were often made early in the program, usually unnecessarily early; that is to say, they were not done close to the commit point. Sometimes the transactions updating the counters were somewhat slow, perhaps because of disk drive queuing. Consequently, a row could be locked for a few seconds, causing other transactions needing to update the row to wait.

Lock implementation varies considerably from one DBMS to another, but lock waits tend to be application design problems. The first thing to be done is to find the hot resource—the row or page that causes lock waits. If the DBMS provides a lock wait trace, it is quite easy to identify the resource by running the trace for the program suffering from lock waits. An alternative approach

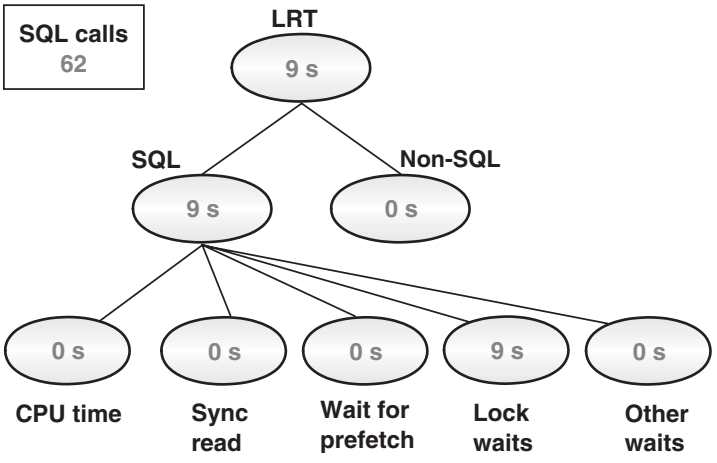


Figure 7.10 Victim.

is to determine which tables are accessed by the module suffering from long lock waits; assuming index pages or rows are not locked by the DBMS, the hot resource must be a page or a row in one of these tables—probably the most popular row in a small table or the last page in a large table when new rows are inserted at the end of the table.

Basic queuing theory is useful when analyzing lock wait problems.

$$Q = u/(1 - u)S$$

where Q = average queuing time
u = utilization (resource busy)
S = average service time (resource busy per request)

This simple formula assumes a single server and a Poisson distribution for the interarrival times and the service times.

In database locking, the time an object is "busy" (locked) is the product of the lock request arrival rate and the average lock duration:

$$2 \text{ requests per second} \times 0.2 \text{ s per request} = 0.4$$

Our experience shows that the alarm limit for resource locked (u) should be no more than 0.1; this means that the object is locked for 10% of the time during peak hour. The average queuing time would then be $u/(1 - u) = 11\%$ of the resource locked time. Because of random variation, some individual queuing times will be *much longer* than the *average* queuing times, and these sorts of figures would obviously be likely to cause visible problems. When an object is locked for say, 50% of the time, the formula shows that the *average* lock wait time would become *equal* to the lock duration; some lock waits would be very much higher than the average; at this level the problems are likely to be *very serious.*

Index improvements reduce lock durations (S) and therefore also lock waits (Q). This is why we made the recommendation to start the tuning process by analyzing the promising culprits; lock victims should probably be the next step. Locking problems are often relatively easy to resolve once the bottlenecks have been found.

Other victims, such as transactions with long CPU or disk drive queuing times, are normally difficult to resolve. Long queuing implies high utilization. The only solution may be a hardware upgrade, assuming the significant culprits have already been tuned. Transactions that have to wait for events such as opening or expanding a file may also be classified as victims. Delays like these are normally system tuning problems; they may be unavoidable for infrequently used files or on Monday mornings if the system has been restarted (and the files closed) during the weekend.

These unusual waits are recorded in Other Waits time (which also includes CPU queuing). More detailed exception reports might identify the type of event or wait. For this reason it is wise to save the detailed reports for a week or two.

For lock victims, the tuning potential is equal to the lock wait time, 9 s in Figure 7.10. For other victims, it is equal to the relevant component in Other Waits time.

Finding the Slow SQL Calls

Having identified the dominant component of the SQL time (SQL CPU time, Synchronous Read, Wait for Prefetch, Lock Waits, or Other Waits), the next step is to find the SQL calls that contribute the most to this component (it should be realized that the figures shown in the bubble charts may well refer to a *large number of different* SQL calls). This can be done simply by reading the SQL statements of the dominant module. If the problem is due to CPU time or I/O time, many statements may be excluded immediately, for instance, those that access a row using the primary key (WHERE PK = :PK). Another way is to use an SQL trace for the program if one is provided by the DBMS. These reports show the elapsed time, together with the main components of the elapsed time, by SQL statement.

The most commonly used (and affordable) trace is provided at the module level. In this case, identifying the slow SQL calls becomes easier the smaller the modules. From the tuning point of view an application built with one large module is a nightmare. At the other extreme, some modules may contain only a single SQL statement. The SQL statement that contributes significantly to the local response time is then identified directly from an exception report.

When the SQL calls have been identified and shown to be caused by inadequate indexes, improvements need to be made according to the techniques addressed in Chapters 4 and 5.

CALL-LEVEL EXCEPTION MONITORING

Reports that show the slowest SQL calls in a monitoring period are the next best thing after LRT-level exception monitoring. They can also be of great assistance in finding the slow SQL calls in a slow program that has a large number of different SQL statements.

We will assume we have taken a peak hour trace and produced a list of the worst SQL calls, based on the longest elapsed times during that period; here we will be dealing with *individual* calls, not the *averages*. Let us further assume that the tool we are using, as is often the case, only provides four relevant measurements for each call; these are (together with the abbreviations we will use): *the elapsed time (ET), the CPU time (CPU), the number of pages read from disk (READS), and the number of database page requests (PR).* A database page request is called *get page* in DB2, *logical I/O* in SQL Server, and *get* or *LIO* in Oracle; to keep everything as simple as possible, we will use a single term that reflects what they all mean—page requests (PR); refer to Figure 7.11.

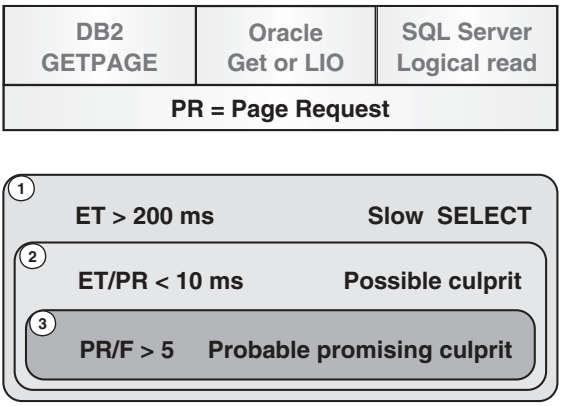


Figure 7.11 Filtering slow SQL calls.

Starting with the SQL calls that have the longest elapsed time (ET), in order to improve the response time by access path tuning, we should try to answer the same two questions that were discussed earlier at the LRT level.

First, does the SQL call appear to be a culprit? In other words, is the ET too long because the *service time* is too long? There are several ways to approach this question. A high CPU time indicates a long service time, and so does a high value for READS; but, in fact, PR seems to be the most useful indicator for service time because it correlates both to CPU and to READS, and is to a large extent repeatable; it does not depend on the system load or warm-up effects.

To identify the SQL calls that are likely to be culprits, we should ignore calls that have a low PR. If a SELECT takes 1 s and issues less than 100 page requests, it is probably a victim. In the worst possible case, each of these would result in a random read; this would take 100×10 ms $= 1$ s if drive queuing is not excessive. In reality, many of the page requests would be satisfied from the database buffer pool and some from the disk server read cache; remember that the reported PR includes nonleaf index pages; also many page requests are sequential. The expected I/O time for the 100 page requests is much less than one second and so is the expected CPU time. All in all, if the measured elapsed time for an SQL call with 100 page requests is one second, it is very probable that the *queuing time* (disk drive queuing, CPU queuing, lock waits, or other waits) is longer than the *service time*. In general, if the elapsed time (ET) for a slow SELECT is more than the number of page requests (PR) $\times$ 10 ms, the SELECT is probably a victim; it belongs to the first subset (1) in Figure 7.11 but not to the second subset (2).

The possible culprit subset (2) consists of SQL calls whose ET/PR is not very long; they are not probable or obvious victims. However, subset (2) may contain many victims as well; for instance, calls that do sequential processing (low service time per page), which are slow due to excessive CPU queuing time. If we were aware of any lock wait or drive and CPU queuing time, as we would be with LRT-level spike reports, we could filter out most of these victims. As it is, we only have the four values, and so unfortunately the second subset (2) may

be quite large. This is why the second question is important: Does the SQL call appear to be a *promising* culprit?

To find the promising culprits, the third subset (3), we need to determine the number of result rows. If the number of FETCHes (F) is included in the exception report (or if it is easily obtained), we can use it to provide an approximate number of result rows. If PR/F is more than 5, the SELECT is *probably* a promising culprit because a typical FETCH with a good access path should require no more than one or two page requests per table. The most likely cause would be a large number of unproductive random touches, many rows being touched and then rejected.

Subset (3) contains SELECTs that cause a large number of random reads because they do not even have a semifat index; these are the most rewarding calls. It will also contain some SQL calls that do actually have a good access path (e.g., SELECT COUNT in a join with fat indexes), but most of the SELECTs are likely to have interesting access path problems.

These should first be analyzed with the Basic Question (or the Basic Join Question, to be discussed in the next chapter). This may immediately reveal inadequate indexing. The next step is to EXPLAIN these SQL calls to see which access path has been chosen by the optimizer. This may reveal an optimizer problem: a difficult predicate (too few matching columns) or a poor filter factor estimate resulting in a wrong index or wrong table access order, for instance. If the optimizer is not guilty, we should design better indexes.

The values CPU and READS provide useful pointers to the access path. If CPU is close to ET, for instance, processing must be mostly sequential. If CPU is low, ET/READS will normally be close to the average I/O time per page. If that value is clearly less than 1 ms, most of the I/Os are sequential; if it is several milliseconds, most of them are probably random. This will complement the information provided in the execution plan. In addition, any SQL call with a high CPU, say more than 500 ms, can be considered to be an interesting culprit.

When the top calls in subset (3) have been analyzed (those that have at least one very long ET and those that are frequently quite slow), it is time to look at the slowest and most frequent SQL calls in the possible culprit subset (2). We may then find SELECTs that are much too slow, even with a semifat index, and which could be made dramatically faster with a fat index; with a profile such as this, for instance:

$$\begin{aligned} \text{ET} &= 8\ \text{s} \\ \text{CPU} &= 0.06\ \text{s} \\ \text{READS} &= 900 \\ \text{PR} &= 1050 \\ \text{F} &= 1000 \end{aligned}$$

After a few SQL calls have been improved with better indexing or by helping the optimizer, a new exception report should be produced. The index improvements may have helped many SQL calls, culprits as well as victims, but it is possible that the next list produced may show *new* access path problems—mainly

due to changes in the workload, or sometimes (fortunately not very often) because of the index changes.

How far down the list, sorted by ET, would it be necessary to go? In an operational system response times exceeding 1 s should be rare. Ideally then, we should check any SQL call that sometimes takes more than 200 ms. Unfortunately, in real life, we may never be able to do this. So many SQL calls, so little time!

The terms and criteria we have discussed above are summarized in Figure 7.11. In order to illustrate how they may be used, two examples of call-level monitoring will be provided shortly, one each for Oracle and SQL Server.

This approach will reveal many serious access path problems with only minimal effort. Why then, do we claim that LRT-level exception monitoring may be preferable to call level?

First, a report showing the worst SQL calls does not reveal problems with *fairly fast* calls that are *executed many times* per transaction. Consider, for instance, a singleton SELECT executed 500 times in a loop. If the access path is not index only, the average time per call may be 20 ms with two random reads. Few analysts would be concerned about an *SQL call* taking 20 ms, but most would be *very concerned* about a *transaction* taking 10 s (500 × 20 ms)! Adding any missing columns to the index would cause the contribution of the SELECT to the overall LRT to be reduced from 10 to 5 s.

Second, LRT-level exception monitoring reveals *all* response time problems, not just those related to slow access paths. It also gives an early warning of problems that are beginning to be significant as databases grow or transaction rates increase.

Oracle Example

In the Oracle 9i exception report, the numbers shown below are reported for each slow SQL call—the N slowest calls in a measurement period, for instance.

```
Statistics
-----------------------
   0 db block gets
1021 consistent gets
1017 physical reads
CPU time      0.031250 s
Elapsed time  6.985000 s
```

Consistent gets represent the number of blocks requested in consistent mode and *db block gets* is the number of blocks requested in current mode. Blocks must be requested in current mode in order to be changed. Together these two values comprise the total page requests (PR) for the SELECT.

Physical reads represent the number of database blocks that the database instance has requested the operating system to read from disk.

This call took 7 s and consumed 31 ms of CPU time. It accessed 1021 database pages (blocks); almost all of them, 1017, were read from disk.

Is this SQL call a victim? Probably not, because

$$\text{ET/PR} = 7000 \text{ ms}/1021 = 7 \text{ ms: less than 10 ms}$$

Is this SQL call a promising culprit?

When we discover that the result table consists of only 5 rows (which is the case measured), we can see that this SELECT is a very promising culprit: PR/F = 1021/5 = 200. We would expect a large number of unproductive random touches.

If the result table had been much larger, PR/F could have been below 5, with the other values broadly similar other than CPU being a little higher. This SQL call would then be found with the slowest ones in subset (2) of Figure 7.11. In any case, we can make the following observations:

Reading 1017 pages from disk implies a long service time. As the CPU time is short, the I/O time may be close to the elapsed time, about 7 s. The average I/O time per page is then about 7 ms; most of the reads are probably random. Even if 1000 rows were being retrieved, we wouldn't want a random read for each. If only a few rows were being retrieved, we would be even more surprised. We need to look at the SQL call and determine what access path was used and which indexes were involved. This information hopefully will not only explain why the SELECT is behaving in this way but also show what improvements should be made.

SQL 7.1

```
SELECT   CNO, FNAME, ADDRESS, PHONE
FROM     CUST
WHERE    LNAME = 'LNAME287'   FF = 0.1%
         AND
         CITY = 'TRURO'       FF = 0.5%
ORDER BY FNAME
```

```
Execution Plan
----------------------------------
 0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1020
        Card=5 Bytes=545)
 1    0  SORT (ORDER BY) (Cost=1020 Card=5 Bytes=545)
 2    1   TABLE ACCESS (BY INDEX ROWID) OF 'CUST'
          (Cost=1016 Card=5 Bytes=545)
 3    2    INDEX (RANGE SCAN) OF 'CUST_X1' (NON-UNIQUE)
           (Cost=16 Card=5)
```

The only useful index on table CUST is CUST_X1 (LNAME, FNAME). Table CUST has 1,000,000 rows. The filter factors are shown next to the predicates above in SQL 7.1.

The execution plan shows that index (LNAME, FNAME) was used; a slice was read (RANGE SCAN), not the whole index. The table was accessed and then, the only surprise, the result was sorted. The sort is not a problem, however, because the result is only 5 rows. This happens to be exactly what the optimizer had assumed (CARD = 5) because the input was chosen to represent the average case.

The measured times are now easy to understand: 1000 sequential touches to the index cause a relatively small number of disk I/Os; if there are 40 index rows per leaf page, the number of leaf pages read sequentially will be 25. This should take only a few milliseconds. The average time per random read seems to be about 7 ms, which is quite a short time. Oracle may have used data block prefetching to do random reads in parallel (an unofficial access path, not reported in the execution plan). That would explain the surprising sort in the execution plan.

The QUBE for the *current index* with the *above* input is

$$1001 \times 10 \text{ ms} + 1000 \times 0.01 \text{ ms} + 5 \times 0.1 \text{ ms} = 10 \text{ s} : \text{measured } 7 \text{ s}$$

The reason for the long response time, 7 s, can now be ascertained by comparing the SELECT statement with the index: Column CITY is not in the index. The index is not even semifat! As the filter factor for predicate LNAME = 'LNAME287' is 0.1%, Oracle must do 1000 random touches to table CUST to evaluate predicate CITY = 'TRURO' in order to find the required 5 rows. The solution is obvious: to at least add CITY to the index, thereby making it semifat to eliminate the table touches.

Before we rush ahead and devise a solution, we should reflect that 5 rows is actually a very small result set; consequently, this may not indeed be the worst case. Any solution considered should also be adequate for the worst case; a semifat index may not be adequate for the *worst* input. The filter factors for this are LNAME 1% and CITY 10%.

The QUBE for the *current index* and the *worst* input is

$$10{,}001 \times 10 \text{ ms} + 10{,}000 \times 0.01 \text{ ms} + 1000 \times 0.1 \text{ ms} = 100 \text{ s} : \text{measured } 72 \text{ s}$$

while for a *semifat index* with the *above* input, the QUBE would be

$$6 \times 10 \text{ ms} + 1000 \times 0.01 \text{ ms} + 5 \times 0.1 \text{ ms} = 0.07 \text{ s}$$

and for a *semifat index* with the *worst* input, the QUBE would be

$$1001 \times 10 \text{ ms} + 1000 \times 0.01 \text{ ms} + 1000 \times 0.1 \text{ ms} = 10 \text{ s}$$

It was certainly worthwhile checking against the worst case; a semifat index would have been totally unacceptable. The solution must be a *fat index*, with a QUBE (*worst* input) of

$$1 \times 10 \text{ ms} + 1000 \times 0.01 \text{ ms} + 1000 \times 0.1 \text{ ms} = 0.1 \text{ s}$$

WORST CASE SQL 7.2

```
SELECT   CNO, FNAME, ADDRESS, PHONE
FROM     CUST
WHERE    LNAME = 'SMITH'   FF = 1%
         AND
         CITY = 'LONDON'   FF = 10%
ORDER BY FNAME
```

A fat index was duly created:

```
CREATE INDEX Cust_X4 ON Cust (LNAME, CITY, CNO, FNAME,
   ADDRESS, PHONE)
```

and the execution plan confirmed the access path to be index only:

```
Execution Plan
----------------------------------
 0      SELECT STATEMENT Optimizer=CHOOSE (Cost=8
        Card=5 Bytes=545)
 1    0 SORT (ORDER BY) (Cost=8 Card=5 Bytes=545)
 2    1  INDEX (RANGE SCAN) OF 'CUST_X4' (NON-UNIQUE)
         (Cost=4 Card=5 Bytes = 545)
```

SQL Server Example

SQL Server 2000 reports similar information about the SQL call:

Physical reads	2
Read-ahead reads	50,160
Logical reads	50,002
Elapsed time	34,057 (ms)
CPU time	1,933 (ms)

This SELECT takes 34 s and uses 1.9 s of CPU time. The number of page requests (logical reads) is 50,002, and $50,160 + 2 = 50,162$ pages are read from disk.

The SQL call is the same one used in the Oracle example; likewise the index used, CUST_X1 (LNAME, FNAME). However, the filter factors represent the worst input (1% for LNAME, 10% for CITY—result 1000 rows). Furthermore, the table rows are stored in the clustered index (CNO).

With the worst input (FF for LNAME 1%), the optimizer chose a full table scan (a good choice if the access path costs can be estimated every time—10,000 random reads would have taken much longer) when the only relevant index was (LNAME, FNAME).

The execution plan shows that the whole clustered index (CNO) is scanned (no Index Seek), and the result rows are sorted.

```
SHOWPLAN
------------------------------------------------------
|--Sort(ORDER  BY([Cust].[Fname]ASC))
  |--Clustered Index
     Scan(OBJECT:([Invoices].[dbo].[Cust].[Cust_PK]),
     WHERE:( [Cust].[LNAME] = 'Adams'
      AND [CUST].[CITY]='BigCity').
```

This SELECT is certainly not a victim (ET/PR = 34 s/50,002 = 0.7 ms, which is much less than 10 ms); it appears to be a promising culprit (PR/F = 50,002/1000 = 50, which is considerably greater than 5).

The size of the CUST table is 400 MB (about 50,000 8K pages). The speed of the sequential read would therefore be 12 MB/s (400 MB in 34 s). The small server had only one slow disk drive; no striping.

With a fat index, the worst input performance improvement was almost as expected:

Physical reads	3
Read-ahead reads	175
Logical reads	176
Elapsed time	244 (ms)—down from 34 s
CPU time	40 (ms)—down from 1.9 s

This SELECT would still be reported as a slow one (ET > 200 ms), and it is certainly not a victim (ET/PR = 244 ms/176 = 1.4 ms, which is much less than 10 ms), but neither is it a promising culprit (PR/F = 176/1000 = 0.2, which is much less than 5).

The worst input QUBE for the fat index is the same as in the Oracle example:

$1 \times 10 \text{ ms} + 1000 \times 0.01 \text{ ms} + 1000 \times 0.1 \text{ ms} = 0.1 \text{ s}$: measured 0.2 s

Note: All the measurements above were made with a cold system (no pages in the database pool or the disk cache). Because of the warm-up effects, the service times, both CPU and I/O, are longer than they would be in a normal production environment. On the other hand, there would be more contention, queuing time, in a production environment.

It might be useful to see how an access path using a nonfat index is reported by SQL Server:

```
SHOWPLAN
------------------------------
|--Filter(WHERE:( [CUST].[CITY]= 'Sunderland'))
     |--Bookmark Lookup(BOOKMARK:([Bmk1000]),
        OBJECT([Invoices].[dbo].[Cust]))
         |--Index
            Seek(OBJECT([Invoices].[dbo].[Cust_X1]),
            SEEK: ([Cust].[Lname]='Lname265')
               ORDERED FORWARD)
```

The SELECT and the index used are the same as we had before the index improvement, although the input corresponds to average filter factors. The table Cust is stored in the clustered index (CNO).

The execution plan shows that the optimizer scans a slice, defined by predicate Lname = 'Lname265' (Index Seek), of CUST_X1 index (LNAME, FNAME), maintaining the order of the index rows (ORDERED) to avoid a sort. For every index row in the slice, it checks the corresponding table row in table Cust (Bookmark Lookup) and accepts only rows that have the right value in column City (Filter).

Conclusion

Call-level exception monitoring, with the basic information assumed above, reveals many index problems with minimal effort. However, due to many false alarms, it may take a long time to find all index problems that cause significant response time problems, much longer than with LRT-level exception monitoring.

Identifying SQL calls that are likely to be promising culprits would become easier if the reports were to show more information than just the four values shown in the examples. The number of result rows and the number of random reads to tables would be most valuable. Ideally, one should have *all* the figures we have shown in the bubble charts.

DBMS-SPECIFIC MONITORING ISSUES

From the index design point of view, the most significant differences between environments (the DBMS and the operating system) today are those involved in performance monitoring. Many other limitations can be overcome; for example, if the maximum index key length is too short, a table can be created, maintained by triggers, to be used in its place. It is, of course, impossible to measure the components of the local response time if the system does not provide appropriate trace records, or provides them with an inadequate degree of accuracy.

If it is not possible to produce the spike reports, bubble charts or SQL call exception reports described in this chapter, index design must be based mainly on predictions, quick estimates like the QUBE, or those made by the optimizer. The spikes can be detected by timing the transactions and SQL calls, or by users' complaints. It may then be possible to trace the program to determine the elapsed time of each SQL call and the number of page requests. The slow SQL calls should then be EXPLAINed and analyzed accordingly.

A QUBE for such a call, using the access path reported by EXPLAIN, may be helpful in determining the main causes for the long response time.

In his excellent Oracle tuning guide (2), Mark Gurry shows the SQL*Plus commands, which reveal the SQL statements that will be slow—*according to the optimizer estimates:*

Identify Bad SQL

> *The SQL statements in this section demonstrate how to identify SQL statements that have an expected response time of more than 10 seconds. The assumption*

has been made that 300 disk I/Os can be performed per second, and that 4,000 buffer page requests can be performed per second. These times are typical to medium to high-end machines.

Thus, the recommended queries report the SQL statements that will cause more than 3,000 disk I/Os or access more than 40,000 database pages according to the optimizer estimate.

(Ref. 2, p. 67)

Mark Gurry (2, p. 68) continues, addressing *monitoring*:

Oracle 8i and later has a great feature that stores information on long-running queries currently active in the V$SESSION_LONGOTS

. . .

Perhaps the DBA should phone the HROA user and question the statement, maybe even cancelling it if it is going to run much longer.

Online monitoring may be an adequate approach in data warehouse environments and with small and noncritical applications. However, in operational systems with many users, single programs that become slower and slower may one day bring the whole database server to a standstill. An exception report covering all interactive transactions gives an early warning of all kinds of performance problems. Monitoring the response times of individual transactions is proactive tuning, minimizing the risk of a performance crisis.

As many DBMS suppliers are now investing heavily to make their systems more self-tuning, it seems likely that the monitoring facilities will be improved in many products over the next few years.

Spike Report

Spike reports like the ones shown in this chapter are being produced in two ways in DB2 for z/OS installations.

The starting point is the DB2 Accounting Trace records, classes 1, 2, 3, 7, and 8. Using DB2PM (DB2 Performance Monitor), a filter command selects the records relating to interactive transactions whose local response time or SQL time exceeds alarm limits set by the installation. The profiles of these transactions are produced automatically, but the production of a concise report, one line per spike, requires a little program code and a bit of tailoring according to the database buffer pool setup (which objects are associated with which pools). This may be done by either writing a REXX program to read the standard report records, or by saving the trace data of the spikes in three tables (BASE, PACKAGE, POOL), which are then read by a program to produce the spike report. The second solution is a better option for the long term because it is less sensitive to changes in the trace records. Once created, the spike report rows may be saved in a history table to enable future comparisons to be made easily. Even with continuous exception reporting, disk space is negligible.

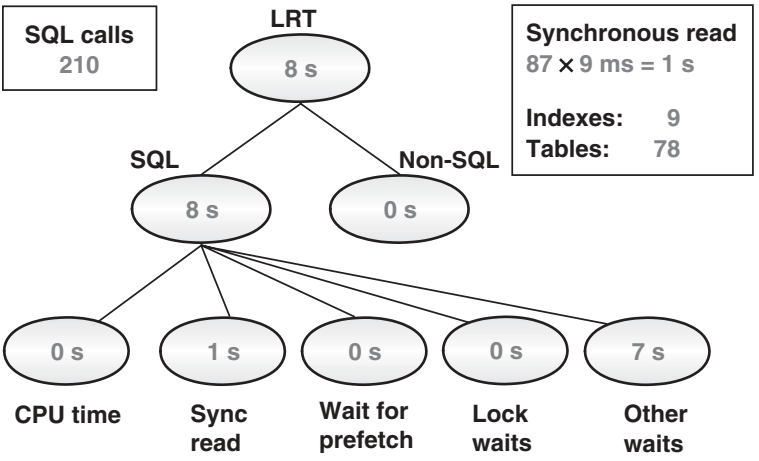


Figure 7.12 An atypical spike.

EXERCISES

7.1. How would you classify the spike shown in Figure 7.12: a promising culprit, an unpromising culprit, or a victim?

7.2. What can be done to reduce the local response time?

Chapter 8

Indexing for Table Joins

- Introduction to, and the terms used by, the most common join techniques
- Simple table join examples to illustrate the processing that takes place in a nested-loop join—the most common join method
- Comparison of a two-table join SELECT with a program-implemented two-table access
- Terms involved in the join process together with their significance
- Local and join predicates
- Table access order
- Inner and outer tables
- Case study to consider the importance of the table access order on the index design process, using the QUBE to illustrate the relative performance of three different program implementations
- Merge scan joins and hash joins
- Comparisons with nested-loop joins
- Why these join techniques have become more widely used with current hardware
- Adapting BQ to table joins to formulate the basic join question, BJQ
- Index design considerations when joining more than two tables together, and when using subqueries and unions
- Why joins often perform poorly
- Table design issues that should be considered with respect to joins
- Downward and upward denormalization
- Cost of denormalization
- NLJ and MS/HJ compared to denormalization
- Unconscious table design

INTRODUCTION

All the examples we have encountered so far have been single-table SELECTs. It is now time to consider SELECT statements that access several tables, *table*

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

joins. It is more difficult to design good indexes for a join than for a single-table SELECT because the join method and the table access order have a great impact on index requirements. The methods discussed in previous chapters, however, are still applicable to joins but with additional considerations.

In a join SELECT there are two kinds of predicates, *local predicates* that access a single table and *join predicates* that define the connection between tables. Most optimizers choose the join method and table access order by estimating the local response time of numerous alternatives. The most common join method is the *nested-loop join* (SQL Server: *loop join*), although *merge scan join* (Oracle:*sort-merge join*; SQL Server: *merge join*) and *hash join* are becoming more popular; this is because sequential read is much faster than it used to be. In a nested-loop join, the DBMS first finds a row in the *outer table* that satisfies the local predicates referring to that table. Then it looks for the related rows in the next table, the *inner table*, and checks which of these satisfy their local predicates—and so on.

A two-table join could be replaced by two single-table cursors and two nested loops coded in the program. It is then the programmer who makes the decision about the table access order—which table is accessed in the outer loop. When a join is used, the optimizer makes that decision, although this may be overridden by hints or other tricks.

Join predicates are most often based on *primary key = foreign key*. Assuming the correct foreign key indexes (starting with the foreign key columns) have been created and the result table is not exceptionally large, a nested loop is likely to be the fastest method, at least if all local predicates refer to one table.

If a nested-loop join is not appropriate, a merge scan or hash join may be faster; with the former, one or more tables are sorted into a consistent order, if necessary (after local predicates have been applied), and the qualifying rows in the tables or work files are merged; the hash join is basically a merge scan with hashing instead of sorting—more details later. With these methods, no table page is accessed more than once. Oracle, SQL Server, and DB2 for LUW tend to choose hash join instead of merge scan.

DB2 for z/OS does not use a hash join; in some less common situations a *hybrid join*— essentially a nested loop with list prefetch—may be preferred.

Note

- In this chapter we will initially concentrate on the nested-loop join and then compare this with the merge scan/hash join.
- Where data is required from more than one table, the column names in this chapter are prefixed with the table names to show from which table or index the columns are to be found.

TWO SIMPLE JOINS

We will first analyze two simple join examples to show the processes involved in a nested-loop join and how we can calculate the cost of the joins.

Example 8.1: Customer Outer Table

SQL 8.1

```
DECLARE CURSOR81 CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       CUST.CNO = :CNO
            AND
            CUST.CNO = INVOICE.CNO
WE WANT 20 ROWS PLEASE
```

This is a very simple example of accessing columns from two different tables (Fig. 8.1). The starting point is the primary key via the primary index CNO, and so only a single row from the customer table will be accessed to obtain two columns.

The join predicate shows that the customer number is also to be used to access the Invoice table (the foreign key in this case) using the foreign index CNO, where two further columns are obtained. Each customer will have on average 20 invoices (1 million customer rows and 20 million invoice rows), but, of course, some will have more.

We are assuming the CNO index on the invoice table is *not* the clustering index.

The QUBE (based on a 20-row result set) would show a very respectable response time as follows:

```
Index CNO       TR = 1
Table CUST      TR = 1

Index CNO       TR = 1       TS = 20
Table INVOICE   TR = 20

Fetch 20 × 0.1 ms

LRT             TR = 23      TS = 20
                23 × 10 ms   20 × 0.01 ms
                230 ms + 0.2 ms + 2 ms = 232 ms
```

There is no question that the outer table of the join (i.e., the starting point) will be the customer table; the WHERE clause contains the primary key so the DBMS will know that only a single row will be required, and therefore a single scan of the inner table. There are *no* local predicates for the invoice table and so this table must be the inner table.

This same process would also take place where two *separate* SELECT statements were used instead of a single join statement. The first would use the primary key on the customer table; the second would use a cursor to use this

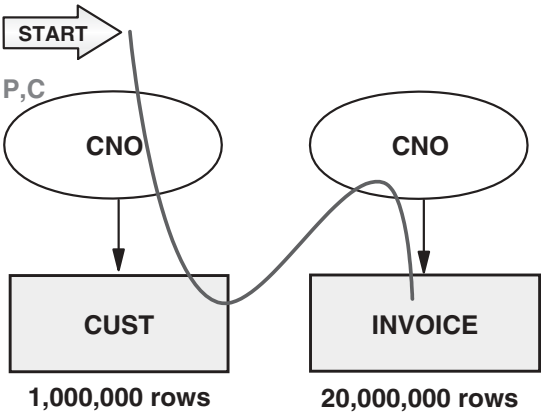


Figure 8.1 Customer outer table.

same customer number in the second SELECT statement, 20 rows being fetched (on average). Likewise the QUBE would be the same, (except for the number of FETCH calls).

Example 8.2: Invoice Outer Table

SQL 8.2

```
DECLARE CURSOR82 CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       IDATE = :IDATE
            AND
            CUST.CNO = INVOICE.CNO
WE WANT 20 ROWS PLEASE
```

This is another simple example of accessing columns from two different tables (Fig. 8.2), but this time the invoice date index IDATE will be the starting point as there is no customer information provided in the query. Assuming only the first 20 invoices are required, these will be accessed both to pick up the two columns required and to obtain the customer number of each invoice for the access to the customer table, now the inner table, via the CNO index. The access to the customer table will, as before, be via the primary index, one customer row for each invoice.

It is quite possible that each invoice will be for a different customer and so there will be quite a large number of random touches. The response time for this

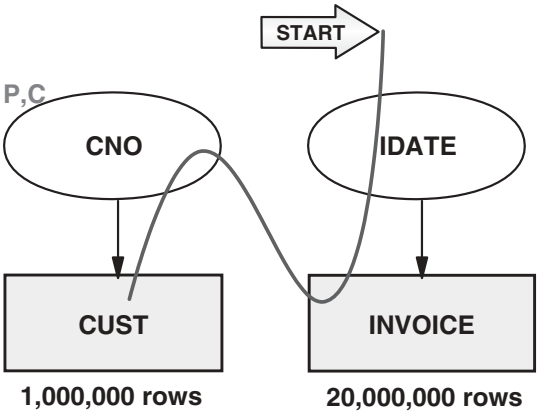


Figure 8.2 Invoice outer table.

query will therefore be considerably longer than the previous one as the QUBE will show:

```
Index IDATE      TR = 1        TS = 19
Table INVOICE    TR = 20

Index CNO        TR = 20
Table CUST       TR = 20

Fetch 20 × 0.1 ms

LRT              TR = 61       TS = 19
                 61 × 10 ms    19 × 0.01 ms
                 610 ms + 0.2 ms + 2 ms = 612 ms
```

As before, there is no question as to which table will be the outer table, the starting point. There are no local predicates for the customer table and so this table will be the inner table.

Again the same process would apply where two separate SELECT statements were used instead of a single join statement. The first would be a cursor for the invoice access; the second would be a SELECT to provide the customer information using the customer number provided by the cursor. Likewise, the QUBE would be the same (except for the number of FETCH calls).

IMPACT OF TABLE ACCESS ORDER ON INDEX DESIGN

In the two previous examples the local predicates, together with the indexes available, leave no choice with regard to the starting point. Unfortunately, this isn't always the case as many people have discovered to their cost. We will now consider one of these less straightforward cases by means of a case study.

In the first part of this case study we will focus entirely on nested-loop joins. Once we have become familiar with the issues involved, and in particular the potential problems associated with this join method, we will continue the case study by focusing on the merge scan join and the hash join. Finally we will compare the advantages and disadvantages of the two techniques from the index design point of view.

Case Study

A global company has 800,000 domestic customers and 200,000 customers in 100 other countries. Basic data for all its 1,000,000 customers are stored in table CUST. The operational invoice table INVOICE contains 20,000,000 rows. The customers, both domestic and nondomestic, each have an average of 20 invoices.

A new query transaction is about to be implemented to find, in descending order, high value invoices *for a specified foreign country*, based on the IEUR column in the invoice table—we will call these *large* invoices. The user input consists of the high value limit for the invoice total, together with the foreign country code.

The query can be implemented either as a two-table join SELECT or by using one of the many program alternatives that implement the join logic with two single-table SELECTs. The program can build a *nested-loop structure*; program A, for instance, may start with table CUST (CUST being the outer table); program B may use the INVOICE table as the outer table. We have this freedom of choice because the operation, in join terminology, is an *inner join*; we are only interested in customers that have at least one large invoice. If the requirement had been to obtain customer information, together with their invoices *if any*, this would have been an *outer join*; the program would then, of course, *have to* access table CUST first.

In the following analysis, program A accesses table CUST first (CUST is the outer table), while program B starts with table INVOICE (INVOICE is the outer table). Program C, with a two-table join SELECT, is neutral; if this alternative is chosen, we are leaving the optimizer to choose both the join method and the table access order. The table level decisions, which index to use and so forth, are, of course, made by the optimizer in all three cases.

Before we analyze the three programs, take a moment to consider which program *you* would choose. Some programmers would intuitively choose program A because it seems more natural to start with table CUST; one customer has many invoices. On the other hand, we have to consider the requirement to provide the high value invoices *in descending order*; to do this it is possible to include ORDER BY IEUR DESC in program B. The DBMS then takes care of the sort if indeed it is not rendered unnecessary by the use of an index on table INVOICE. Program B seems a bit more convenient.

Program A: Outer Table CUST

SQL 8.3

```
DECLARE CURSORC CURSOR FOR          Program A
SELECT    CNO, CNAME, CTYPE
FROM      CUST
WHERE     CCTRY = :CCTRY

DECLARE CURSORI CURSOR FOR
SELECT   INO, IEUR
FROM     INVOICE
WHERE    IEUR > :IEUR
         AND
         CNO = :CNO

OPEN CURSORC
     FETCH CURSORC         while CCTRY = :CCTRY
     OPEN CURSORI
           FETCH CURSORI   while IEUR > :IEUR
     CLOSE CURSORI
CLOSE CURSORC

 SORT THE RESULT ROWS BY IEUR DESC
```

Program B: Outer Table INVOICE

SQL 8.4

```
DECLARE CURSORI CURSOR FOR          Program B
SELECT   CNO, INO, IEUR
FROM     INVOICE
WHERE    IEUR > :IEUR
ORDER BY IEUR DESC

OPEN CURSORI
     DO
         FETCH CURSORI    while IEUR > :IEUR
         SELECT      CNAME, CTYPE
         FROM        CUST
         WHERE       CNO = :CNO
                     AND
                     CCTRY = :CCTRY
     DO END
CLOSE CURSORI
```

Program C: Optimizer Chooses Outer Table

SQL 8.5

```
                                                    Program C
DECLARE CURSORJ CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       IEUR > :IEUR
            AND
            CCTRY = :CCTRY
            AND
            CUST.CNO = INVOICE.CNO

ORDER BY    IEUR DESC

OPEN CURSORJ
  FETCH CURSORJ  while IEUR > :IEUR and CCTRY = :CCTRY
CLOSE CURSORJ
```

If only one of the tables had local predicates, starting with that table would seem like a good idea from the performance point of view, as we saw in SQL 8.1 and 8.2. As both tables now have local predicates, it is not obvious which program would be faster—unless we know how the tables are indexed.

For the purpose of this case study let us assume that, apart from the primary and foreign key indexes, the only additional index is (CCTRY) on table CUST, as shown in Figure 8.3. CCTRY is the *code* representing the country. Now most readers would *probably* vote for program A; we appear to have an appropriate index on the customer table but not on the invoice table.

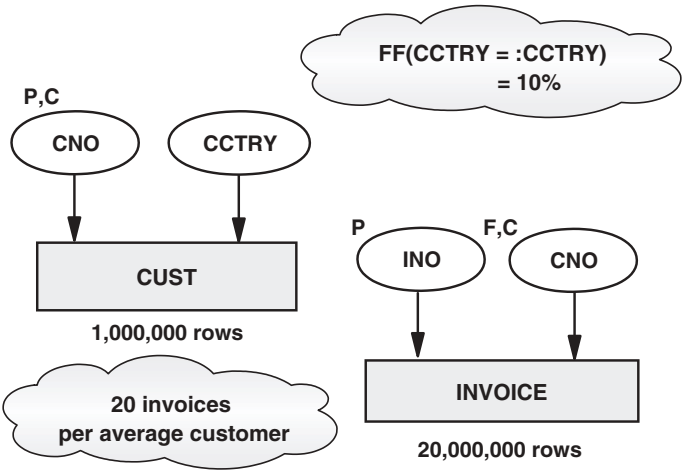


Figure 8.3 Current indexes for the case study.

As usual, when creating a new program, we should check whether the current indexes are adequate, even with the worst input. If they aren't, we should design the best affordable index, probably starting by designing and evaluating the *ideal* index.

To form an understanding of the actual difference in performance between these three programs, let us compare the estimates, using the QUBE, to see how they would perform with *the current indexes* shown in Figure 8.3.

We will make the estimates using the highest filter factors, assuming that these represent the worst case.

```
FF (CCTRY = :CCTRY)      max 10%.
FF (IEUR > :IEUR)        max 0.1%
```

Current Indexes

The current indexes are shown in Figure 8.3. This diagram is shown again for each of the three programs, together with the appropriate access path involved and the touches required to each index and table.

Program A: Outer Table CUST

Step 1: Access Table CUST via Nonclustering Index CCTRY The DBMS must find all the customers from the specified country. It can do this either by a full table scan or by using the index CCTRY (Fig. 8.4). With a high filter factor for the predicate CCTRY = :CCTRY, a full table scan will be faster; with a very low filter factor, an index scan will be faster. If the optimizer selects the access path *only once for many executions*, it may choose an index scan based on a filter factor of 1% (one divided by the number of distinct values of the column CCTRY). If the optimizer selects the access path *every time* based on actual user input, it will undoubtedly choose a table scan as long as it is aware of the true filter factor.

SQL 8.6

```
DECLARE CURSORC CURSOR FOR                          Program A
SELECT      CNO, CNAME, CTYPE
FROM        CUST
WHERE       CCTRY = :CCTRY

DECLARE CURSORI CURSOR FOR
SELECT      INO, IEUR
FROM        INVOICE
WHERE       IEUR > :IEUR
            AND
            CNO = :CNO

OPEN CURSORC
      FETCH CURSORC          while  CCTRY = :CCTRY
      OPEN CURSORI
            FETCH CURSORI    while  IEUR > :IEUR
      CLOSE CURSORI
CLOSE CURSORC

 SORT THE RESULT ROWS BY IEUR DESC
```

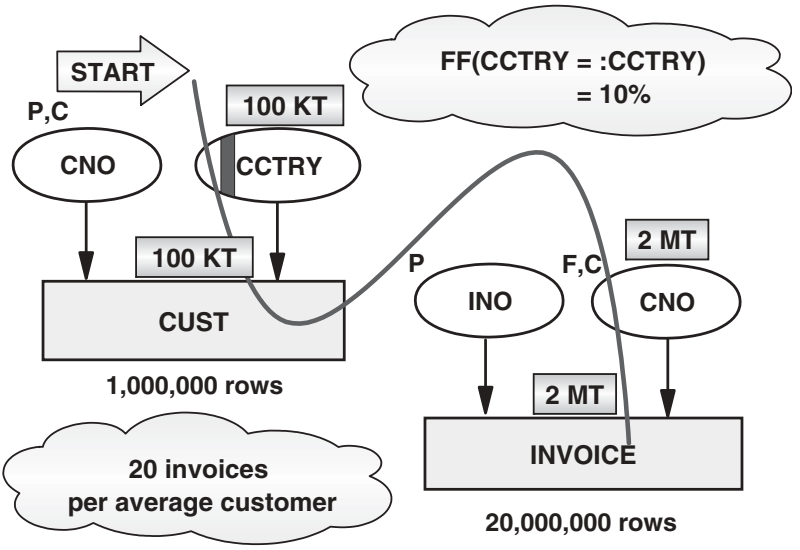


Figure 8.4 Program A with current indexes.

```
Index CCTRY   TR = 1            TS = 10% × 1,000,000
Table CUST    TR = 100,000      TS = 0

Fetch 10% × 1,000,000 = 100,000 × 0.1 ms

LRT           TR = 100,001      TS = 100,000
              100,001 × 10 ms   100,000 × 0.01 ms
              1000 s + 1 s + 10 s = 1000 s
```

According to the QUBE, this step contributes 1000 s to the response time! The large number of TRs for the customer table occurs because the index CCTRY is *not* the clustering index.

So much for using the "obvious" access path. With a filter factor of 10%, 100,000 TRs (17 min) is clearly not an option. Even a filter factor of 1% would require 10,000 TRs (1.7 min). The alternative, a full table scan, would only take

$$1 \times 10 \text{ ms} + 1 \text{ m} \times 0.01 \text{ ms} = 10 \text{ s}$$

A filter factor of 0.1% with the index scan would require 1000 TRs, taking the same length of time as the table scan. This is, of course, because one TR, according to the QUBE, takes 1000 times longer than one TS. A full table scan is faster than a nonfat, nonclustered index scan for filter factors greater than 0.1%. We must be very careful, however, when making sweeping statements such as this, based solely on figures produced by the QUBE. It should also be understood that the CPU time used would be very much greater with the table scan. Nevertheless, it should be appreciated that *very low filter factors are required these days for nonfat, nonclustered index scans to be preferred to table scans*.

Together with the FETCH cost, the LRT for this step will be 20 s.

Step 2: Access Table INVOICE via Clustering Index CNO When a customer from the specified country is obtained from the customer table, all the invoices for that customer must be checked to find the large invoices. These index rows (on average, 20 per customer) are next to each other, as are the table rows because index CNO on table INVOICE is the clustering index. It is important to have consistent clustering in large related tables like CUST and INVOICE. It takes 21 index touches and 20 table touches to check all the invoices of an average customer, but only the first index touch and the first table touch are random. If 100,000 customers (worst input: 10% of customers) are processed, the touches will be as shown below. Note that the customer number values the program moves from FETCH CUSRSORC to OPEN CURSORI are not consecutive; each one causes one random index touch and one random table touch.

```
Index CNO       TR = 100,000      TS = 100,000 × 20
Table INVOICE   TR = 100,000      TS = 100,000 × 19

Fetch 100,000 × 0.1ms

LRT             TR = 200,000      TS = 3,900,000
                200,000 × 10 ms   3,900,000 × 0.01 ms
                2000 s + 39 s + 10 s = 2050 s
```

According to the QUBE, this step contributes *over twice* as much as the first step if an index scan had indeed been chosen; a table scan in step 1 would have been insignificant compared to the cost of the inner table processing.

Step 3: Sort the Result Rows With the worst input the result table consists of $0.1 \times 0.001 \times 20{,}000{,}000 = 2000$ large orders. This must be displayed in descending order by invoice total (IEUR). The sort will take place as an additional step at the end, perhaps using a temporary table. According to our estimates, the CPU time for the sort is about 0.01 ms per row; 0.02 s for 2000 rows. Such a small sort will probably not cause any disk I/O, so the elapsed time is roughly 0.02 s as well. The QUBE assumes that the sort time is absorbed in the FETCH cost because it is very small and so can be ignored.

Local Response Time With the worst input, the local response time with this access path is the sum of the three components:

$$20 \text{ s} + 2050 \text{ s} + 0 \text{ s} = 2070 \text{ s} = 35 \text{ min}$$

Over half an hour to wait for the first screen! Even if the users were to be told to take a lunch break after entering the required country input, the disk traffic burst, up to 200,000 random reads in half an hour, could cause noticeable drive queuing for other concurrent users.

With the current indexes as shown in Figure 8.5, the DBMS must scan the whole INVOICE table looking for large invoices. When one is found, it checks

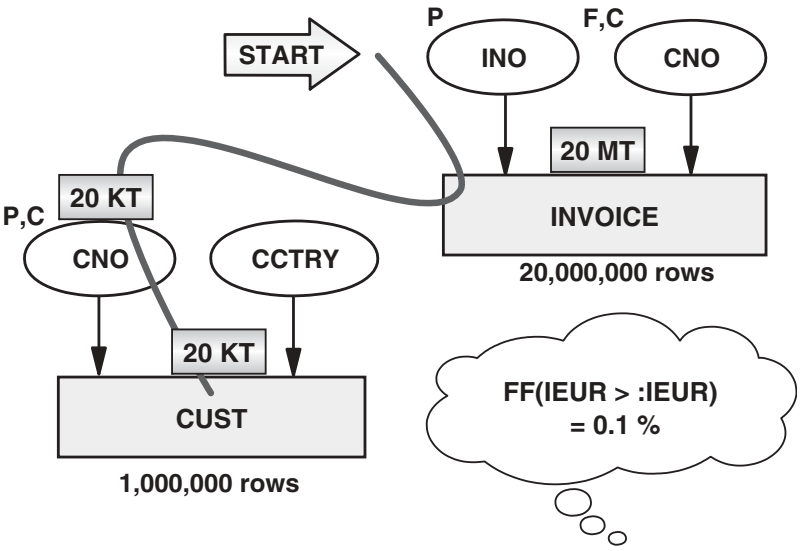


Figure 8.5 Program B with current indexes.

the country of the customer by reading the corresponding CUST row via the primary key index CNO.

Program B: Outer Table INVOICE

SQL 8.7

```
                                                        Program B
DECLARE CURSORI CURSOR FOR
SELECT     CNO, INO, IEUR
FROM       INVOICE
WHERE      IEUR > :IEUR
ORDER BY   IEUR DESC

OPEN CURSORI
      DO
           FETCH CURSORI        while  IEUR > :IEUR
           SELECT      CNAME, CTYPE
           FROM        CUST
           WHERE       CNO = :CNO
                       AND
                       CCTRY = :CCTRY
      DO END
CLOSE CURSORI
```

Step 1: Full INVOICE Table Scan The INVOICE table has 20,000,000 rows. Checking each of these takes 20,000,001 touches, including the touch for the end-of-file mark. Only the first touch is random.

```
Table INVOICE   TR = 1      TS = 20,000,000

Fetch 0.1% × 20,000,000 = 20,000 × 0.1 ms

LRT             TR = 1      TS = 20,000,000
                1 × 10 ms   20,000,000 × 0.01 ms
                10 ms + 200 s + 2 s = 200 s
```

Step 2: Read the CUST Row Corresponding to Each Large Invoice via Clustering Index CNO Using the assumed worst input filter factors, $0.001 \times 20{,}000{,}000 = 20{,}000$ of the invoices are large. Therefore the DBMS will read 20,000 CUST rows via the CNO index. Only 10% of these belong to the required country, and so most of the rows will be rejected, only 2000 being accepted.

```
Index CNO    TR = 20,000
Table CUST   TR = 20,000

Fetch 20,000 × 0.1 ms

LRT          TR = 40,000
             40,000 × 10 ms + 2 s = 400 s
```

Step 3: Sort the Result Rows The 20,000 rows fetched in step 1 from CURSORI must be sorted in descending order by invoice total, IEUR. Sorting 20,000 rows takes a negligible amount of time (roughly 200 ms), again assumed to be absorbed in the FETCH cost.

Local Response Time With the worst input, the local response time with this access path is the sum of the three components:

$$200 \text{ s} + 400 \text{ s} + 0 \text{ s} = 600 \text{ s} = 10 \text{ min}$$

Surprisingly, program B is considerably faster than program A, even with the current indexes—no index for IEUR! How is this possible? There are two reasons:

1. With current hardware, sequential read is really fast: A table with 20,000,000 rows may be scanned in about 3 min (based on the QUBE figures).
2. Because the highest filter factor for the predicate IEUR < :IEUR is only 0.1%, the number of random touches to the *inner* table is lower than with program A.

Program C: Outer Table Chosen by the Optimizer

SQL 8.8

```
                                                        Program C
DECLARE CURSORJ CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       IEUR > :IEUR
            AND
            CCTRY = :CCTRY
            AND
            CUST.CNO =  INVOICE.CNO
ORDER BY    IEUR DESC

OPEN CURSORJ
 FETCH CURSORJ  while  IEUR > :IEUR and  CCTRY = :CCTRY
CLOSE CURSORJ
```

This is a neutral program. The optimizer estimates the filter factors for predicates IEUR > :IEUR and CCTRY = :CCTRY, as we saw in Chapter 3, and then chooses the table access order accordingly.

The predicate IEUR > :IEUR is not an easy one for the optimizer because it does not know the range of values the user will enter for the field that is associated with the host variable :IEUR. The optimizer may use a default value like 33 or 50%. Then, if it chooses a nested-loop join, it is likely to choose the same access path as for program A, namely the CUST outer table as shown in Figure 8.6. This would result in a worst input LRT of 35 min instead of 10 min.

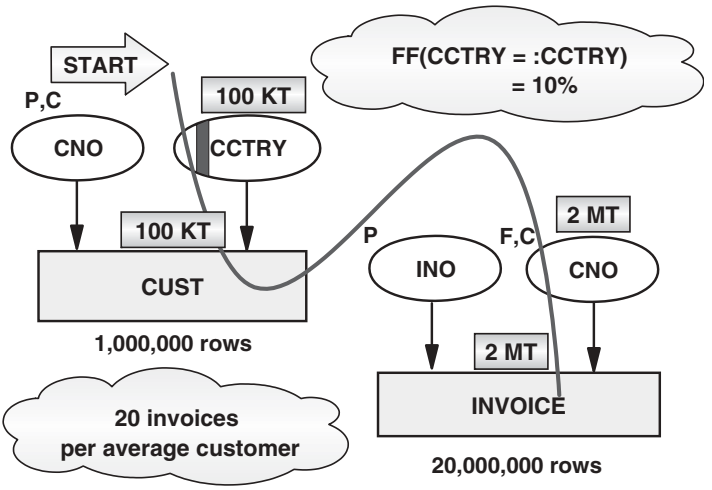


Figure 8.6 Program C with current indexes.

Apart from the choice of outer table, the only difference between the QUBEs for programs A and B compared to program C will be the number of FETCH calls:

Program A	$200{,}000 \times 0.1$ ms $= 20$ s
Program B	$40{,}000 \times 0.1$ ms $= 4$ s
Program C	2000×0.1 ms $= 0.2$ s

This benefit is actually very small in terms of the overall figures although it could, of course, be significant in other circumstances.

Conclusion: Current Indexes

It is clear that the current indexes are not adequate with the worst input. We should now consider the best affordable indexes, probably starting by designing and evaluating the *ideal* indexes.

Ideal Indexes

Program A: Outer Table CUST

Using the approach discussed in Chapter 4 for the design of ideal indexes, candidate A for CURSORC is (CCTRY, CNO, CNAME, CTYPE). It is a three-star index. Candidate A for CURSORI (CNO, IEUR DESC, INO) also has three stars. The CNO column, of course, comes from CURSORC; we have now a thin index slice based on CNO and IEUR. As both are ideal indexes, there is no need to consider candidate B. These indexes are shown in Figure 8.7. Strictly speaking, CURSORI does *not* require IEUR to be a *descending* key in the ideal index

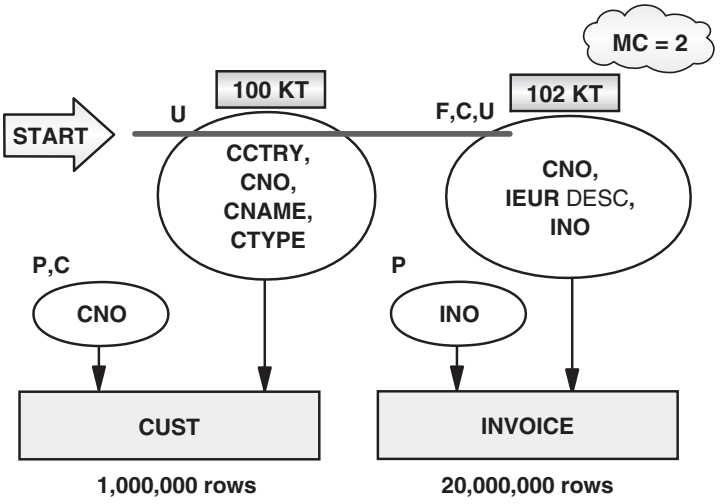


Figure 8.7 Program A with ideal indexes.

for program A; in order to simplify comparisons made with program B, we will nevertheless assume a descending key.

SQL 8.9

```
                                                        Program A
DECLARE CURSORC CURSOR FOR
SELECT      CNO, CNAME, CTYPE
FROM        CUST
WHERE       CCTRY = :CCTRY

DECLARE CURSORI CURSOR FOR
SELECT      INO, IEUR
FROM        INVOICE
WHERE       IEUR > :IEUR
            AND
            CNO = :CNO

OPEN CURSORC
      FETCH CURSORC             while  CCTRY = :CCTRY
      OPEN CURSORI
            FETCH CURSORI       while  IEUR > :IEUR
      CLOSE CURSORI
CLOSE CURSORC

 Sort the result rows by IEUR DESC
```

Step 1: Read the Slice (*CCTRY = :CCTRY*) from Index (CCTRY, CNO, CNAME, CTYPE) With a filter factor of 10% for predicate CCTRY = :CCTRY, the slice has 0.1 × 1,000,000 index rows. No table access is required as we are using a three-star index.

The two major costs in this step are now the index scan, which is as thin as it can be, and, much more importantly, the FETCH calls; the huge number of calls is taking 10 s!

```
Index CCTRY, CNO, CNAME, CTYPE   TR = 1      TS = 10% × 1,000,000

Fetch 10% × 1,000,000 = 100,000 × 0.1 ms

LRT                              TR = 1      TS = 100,000
                                 1 × 10 ms   100,000 × 0.01 ms
                                 10 ms + 1 s + 10 s = 11 s
```

Step 2: For Each CNO, Read All the Large Invoices Using Index (CNO, IEUR DESC, INO) Now the index slice for each CNO is defined by two columns, CNO and IEUR. When a customer doesn't have any large invoices, the DBMS needs to do only one random touch to index (CNO, IEUR DESC, INO). If the first index row relates to a large invoice, the DBMS must read the next index row, and so on. Therefore, the number of random touches is 100,000

(the number of customers with a specified CCTRY value) and the number of sequential touches is 2000 (the number of large invoices for that country).

```
Index CNO, IEUR DESC, INO  TR = 100,000     TS = 2000

Fetch 100,000 × 0.1 ms

LRT                          TR = 100,000     TS = 2000
                             100,000 × 10 ms  2000 × 0.01 ms
                             1000 s + 0.02 s + 10 s = 1000 s
```

The FETCH calls take a further 10 s, but this now becomes insignificant compared to the TRs against the index.

Local Response Time Ignoring the sort cost as before

$$11 \text{ s} + 1000 \text{ s} = 1011 \text{ s} = 17 \text{ min}$$

It is extremely important to appreciate that in this case the benefit of using an ideal index is minimized because of the huge number of TRs to the inner table index.

Program B: Outer Table INVOICE

The ideal indexes for program B are *not the same* as the ideal indexes for program A because the information used to access the two tables is now different.

Candidate A for CURSORI is now (IEUR DESC, INO, CNO). Candidate A for the customer table is (CCTRY, CNO, CNAME, CTYPE). The CNO column comes from CURSORI. They are both three-star indexes and are shown in Figure 8.8; there is no need to consider candidate B.

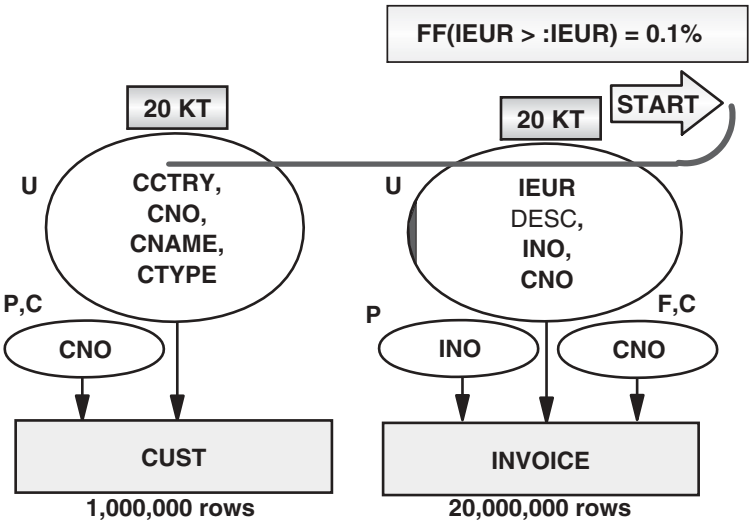


Figure 8.8 Program B with ideal indexes.

SQL 8.10

```
                                                        Program B
DECLARE CURSORI CURSOR FOR
SELECT      CNO, INO, IEUR
FROM        INVOICE
WHERE       IEUR > :IEUR
ORDER BY    IEUR DESC

OPEN CURSORI
      DO
            FETCH CURSORI          while  IEUR > :IEUR
            SELECT      CNAME, CTYPE
            FROM        CUST
            WHERE       CNO = :CNO
                        AND
                        CCTRY = :CCTRY
      DO END
CLOSE CURSORI
```

Step 1: Read the Slice (IEUR > :IEUR) from Index (IEUR DESC, INO, CNO) The index slice contains $0.001 \times 20{,}000{,}000 = 20{,}000$ index rows. Thus, the DBMS must do one random touch and 20,000 sequential touches. The major cost in this step is now the FETCH processing; not as large as in program A but still 2 s. The index provides the result set in the desired sequence without the need for a sort.

```
Index IEUR DESC, INO, CNO   TR = 1      TS = 20,000

Fetch 0.1% × 20,000,000 = 20,000 × 0.1 ms

LRT                         TR = 1      TS = 20,000
                            1 × 10 ms   20,000 × 0.01 ms
                            10 ms + 0.2 s + 2 s = 2 s
```

Step 2: Read One Index Row (CCTRY, CNO, CNAME, CTYPE) for Each Large Invoice As the required customer numbers are *not* consecutive, the DBMS must do 20,000 random touches. Only 10% of these will be for the appropriate country.

```
Index CCTRY, CNO, CNAME, CTYPE   TR = 20,000

Fetch 20,000 × 0.1 ms

LRT                              TR = 20,000
                                 20,000 × 10 ms
                                 200 s + 2 s = 202 s
```

Local Response Time

$$2 \text{ s} + 202 \text{ s} = 3.5 \text{ min}$$

This is a huge reduction from the 17-min estimate for program A but, unfortunately, the message is the same as the one we emphasized in program A—the benefit of using an ideal index is minimized because of the huge number of TRs to the inner table index.

Program C: Outer Table Chosen by the Optimizer

With the ideal indexes, the estimates we derived for programs A and B show that if a nested-loop join is chosen by the optimizer, the INVOICE table should be the outer table. Therefore, the required ideal indexes should be those shown in Figure 8.8. The response time will be a little less than that for program B because of the reduction in FETCH calls, although the reduction, as we have seen, is minimal.

SQL Joins Compared to Program Joins It is often said that a join is more efficient than a program with multiple cursors. Is this true?

The critical factor is the access path chosen, namely the *join method* and the *table access order*. Sometimes one might choose a better access path than the optimizer; sometimes vice versa. Ultimately, the access paths should be identical, as with programs B and C above. The number of disk I/Os will then be the same; the only difference will be the number of SQL calls. Using a join SELECT is better in this respect as the figures shown above clearly indicate.

Because of the high number of random I/Os in programs A and B, the CPU time is small compared to the overall elapsed time. There are cases, however, in which the CPU time is the *largest* component of the local response time; a program with a join cursor may then be noticeably faster than a program with several single-table SELECTs.

Conclusion: Ideal Indexes

Even with three-star indexes, although reducing the worst input local response time from 35 to 3.5 min, this is still far too high. The next step must be to consider changing the program so that it builds only one screen, say 20 result rows, per transaction. *Please note that we have so far confined our attention to the nested-loop join. Our conclusion so far may well be affected by what is yet to be discussed.*

Ideal Indexes with One Screen per Transaction Materialized

Program B+: Materialize One Screen per Transaction

It is quite easy to modify program B so that it only issues 20 FETCH calls. We simply need to

- Ensure that the access path does not need to perform a sort.

- Change the program so that it counts the number of FETCH calls issued.
- Save positioning information in an auxiliary table to enable the next transaction to continue from that point.

The SQL changes required for the *first* transaction are shown in SQL 8.11. Note the 20 rows please and the SAVE values, and that the invoice number has been added to the ORDER BY clause in CURSORI; this is necessary for positioning purposes.

When the next screen is requested, the program must start from the point defined by the saved values. Let us call these values IEURLAST and INOLAST. The SQL changes required for the subsequent transactions are shown in SQL 8.12. It is possible that there are two or more invoices with the same invoice total (column IEUR). Therefore, the program must first open a cursor, CURSORI1, for any undisplayed invoices with the same invoice value as IEURLAST. When all the rows for that value have been FETCHed, the first cursor is closed and a second cursor, CURSORI2, is opened for the next large invoices.

SQL 8.11

```
                        Program B+ First transaction
DECLARE CURSORI CURSOR FOR
SELECT    CNO, INO, IEUR
FROM      INVOICE
WHERE     IEUR > :IEUR
ORDER BY  IEUR DESC, INO
WE WANT 20 ROWS PLEASE

OPEN CURSORI
     DO
            FETCH CURSORI           max 20 times
            SELECT       CNAME, CTYPE
            FROM         CUST
            WHERE        CNO = :CNO
                         AND
                         CCTRY = :CCTRY
     DO END
CLOSE CURSORI

SAVE / INSERT IEUR, INO  the values of the last line
displayed to the user
```

How long does it take now to build one screen with the ideal indexes? Let us again assume the largest filter factors, 10 and 0.1% using SQL 8.12 and Figure 8.9.

Step 1: Find 20 Large Invoices from Index (IEUR DESC, INO, CNO) It does not matter how many rows are found by CURSORI1 and by CURSORI2;

SQL 8.12

```
                  Program B+ Subsequent transactions
DECLARE  CURSORI1 CURSOR FOR
SELECT   CNO, INO, IEUR
FROM     INVOICE
WHERE    IEUR = :IEURLAST
         AND
         INO > :INOLAST
         AND
         IEUR > :IEUR
ORDER BY IEUR DESC, INO
WE WANT 20 ROWS PLEASE

DECLARE CURSORI2 CURSOR FOR
SELECT    CNO, INO, IEUR
FROM      INVOICE
WHERE     IEUR < :IEURLAST
          AND
          IEUR > :IEUR
ORDER BY  IEUR DESC, INO
WE WANT 20 ROWS PLEASE

READ IEURLAST, INOLAST

OPEN CURSORI1
     DO
           FETCH CURSORI1              max 20 times
           SELECT        CNAME, CTYPE
           FROM          CUST
           WHERE         CNO = :CNO
                         AND
                         CCTRY = :CCTRY
     DO END
CLOSE CURSORI1

OPEN CURSORI2
     DO
            FETCH CURSORI2    until 20 result rows
            SELECT      CNAME, CTYPE
            FROM        CUST
            WHERE       CNO = :CNO
                        AND
                        CCTRY = :CCTRY
     DO END
CLOSE CURSORI2

SAVE / INSERT IEUR, INO the values of the last line
displayed to the user
```

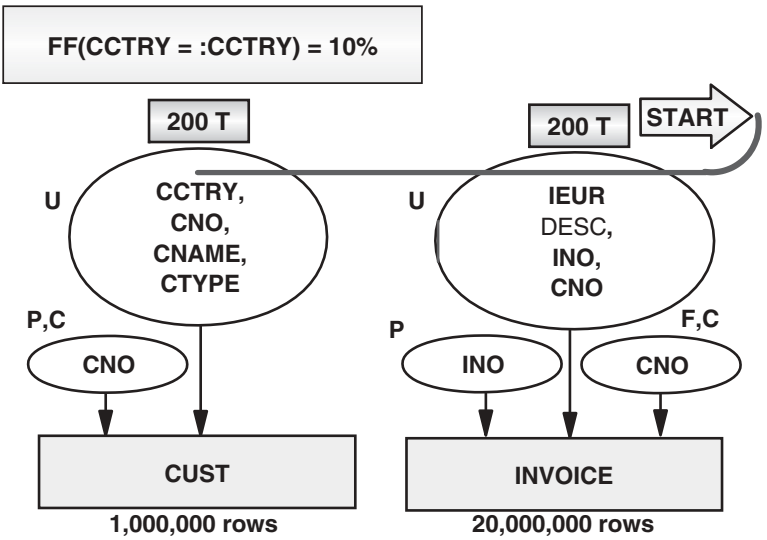


Figure 8.9 Program B+ with ideal indexes and one screen per transaction.

using either cursor, the program scans a 200-row slice of index (IEUR DESC, INO, CNO) because if FF(CCTRY = :CCTRY) is 10%, every tenth invoice, on average, is associated with the specified country. This step takes $20/0.1 = 200$ touches. Only the first of these is a random touch.

```
Index IEUR DESC, INO, CNO   TR = 1      TS = 199

Fetch 20 × 0.1 ms

LRT                         TR = 1      TS = 199
                            1 × 10 ms   199 × 0.01 ms
                            10 ms + 2 ms + 2 ms = 14 ms
```

Step 2: Read One Index Row (CCTRY, CNO, CNAME, CTYPE) for Each Large Invoice As the customer numbers are not consecutive, the DBMS must do 200 random touches.

```
Index CCTRY, CNO, CNAME, CTYPE   TR = 200

Fetch 200 × 0.1 ms

LRT                              TR = 200
                                 200 × 10 ms
                                 2 s + 20 ms = 2 s
```

Local Response Time

$$14 \text{ ms} + 2 \text{ s} = 2 \text{ s}$$

Conclusion: Ideal Indexes and One Screen per Transaction

With the highest filter factor values, the local response time is 2 s, possibly quite acceptable.

Ideal Indexes with One Screen per Transaction Materialized and FF Pitfall

The above estimate was made with the *largest* filter factors, 10 and 0.1%. Unfortunately, these are *not* the worst input now because this case fulfils the criteria for the *filter factor (FF) pitfall* (Fig. 8.10). The code of the country having only 19 large invoices is now the worst input: The whole index slice containing the large invoices must be scanned, and the CCTRY value must be checked for each large invoice. The filter factor for the predicate CCTRY = :CCTRY is then about 0.1%. How long would it take now to build the single-screen response with program B+?

Program B+: Materialize One Screen per Transaction

Step 1: Find All the Large Invoices from Index (IEUR DESC, INO, CNO)
To find all the large invoices for the single customer country, the DBMS must scan the whole index slice containing large invoices if the response fits on one screen. Let us make this worst-case assumption—one screen only.

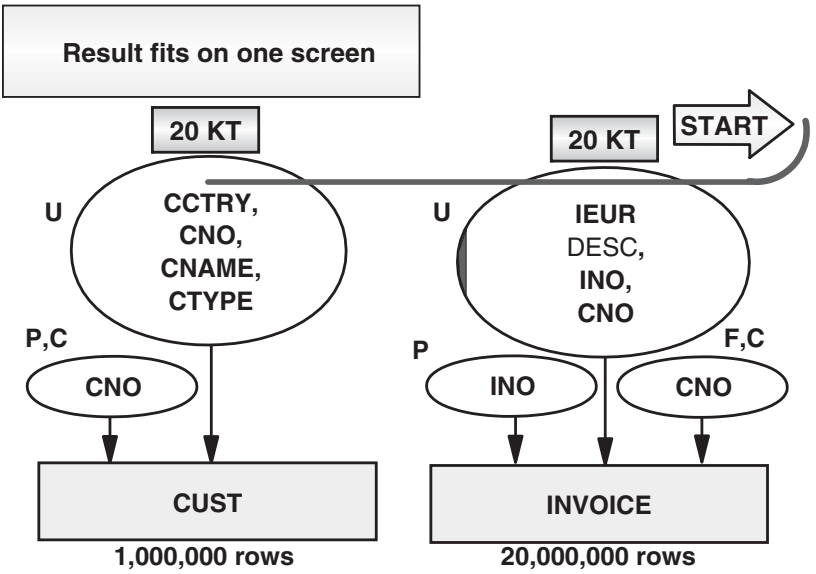


Figure 8.10 Program B+ with ideal indexes and filter factor pitfall.

```
Index IEUR DESC, INO, CNO   TR = 1      TS = 0.1% × 20,000,000

Fetch 20,000 × 0.1 ms

LRT                         TR = 1      TS = 20,000
                            1 × 10 ms   20,000 × 0.01 ms
                            10 ms + 0.2 s + 2 s = 2 s
```

We have now lost the advantage of limiting the result size; fortunately the damage is not too great.

Step 2: Read One Index Row (CCTRY, CNO, CNAME, CTYPE) for Each Large Invoice This step requires a huge number of random touches to the index because *every* large invoice has to be checked:

```
Index CCTRY, CNO, CNAME, CTYPE   TR = 20,000

Fetch 20,000 × 0.1 ms

LRT                              TR = 20,000
                                 20,000 × 10 ms
                                 = 200 s + 2 s = 202 s
```

Local Response Time

$$2 \text{ s} + 202 \text{ s} = 3.5 \text{ min}$$

Conclusion: Ideal Indexes and One Screen per Transaction with FF Pitfall

This is a bitter disappointment; even with the promising program B+ the local response time is 3.5 min when the response fits on *one screen*. This is in marked contrast to the response time for the first screen of a *multiscreen result*, which was only 2 s. Our previous message, about the benefit of using an ideal index being minimized because of the huge number of TRs to the inner table index, has come back to haunt us!

Program C+: Materialize One Screen per Transaction

The modifications needed to make program C materialize only one screen per transaction are the same as those made for program B+. The program shown in SQL 8.13 produces the first screen. The local response times will be the same as for program B+, good with large filter factors, very poor with the low filter factor worst case. The savings to be made by reducing the number of FETCH calls is of minimal benefit in the latter case.

BASIC JOIN QUESTION (BJQ)

This simple case study has shown that even the *ideal indexes* for the *best table access order* might result in unacceptable response times. The main issue is the

SQL 8.13

```
                                  Program C+ First transaction
DECLARE CURSORJ CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       IEUR > :IEUR
            AND
            CCTRY = :CCTRY
            AND
            CUST.CNO = INVOICE.CNO
ORDER BY    IEUR DESC,  INO
WE WANT 20 ROWS PLEASE

OPEN CURSORJ
      FETCH CURSORJ while IEUR > :IEUR and CCTRY =
                                  :CCTRY, max 20
CLOSE CURSORJ

SAVE / INSERT IEUR, INO         the values of the last
     line displayed to user
```

potentially huge number of random touches to an inner table (or its index); the index slices of customers who have at least one large invoice are not adjacent in the index (CCTRY, CNO, CNAME, CTYPE).

In Chapter 5, we introduced the Basic Question, BQ: Is there an existing or planned index that contains all the columns referenced by the WHERE clause? We observed that:

> *According to our experience, a significant proportion of index problems encountered after cutover would have been detected early by considering BQ. Applying BQ to single-table SELECTs is straightforward. A join on the other hand must be mentally broken down into several single-table cursors before the BQ can be applied. This is a much more complex process, one that we shall consider in great detail in Chapter 8.*

The reasoning behind BQ, of course, was to ensure, probably using index screening, that we only access a table when we *know* that the table rows are required. We can now extend this argument to include the inner table or its index in a nested-loop join—that is, we only access an inner table or index when we *know* that the table or index rows are required. Our *basic join question*, BJQ, therefore, becomes: *Is there an existing or planned index that contains all the local predicate columns?* This includes the local predicates for *all* the tables involved.

Here, the only way to eliminate the problem of excessive random touches to the inner table, and thereby its index, is to add redundant table columns in such a way that *one* table contains *all* the local predicates. Here, it suffices to add column CCTRY to table INVOICE, replacing predicate CCTRY = :CCTRY by INVOICE.CCTRY = :CCTRY. Now it is possible to create an index that contains all the local predicate columns, as shown in Figure 8.11.

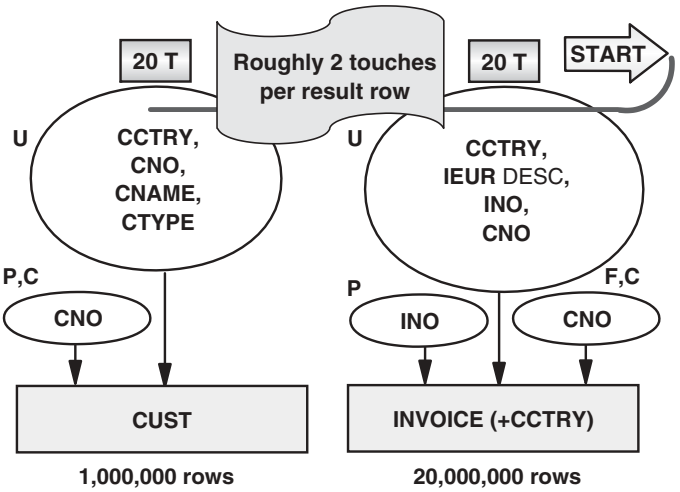


Figure 8.11 Program B+ with ideal indexes satisfying the basic join question.

Perhaps it would be a good idea to apply the BJQ early, when a join has been written or generated, and to implement the necessary denormalization, together with the triggers required to maintain the redundant table data (column CCTRY in table INVOICE).

Now our transaction is fast with any input because there are never more than 20 random touches to index (CCTRY, CNO, CNAME, CTYPE).

```
Index CCTRY, IEUR DESC, INO, CNO   TR = 1       TS = 19
Index CCTRY, CNO, CNAME, CTYPE     TR = 20

Fetch 20 × 0.1 ms

LRT                                TR = 21      TS = 19
                                   21 × 10 ms   19 × 0.01 ms
                                   = 210 ms + 0.2 ms + 2 ms = 0.2 s
```

Adding column CCTRY to the invoice table and the index may require $1.5 \times 40{,}000{,}000 \times 3$ bytes $= 180$ MB of disk space. If a customer moves to another country, probably a rare occurrence, one table row and one index row must be updated. For the average customer with 20 invoices, this means 20 random touches to the table and 40 random touches to the index, 60×10 ms $= 0.6$ s.

Please note that a further, more comprehensive, discussion on table design aspects of joins, will be found in the Table Design Considerations section at the end of this chapter.

Conclusion: Nested-Loop Join

A summary of the local response times for the various indexes and programs used in the case study so far is shown in Table 8.1.

Table 8.1 Summary of the Local Response Times—1

Type	Program A	Program B
Current indexes	35 min	10 min
Ideal indexes	17 min	3.5 min
Ideal indexes, ideal program (+), multi-screen result		2 s
Ideal indexes, ideal program (+), single-screen result (FF pitfall)		3.5 min
BJQ ideal indexes, ideal program (+), single-screen result (FF pitfall)		0.2 s

We must stress once again, that our discussion of join methods is, as yet, far from complete.

PREDICTING THE TABLE ACCESS ORDER

We have seen that with the nested-loop join method the table access order may have a dramatic impact on performance—and indexing. *The ideal indexes cannot be designed until an assumption has been made about the best table access order.*

The following rule of thumb, *in most cases*, predicts the best table access order:

The outer table is the one with the *lowest number of local rows*.

The *number of local rows* is the number of rows remaining after the local predicates have been applied using the maximum filter factors, as shown in Figure 8.12.

This rule of thumb *ignores* the following factors:

1. ***Order by*.** In our simple case study, the sort for ORDER BY IEUR DESC could only be avoided by making table INVOICE the outer table. This

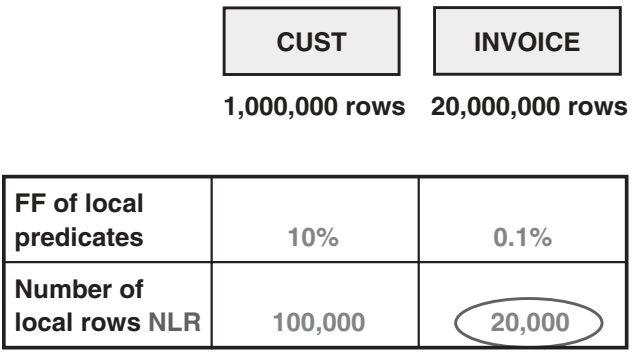


Figure 8.12 Predicting table access order in a nested-loop join.

table also happened to have the lower number of local rows, but obviously, this is not always the case.

2. ***Very small tables.*** Very small tables and their indexes may stay in the database buffer pool all day long—or at least no page would be read more than once during a join. Random touches to such tables and indexes take less than 0.1 ms, at least after the first touch per page; so when the table is *not* the outer table, a large number of random touches to them is not a problem.
3. ***Clustering ratio.*** The correlation between the order of the index rows and the table rows (100% for a clustering index after a table reorganization) may affect the choice of the best table access order, unless, of course, the indexes are fat.

The best cost-based optimizers take these factors into account when they choose the table access order. Therefore, they may find a better sequence than the one given by the rule of thumb based on the number of local rows.

The reader may wonder why the rule of thumb uses the maximum filter factors for the local predicates. This is simply because it is probably too time consuming to include the filter factor pitfall. This simplification certainly increases the risk of inappropriate recommendations where one alternative avoids a sort and the program does not fetch the whole result in one transaction. Nevertheless, indexing based on an assumed table access order may lead to adequate performance even if the table order access is not optimal.

When designing indexes for a join, where should we start? In theory, we should first create the best indexes for all table access order variations and then let the optimizer choose. Then we could remove any unused indexes and index columns. However, this would be too time consuming, particularly if there are more than two tables in the join—and the choice made by the optimizer will still depend on the input values. The index design tools that use the optimizer solve the first issue but not the second one. The quality of their proposals depends on the filter factor assumptions, which in turn depend on the input affecting the host variables in the predicates.

Fortunately, the rule of thumb leads to a satisfactory result in most cases—and often we do not even need to calculate the number of local rows because the best table access order is obvious.

The important issue is to keep the assumed join method and table access order in mind (or probably on the wall) when designing indexes for a join. Then the indexes are designed just as if the program had been coded with several single-table cursors instead of a join cursor. It is a common mistake to design indexes for a join without a consistent access path assumption with regard to the join method and table access order. It is not sufficient merely to have indexes for both the join predicates and the local predicates; actually this approach often results in a few redundant indexes. It is often essential that the inner table in a nested-loop join has a good fat index, starting with the join predicate column.

MERGE SCAN JOINS AND HASH JOINS

When we analyzed both program A and program B in our case study, we discovered that the benefits to be gained by using ideal indexes were limited because of the huge number of TRs to the inner table index. One of the main advantages of a merge scan join or a hash join is that this problem is avoided.

Merge Scan Join

Merge scan joins take place as follows:

- Table scans or index scans are carried out to find all the rows that satisfy the local predicates.
- Possible sorts, if these scans do not provide the results in the required sequence.
- A merge of the temporary tables created in the previous two steps.

A merge scan join is often faster than a nested-loop join in the following cases:

1. A join column index is not available. In this case, the inner table might have to be scanned several times with a nested-loop join. In real life it is unlikely that a join column index does not exist because most join predicates are based on primary key = foreign key as in our case study—CUST.CNO = INVOICE.CNO.
2. The result table is very large. In this case a nested-loop join might cause the same page to be accessed over and over again.
3. There are local predicates with low filter factors to more than one table. As we have seen, nested loop may then result in too many random touches to the inner table (or index).

Before using the case study to compare a nested-loop join with a merge scan join, we will first take a simple example to show what steps need to take place, and how we may calculate the LRT using the QUBE.

Example 8.3: Merge Scan Join

SQL 8.14

```
DECLARE CURSOR81 CURSOR FOR
SELECT   CNAME, CTYPE, INO, IEUR
FROM     CUST, INVOICE
WHERE    CUST.CTYPE = :CTYPE
         AND
         IDATE > :IDATE
         AND
         CUST.CNO = INVOICE.CNO
```

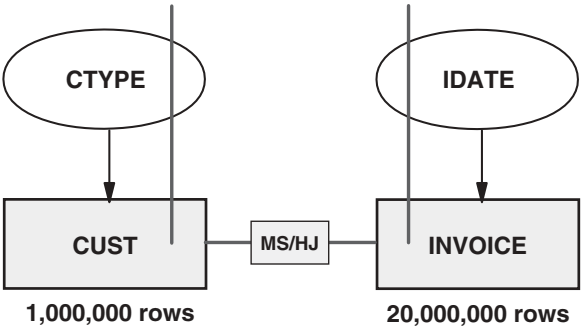


Figure 8.13 Merge scan join example.

The choice of outer and inner table is less important now, although the terms are still relevant. (See Fig. 8.13.) Each local predicate is used, in turn, to produce temporary tables that contain only those rows that are required. Using 0.1% filter factors for both local predicates, the QUBE for these accesses would be:

Step 1: Access Table CUST via Nonclustering Index CTYPE

```
Index CTYPE   TR = 1          TS = 1000
Table CUST    TR = 1000

LRT           TR = 1000       TS = 1000
              1000 × 10 ms    1000 × 0.01 ms
              10 s + 0.01 s = 10 s
```

A table scan would, at least according to the QUBE, also take 10 s but use more CPU time.

Step 2: Access Table INVOICE via Nonclustering Index IDATE

```
Index IDATE     TR = 1            TS = 20,000
Table INVOICE   TR = 20,000

LRT             TR = 20,000       TS = 20,000
                20,000 × 10 ms    20,000 × 0.01 ms
                200 s + 0.2 s = 200 s
```

A table scan would, at least according to the QUBE, also take 200 s but use much more CPU time.

Step 3: Merge and FETCH the Result Rows As the rows are retrieved, they will be moved into two temporary tables. The cost of doing so will be easily absorbed within the cost of accessing the rows in the first place. Neither temporary table will be in the join column sequence, and so they will have to be sorted. The cost of the sort will be the usual figure of 0.01 ms per row. The two tables will finally be merged, the cost of which can again use the same figure of 0.01 ms per row. Thus:

Building temporary tables for Customers and Invoices	0 ms
Sorting customers	1000×0.01 ms $= 0.01$ s
Sorting invoices	$20{,}000 \times 0.01$ ms $= 0.2$ s
Merging	$(1000 + 20{,}000) \times 0.01$ ms $= 0.2$ s

or to keep things simple:

$$\text{Sort and Merge} : 2(1000 + 20{,}000) \times 0.01 \text{ ms} = 0.4 \text{ s}$$
$$\text{Fetch } 1000 \times 20 = 20{,}000 \times 0.1 \text{ ms} = 2 \text{ s}$$

Local Response Time

$$10 \text{ s} + 200 \text{ s} + 2 \text{ s} = 3.5 \text{ min}$$

With DB2 for z/OS, if the outer table had been accessed in the join column sequence, there would be no need to create and sort a temporary table. The inner table, however, is always put into a work file. The merge would then take place, with the inner table's temporary table, as the outer rows were being accessed.

With SQL Server, the merge join can be a regular or a many-to-many operation. The latter uses temporary tables to store the rows. If there are duplicate rows from each input, one of the inputs rewinds to the start of the duplicates as each duplicate from the other input is processed.

Hash Joins

As we have mentioned earlier, Oracle, SQL Server, and DB2 for LUW tend to choose hash join (HJ) instead of merge scan (MS). Hash join is basically merge scan with hashing instead of sorting; the smaller row set is first stored in a temporary table, hashed by (a calculation performed on) the join column(s). Then the other table (or index slice) is scanned and for each row that satisfies the local predicates, the temporary table is checked for matching rows, using the hash (computed) value.

Merge scan is faster if the row sets are presorted in the indexes. If merge scan needs a sort, hash join tends to be faster, particularly if one of the row sets fits in memory (a random touch to a row in a small hash table in memory normally takes less CPU time than sorting and merging a row). If both row sets are large, relative to the available memory, the hash table is partitioned (one partition at a time in memory), and the other row set is scanned several times.

With regard to the QUBE, the initial table or index scans will be identical, regardless of whether merge scan or hash join is being used. Any reduced cost resulting from hashing will be difficult to quantify and so, to avoid unnecessary complexity, we will determine the cost of the hashing process in the same way as we did for MS.

In future, therefore, it will not be necessary to differentiate between merge scan joins and hash joins, unless one wishes to use a reduced cost with regard to the final step—the hash process itself. We will subsequently use the term MS/HJ to refer to either process.

Let us now return to our case study and reconsider program C.

Program C: MS/HJ Considered by the Optimizer (Current Indexes)

With our filter factors for the local predicates IEUR > :IEUR and CCTRY = :CCTRY in CURSORJ, the optimizer may indeed choose the MS/HJ, which as we saw with the nested-loop join figures, might not be a bad idea with the current indexes (Fig. 8.14). In order to make comparisons somewhat easier to understand against the nested-loop joins, we will assume that the optimizer uses the same filter factors that we have been using.

Step 1: Full Table Scan on CUST

```
Table CUST   TR = 1                TS = 1,000,000

LRT          1 × 10 ms             1,000,000 × 0.01 ms
             10 ms + 10 s = 10 s
```

SQL 8.15

```
                                                Program C
DECLARE CURSORJ CURSOR FOR
SELECT    CNAME, CTYPE,  INO, IEUR
FROM      CUST, INVOICE
WHERE     IEUR > :IEUR
          AND
          CCTRY = :CCTRY
          AND
          CUST.CNO = INVOICE.CNO
ORDER BY  IEUR DESC

OPEN CURSORJ
      FETCH CURSORJ     while  IEUR > :IEUR and
                               CCTRY = :CCTRY
CLOSE CURSORJ
```

Step 2: Full Table Scan on INVOICE

```
Table INVOICE   TR = 1      TS = 20,000,000

LRT             1 × 10 ms   20,000,000 × 0.01 ms
                10 ms + 200 s = 200 s
```

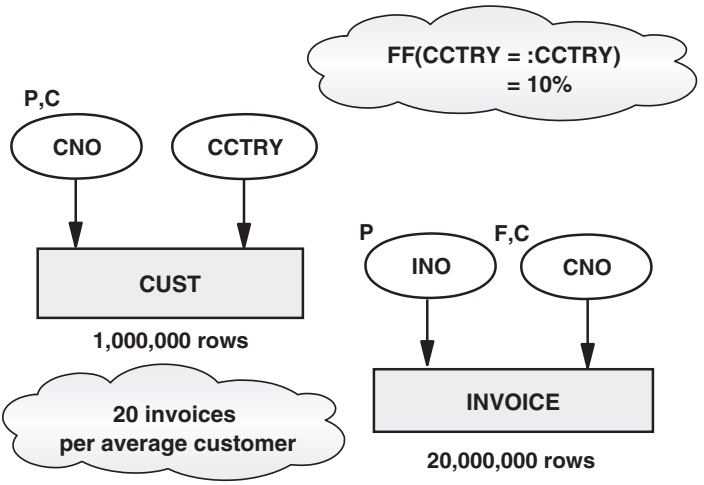


Figure 8.14 Current indexes for the case study.

Step 3: Merge/Hash and FETCH the Result Rows The 100,000 rows resulting from step 1 would have to be sorted into CNO sequence. The 20,000 rows resulting from step 2 would also have to be sorted into CNO sequence. The two sorted tables would then be merged. Taking a common 0.01 ms for each row involved in the sort and merge process:

$$\text{Sort and Merge/Hash } 2(100{,}000 + 20{,}000) \times 0.01 \text{ ms} = 2.4 \text{ s}$$
$$\text{Fetch } 2{,}000 \times 0.1 \text{ ms} = 0.2 \text{ s}$$

Local Response Time With the worst input, the local response time with this access path is the sum of the three components:

$$10 \text{ s} + 200 \text{ s} + 2.6 \text{ s} = 200 \text{ s} = 3.5 \text{ min}$$

Conclusion: MS/HJ with Current Indexes

This is only one third of the LRT achievable with a nested-loop join using the current indexes, but it is certainly not adequate with the worst input using the MS/HJ. As usual, we can now design the best affordable indexes, probably starting by designing and evaluating the *ideal* indexes.

Ideal Indexes

Candidate A for the CUST index access is (CCTRY, CNO, CNAME, CTYPE). This is a three-star index; the access path provides the rows in the join column sequence, and so the rows would not need to be sorted. Candidate A for the INVOICE index access is (IEUR DESC, CNO, INO). This has only two stars because the range predicate means that the access path cannot provide the rows

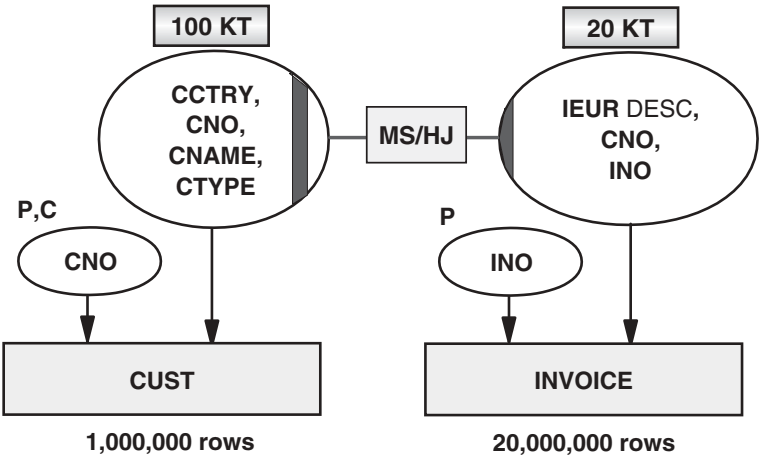


Figure 8.15 MS/HJ with ideal indexes.

in CNO sequence. Candidate B would be (CNO, IEUR DESC, INO), but then the entire table would be involved in the merge/hash process; this would result in scanning 1000 times as many index rows (FF 0.1%). The two candidate A indexes are shown in Figure 8.15. Strictly speaking, IEUR is not required to be a *descending* key in the ideal index; for the same reasons used throughout this entire case study, we will nevertheless assume a descending key.

Step 1: Access Index (CCTRY, CNO, CNAME, CTYPE)

```
Index CCTRY, CNO, CNAME, CTYPE   TR = 1       TS = 10% × 1,000,000

LRT                              TR = 1       TS = 100,000
                                 1 × 10 ms    100,000 × 0.01 ms
                                 10 ms + 1 s = 1 s
```

SQL 8.16

```
DECLARE CURSORJ CURSOR FOR                                  Program C
SELECT    CNAME, CTYPE,  INO, IEUR
FROM      CUST, INVOICE
WHERE     IEUR > :IEUR
          AND
          CCTRY = :CCTRY
          AND
          CUST.CNO = INVOICE.CNO
ORDER BY  IEUR DESC

OPEN CURSORJ
  FETCH CURSORJ  while  IEUR > :IEUR and  CCTRY = :CCTRY
CLOSE CURSORJ
```

Step 2: Access index (IEUR DESC, CNO, INO)

```
Index IEUR DESC, CNO, INO   TR = 1      TS = 0.1% × 20,000,000

LRT                         TR = 1      TS = 20,000
                            1 × 10 ms   20,000 × 0.01 ms
                            10 ms + 0.2 s = 0.2 s
```

Step 3: Merge/Hash and FETCH the Result Rows If the results of steps 1 and 2 had required sorting into the join column sequence, CNO, the cost of step 3 would be twice the cost of the two initial scans:

$$\text{Sort and Merge/Hash } 2(100{,}000 + 20{,}000) \times 0.01 \text{ ms} = 2.4 \text{ s}$$

The outer scan *does*, however, provide the rows in the required sequence; the resulting cost is significantly reduced:

$$\text{Sort and Merge/Hash } 2(20{,}000) \times 0.01 \text{ ms} = 0.4 \text{ s}$$
$$\text{Fetch } 2000 \times 0.1 \text{ ms} = 0.2 \text{ s}$$

Note that this is true in this case only because the first column, CCTRY, is in an equal predicate; with a range predicate, the index scan would not be accessing the rows in CNO sequence.

Local Response Time With the worst input, the local response time with this access path is the sum of the three components:

$$1 \text{ s} + 0.2 \text{ s} + 0.6 \text{ s} = 1.8 \text{ s}$$

Conclusion: MS/HJ with Ideal Indexes

The merge scan figures have been added to the nested-loop comparison figures in Table 8.2.

Table 8.2 Summary of the Local Response Times—2

Type	Program A	Program B/C NLJ	Program C MS/HJ
Current indexes	35 min	10 min	3.5 min
Ideal indexes	17 min	3.5 min	1.8 s
Ideal indexes, ideal program (+), multi-screen result		2 s	
Ideal indexes, ideal program (+), single-screen result (FF pitfall)		3.5 min	
BJQ ideal indexes, ideal program (+), single-screen result (FF Pitfall)		0.2 s	

The MS/HJ with ideal indexes is quite clearly the most promising access path, without having to resort to one screen per transaction or using the BJQ. It has succeeded for two reasons:

1. It has removed the potentially huge problem we encountered in the nested-loop discussion—the random touches to an inner table, or its index.
2. The incredibly fast sequential access now available with current hardware.

Regardless of the tremendous improvement, the resulting elapsed time of nearly 2 s is perhaps still a little high for an on-line query. If this response time is unacceptable, a nested-loop join satisfying the BJQ would be necessary.

NESTED-LOOP JOINS VERSUS MS/HJ AND IDEAL INDEXES

Nested-Loop Joins Versus MS/HJ

Traditionally, nested loop joins (NLJ) have been the preferred join technique, assuming appropriate indexes have been created and the result table is not exceptionally large. These "appropriate indexes" refer, of course, to the relevant semifat, fat, or even ideal indexes with which we are hopefully now perfectly familiar. Unfortunately, as we have seen in this chapter, even these well-designed indexes may not be able to provide good performance; large numbers of random touches to the inner table (or index) may constitute a huge problem. To overcome this in the past, we have had to resort to limiting the size of the result set, or applying the basic join question (BJQ) such that all local predicates refer to one table, neither solution perhaps being desirable.

Over the last few years, as we have observed in this book on several occasions, hardware has favored the sequential touch. The performance of random touches has improved too (e.g., faster disks and more memory), but not to the same extent. The result of this is clearly visible by comparing the 0.01 ms TS with the 10 ms TR in the QUBE. We can access 1000 rows sequentially in the time it takes to access one row randomly (ignoring the additional CPU cost and other issues, such as comebacks, that will be discussed in Chapter 15).

One of the by-products of this transition is that optimizers are now more likely to choose scans rather than random touches. We have already seen that even with filter factors as low as 0.1%, a table scan may well be used in preference to a nonfat, nonclustered index scan. For the same reason, they are now also more likely to choose MS/HJ than they have in the past. These issues are particularly important ones to be appreciated by those who have worked with relational databases for a considerable period of time, but who have not really considered the implications of these changes. One perhaps should not be too hasty, for example, in attempting to persuade the optimizer to revert to an index scan or a NLJ.

It is, of course, the role of the optimizer to determine how to access a table or index, and which join technique to choose, but these issues affect the index

design process, and as such, we should be aware of the trend towards table scans and MS/HJ.

Ideal Indexes for Joins

Ideal indexes do not differ greatly according to the type of join. The ideal index on an outer table in a NLJ will be identical to that for either table in an MS/HJ—preferably a fat index that provides a very thin index slice. The only difference occurs with the inner table of an NLJ; here the join column will be part of the index, but again preferably a fat index that provides a very thin index slice, and one that doesn't give rise to an excessive number of TRs.

Remember also that with MS, a sort may not be required on the outer table if the access path provides the rows in join sequence; the join column should therefore be included in the index, after any equal predicates. The CPU and elapsed time savings involved in not having to create, sort, and access a work file could be considerable.

We have seen that with HJ, the smaller row set is first stored in a temporary table, hashed by the join column(s). The other table (or index slice) is then scanned and for each row that satisfies the local predicates, the temporary table is checked for matching rows, using the hash value.

JOINING MORE THAN TWO TABLES

When there are more than two tables in a join, it is not uncommon that some connections are not supported by common columns (such as primary key and foreign key). In Figure 8.16, tables INVOICE and ITEM can only be joined by a Cartesian join, which would mean creating a temporary table containing all combinations of *local rows* in the two tables, $20{,}000 \times 1000 = 20{,}000{,}000$ with

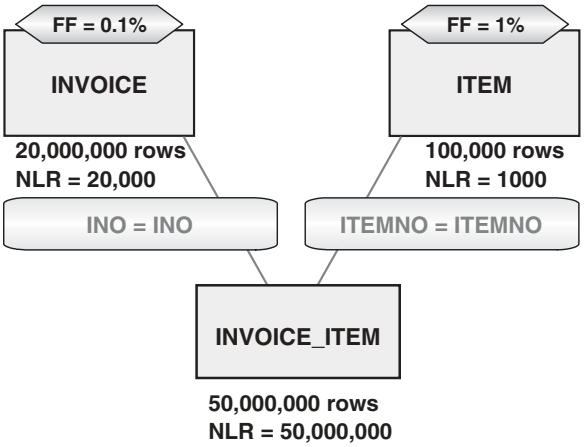


Figure 8.16 Joining more than two tables.

the assumed filter factors (the local predicates are *not* shown in the diagram). As this is a relatively expensive process, the table access order suggested by the NLRs (the number of local rows)—ITEM, INVOICE, INVOICE_ITEM—is probably not the fastest one.

The only alternatives that are possible *without* a Cartesian join are shown in Figures 8.17 and 8.18. The touches shown *above* each index are the *total* number

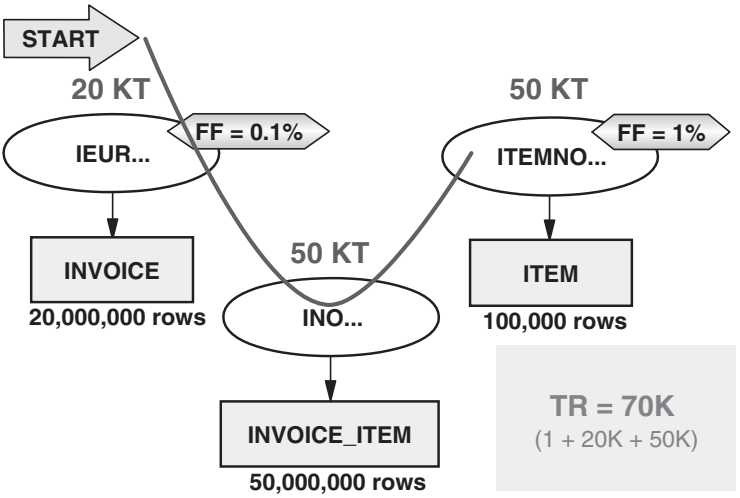


Figure 8.17 Predicting table access order—1.

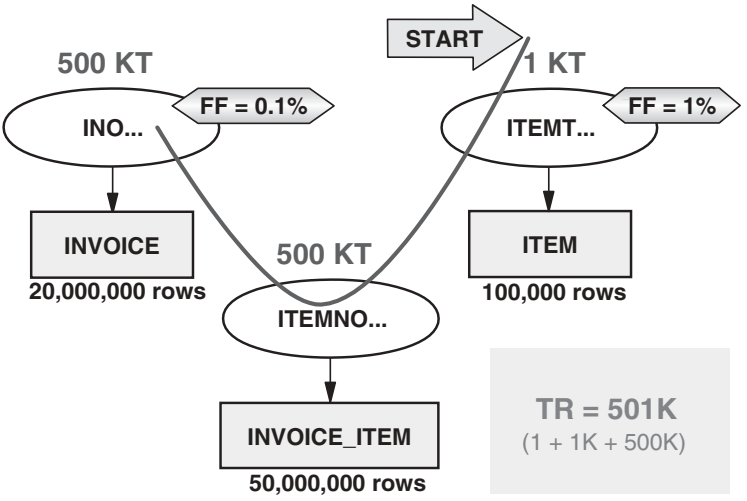


Figure 8.18 Predicting table access order— 2.

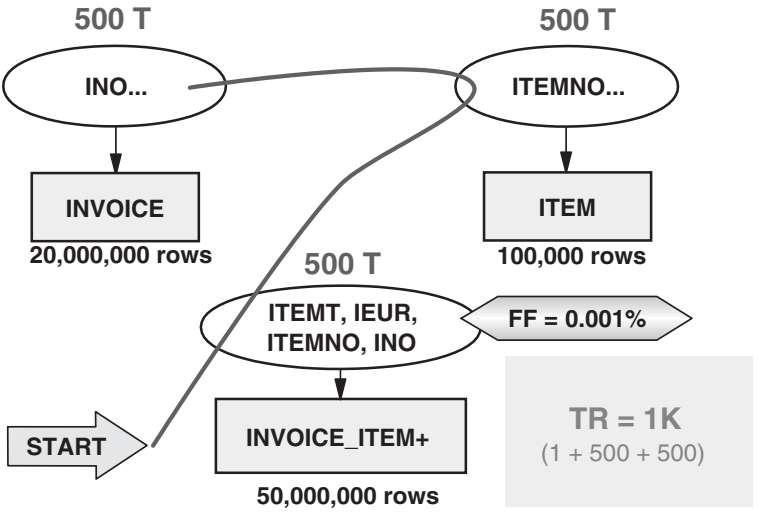


Figure 8.19 Predicting table access order in a nested-loop join.

of touches to the indexes; those shown at the *bottom right* of each diagram summarize the *TRs*—the major component. All accesses are index only.

The indexes designed for table access (INVOICE, INVOICE_ITEM, ITEM) provide a better access path than the alternative. However, 70,000 random touches (Fig. 8.17) take $70{,}000 \times 10$ ms $= 700$ s, according to the QUBE. If this is not satisfactory, the predicate columns would have to be copied to table INVOICE_ITEM. Then, with an index that contained all the predicate columns, as shown in Figure 8.19, the number of random touches would be reduced to 1000—another example of the importance of the BJQ.

Would MS/HJ provide better performance? There are no local predicates on table INVOICE_ITEM, so with a MS/HJ, the first step would require 50,000,000 sequential touches; this would take $50{,}000{,}000 \times 0.01$ ms $= 500$ s, according to the QUBE.

It would appear to be better to first join table ITEM with table INVOICE_ITEM using a nested loop, and then join the intermediate result with table INVOICE using MS/HJ (Fig. 8.20). The ideal index on table INVOICE is now one that starts with the local predicate columns (IEUR...).

Step 1: Join ITEM with INVOICE_ITEM (NL)

```
Index (ITEMT,...)    TR = 1      TS = 0.01 × 100,000 = 1000
Index (ITEMNO,...)   TR = 1000   TS = 0.01 × 50,000,000 = 500,000
```

$$\text{LRT} = 1001 \times 10 \text{ ms} + 501{,}000 \times 0.01 \text{ ms} = 15 \text{ s}$$

The work file contains 500,000 rows.

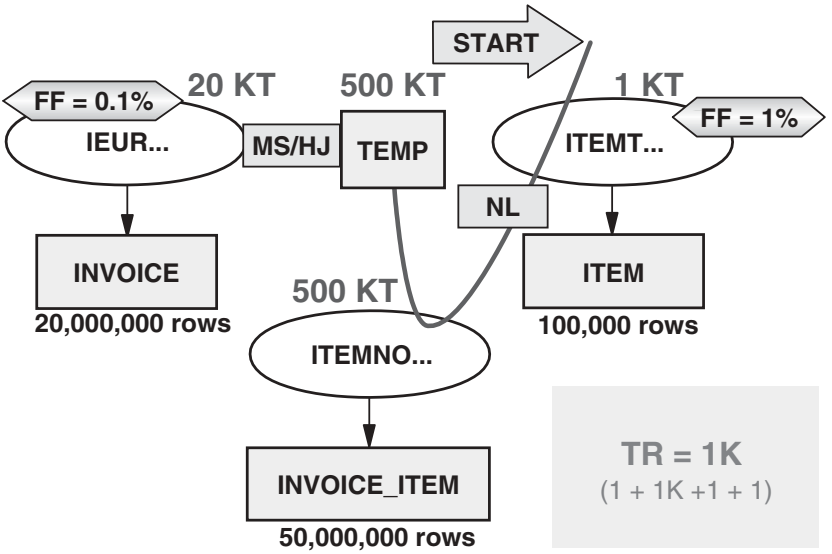


Figure 8.20 Predicting table access order in a nested-loop and merge scan/hash join.

Step 2: Join the Work File with INVOICE (MS/HJ)

```
Workfile            TR = 1   TS = 500,000
Index (IEUR,...)    TR = 1   TS = 0.001 × 20,000,000 = 20,000
```

$$\text{LRT} = 2 \times 10 \text{ ms} + 520{,}000 \times 0.01 \text{ ms} = 5 \text{ s}$$

With merge scan, 520,000 rows must be sorted and merged:

$$2 \times 520{,}000 \times 0.01 \text{ ms} = 10 \text{ s}$$

Hash join is probably faster because the smaller row set, 20,000, should fit in memory.

Local Response Time The total response time, with these indexes, is 30 s using merge scan or between 20 and 30 s with hash join—impressive compared to nested loop (12 min).

WHY JOINS OFTEN PERFORM POORLY

Fuzzy Indexing

One of the oldest indexing recommendations is to "index join columns." Actually this is an expanded version of the basic recommendation "*create an index for the primary key and an index starting with each foreign key*." Indexing the join predicate columns makes the nested loop a feasible alternative, but thin join

column indexes don't necessarily lead to totally acceptable response times. Both the join predicate column indexes and the local predicate columns often need to be fat. Furthermore, different table access orders may lead to totally different index requirements.

Indexing based on an assumed access path increases the probability that the optimizer chooses the assumed access path. While another access path with ideal indexes for that alternative could perform even better, consistent indexing based on an assumed access path is likely to lead to shorter response times than fuzzy indexing without any assumption about the join method and the table access order.

Optimizer May Choose the Wrong Table Access Order

A SELECT statement referring to several tables has many more alternative access paths than a single-table SELECT. It is not uncommon for incorrect filter factor estimates made by the optimizer to lead to an incorrect table access order and even to the wrong join method. Furthermore, as with single-table SELECTs, the access path may be optimal for the *average* input but poor for the *worst* input. With a join, the impact of a poor access path is often dramatic. Fortunately, it is now possible to help the optimizers in many ways—with a hint, for instance. In the early days, programmers sometimes had to replace a join cursor with several simple cursors to achieve a good table access order, or a good join method.

Optimistic Table Design

The application we have discussed throughout this chapter looks like a data warehouse query, but actually the problem caused by local predicates on two large tables is quite common in operational applications; even the best access path is far too expensive with the worst input. With this problem, the index designer, the application programmer and the optimizer may all plead not guilty. The problem cannot even be fixed by replacing the join with two cursors; on the contrary, the CPU time would increase a little as the number of executed SQL calls becomes slightly higher. The fault may lie with the database specialist who firmly believes that redundant data in tables can never be appropriate. The Basic Join Question should be kept in mind both when designing tables and when writing joins. The issues associated with redundant data are discussed in the Table Design Considerations section at the end of this chapter.

DESIGNING INDEXES FOR SUBQUERIES

From the performance point of view, subqueries are quite similar to joins. In fact, the current optimizers often rewrite a subquery into a join before selecting the access path. If the optimizer doesn't do this, the type of subquery may in itself determine the table access order. Noncorrelated subqueries are normally

executed by starting with the *innermost* SELECT. The result table is stored as a temporary table, which is accessed by the next SELECT. Correlated subqueries are normally executed by starting with the *outermost* SELECT. In any case, as with joins, the indexes should be designed based on a table access order that seems to produce the fastest access path. If the best table access order is *not* chosen, the programmer may have to rewrite the statement, or—in some cases—use a join.

DESIGNING INDEXES FOR UNIONS

The SELECT statements that are connected by UNION or UNION ALL are optimized and executed one at a time. The indexes should therefore be designed as appropriate for each individual SELECT. It should be noted that UNION with an ORDER BY would cause early materialization.

TABLE DESIGN CONSIDERATIONS

We saw earlier that many database specialists believe that redundant data in tables can never be appropriate. We also saw that the problems associated with the BJQ need to be kept in mind when designing tables. We will now consider the table design issues that affect SQL performance.

Redundant Data

There are two ways to make a join faster by using redundant data:

1. A column may be copied to the dependent table (*downward denormalization*).
2. Summary data may be added to the parent table (*upward denormalization*).

Downward Denormalization

In our case study, we employed downward denormalization to eliminate the large number of unproductive TRs to the inner table's (CUST) index; adding column CCTRY to table INVOICE was a tempting solution, as reshown in Figure 8.21.

Additional storage cost is low and updating CCTRY in table INVOICE when a customer moves to another country is not a major problem. If the INVOICE rows of a customer are adjacent [index (CNO) is the clustering index], this involves only one random table touch, together with several touches to move a set of rows in index (CCTRY, IEUR, INO, CNO). Admittedly if a large customer has 1000 invoices, these moves could cause 2000 random index touches taking up to 20 s, but as this does not happen frequently, (large customers do not often move to another country), this may be tolerable. In general, however, when

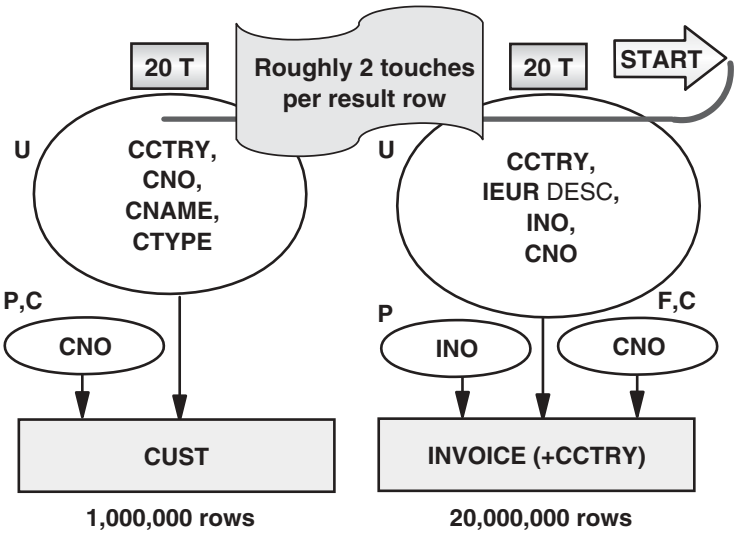


Figure 8.21 Ideal indexes for a nested-loop join that satisfies the basic join question.

downward denormalization is considered, it is important to predict the number of worst-case random index touches that take place when the redundant column (CCTRY in our example) is updated.

Upward Denormalization

Let us assume that the sort requirement in our case study is ORDER BY CNAME, CNO instead of ORDER BY IEUR DESC. Now, a nested loop with CUST as the outer table would eliminate a sort. This access path would then have the potential for making a transaction with 20 FETCHes (one screen) very fast. Without redundant data, however, it could take 1000 random touches—10 s according to the QUBE—to an index on table INVOICE to find one result row, assuming that only 0.1% of the one million customers have at least one large invoice. Upward denormalization should then be considered to reduce the number of these nonproductive random touches, which are required to determine whether a customer has any large invoices.

SQL 8.17L

```
WHERE
      CUST.CNO = INVOICE.CNO
AND   CCTRY = :CCTRY
AND   CLARGE = 1
AND   IEUR > :IEUR
```

The most straightforward solution is to add a new column CLARGE to table CUST, and add the appropriate predicate as shown in SQL 8.17L; when CLARGE = 1, the customer has at least one large invoice. Obviously, this would be clumsy for two reasons:

1. Column CLARGE is not easy to maintain when INVOICE rows are deleted or inserted.
2. Changing the definition of CLARGE would require a fairly massive batch job.

Adding column CTOTAL_IEUR, the total invoice value for each customer to table CUST, together with the appropriate predicate as shown in SQL 8.17T would eliminate these problems but would have higher maintenance overheads: Every INSERT and DELETE INVOICE would require two random touches, one to table CUST and one to an index that contains the CTOTAL_IEUR column. This solution would avoid *some* unproductive accesses to the index on table INVOICE, those for customers whose total invoice value was not greater than :IEUR; any customer whose total invoice value was greater than :IEUR, could have one or more large invoices, or just several small ones.

SQL 8.17T

```
WHERE
      CUST.CNO = INVOICE.CNO
AND   CCTRY = :CCTRY
AND   CTOTAL_IEUR > :IEUR
AND   IEUR > :IEUR
```

Adding column CMAX_IEUR, the largest IEUR value of each customer's invoices, (SQL 8.17M) to table CUST would have *no* unproductive accesses to the index on table INVOICE. The maintenance problem would also be slightly reduced because only those inserts and deletes for invoices that affected CMAX_IEUR would cause the latter to be updated. It should be noted, however, that touches would still be needed on the CUST index to check the largest IEUR value.

SQL 8.17M

```
WHERE
      CUST.CNO = INVOICE.CNO
AND   CCTRY = :CCTRY
AND   CMAX_IEUR > :IEUR
AND   IEUR > :IEUR
```

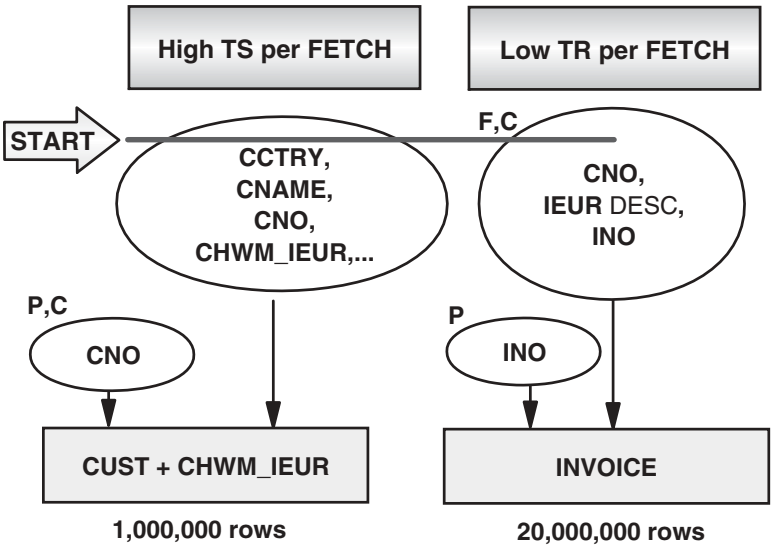


Figure 8.22 Ideal indexes that almost satisfy the basic join question.

The best solution (refer to Fig. 8.22 and the SQL statement shown in SQL 8.17H) may well be to add CHWM_IEUR, a high-water mark, the largest IEUR value that a customer has recently had but not necessarily *still has*. Here we are trying to balance the cost of the maintenance against the benefits achieved. CHWM_IEUR does *not* have to be exact, but it must be high enough to catch *all* large invoices; if it is too high, the invoice table's index will simply be checked unnecessarily. Constantly updating CHWM_IEUR will incur excessive costs, and so the high-water marks need only be periodically refreshed, as far as deletes are concerned, by means of a CHWM_IEUR update refresh program; these deletes will cause CHWM_IEUR to be reduced, thereby eliminating the unproductive TRs. As far as inserts are concerned, CHWM_IEUR needs to be updated only when a new invoice to a customer has a higher IEUR value than any existing invoice for that customer; this only needs one random touch to the index on table CUST for every INSERT INVOICE to check the value of CHWM_IEUR.

SQL 8.17H

```
WHERE
      CUST.CNO = INVOICE.CNO
AND   CCTRY = :CCTRY
AND   CHWM_IEUR > :IEUR
AND   IEUR > :IEUR
```

When all the high-water marks have been refreshed, the average FETCH may require 1000 sequential touches to the first index but only one touch, normally random, to the second index. Thus, the QUBE for a transaction with 20 FETCHes would be:

$$20 \times 10 \text{ ms} + 20{,}000 \times 0.01 \text{ ms} + 20 \times 0.1 \text{ ms} = 0.4 \text{ s}$$

Cost of Denormalization

The biggest performance concern is normally the I/O time required to update the redundant data added to the table and to one of its indexes. With downward denormalization, a large number of index rows may have to be moved, which may make a simple UPDATE very slow. Upward denormalization is not as likely to cause I/O bursts due to a single update, but many INSERTS, UPDATEs, and DELETEs may cause a few extra disk I/Os to the parent table and to one of its indexes. In extreme cases, say more than 10 INSERTs or UPDATEs per second, the *disk drive load* created by these I/Os can be an issue.

Nested-Loop Join and MS/HJ Versus Denormalization

It is understandable that many database specialists are reluctant to add redundant columns to operational tables. Denormalization is not only a trade-off between retrieval speed and update speed—it is also, to some extent, a trade-off between performance and data integrity, even if triggers are used to maintain the redundant data. But then, when nested loop causes too many random reads and MS/HJ consumes too much CPU time, denormalization may appear to be the only option available. Nevertheless, before taking this drastic approach, it is obviously of great importance to ensure that everything that could be done to avoid it, has been done. This means being absolutely sure that the best fat indexes for NLJ or MS/HJ have been considered and making any troublesome indexes resident in memory; the purchase of additional memory to ensure more resident indexes, or of new disks with fast sequential read, might push the break-even point with regard to the need to denormalize just that little bit further.

This is without doubt a difficult balancing act. We do not want to give the impression that non-BJQ is always easy to fix with superfast sequential scans. The figures might suggest this in our case study, but tables in real life are often much larger, often containing over 100 million rows and, with high transaction rates, CPU time is still a very important consideration.

Unconscious Table Design

From the performance point of view, it is more difficult to understand why so many databases have tables that have a 1:1 or 1:C (C = conditional; 0 or 1) relationship, as shown in Figure 8.23.

Why create four tables instead of one CUST table? Flexibility is not an issue when the relationship can never become 1:M. In this example a customer

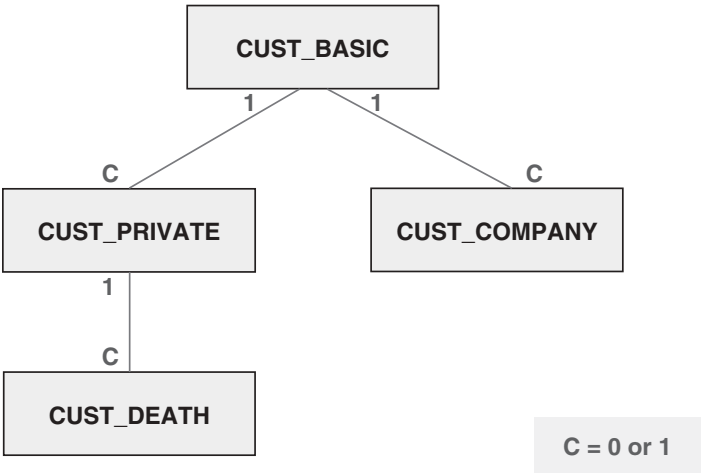


Figure 8.23 Better than one table?

is either a company or a person, and few customers die twice! Combining the four tables into one CUST table with several empty columns (in every row either the company-related or the person-related columns are empty, as well as the death-related columns for all customers that are still alive) is only a trade-off between storage space and processing time (the number of random touches). Missing data is *not* a violation of the normalization rules.

Consider a query that selects some basic columns and some company-related columns for a given CNO. With one CUST table, this can be done with one random touch, but with a multi-table solution the *minimum* will take two random touches. The difference can be even greater; consider the familiar query shown in SQL 8.18:

SQL 8.18

```
SELECT        FNAME, CNO
FROM          CUST
WHERE         LNAME = :LNAME
              AND
              CITY = :CITY
ORDER BY      FNAME
```

Index (CITY, LNAME) may not provide adequate performance, but we know how to make this query very fast by simply adding columns to the index. The two-table solution shown in Figure 8.24 requires a join, and it is a *non-BJQ join*. We have seen what this leads to: either a lot of unproductive random index touches to the inner table (or index) or MS/HJ with potentially high CPU time

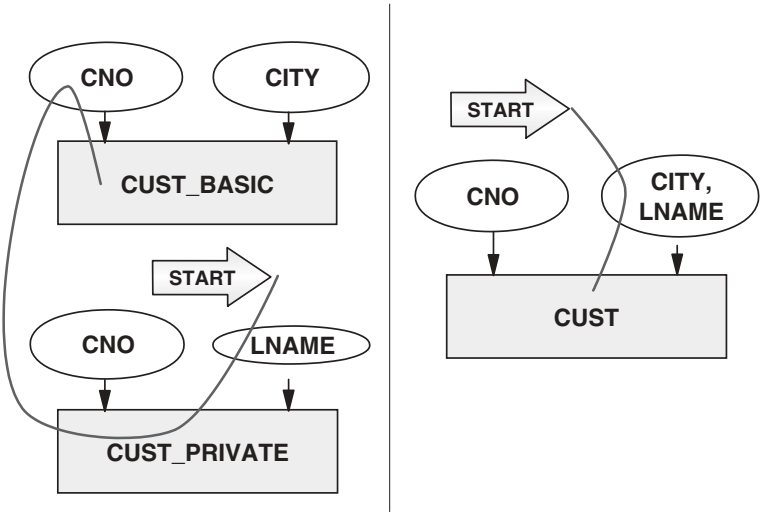


Figure 8.24 Which performs better?

(and probably a sort). Or perhaps we should employ downward denormalization, for example, by copying column CITY to the CUST_PRIVATE table.

Before deciding to split a table, the benefit should be quantified. The disk space needed for the CUST table might be $1.5 \times 1{,}000{,}000$ rows $\times$ 1000 bytes/row = 1.5 GB. If the monthly cost of disk space is U.S.$50 per gigabyte, the saving in storage cost achievable by splitting the CUST table would be less than U.S.$75 per month.

Sometimes a separate table, such as CUST_DEATH, is created because some programs are only interested in the death data. It is true that processing such a table can be very fast, but the same benefits can normally be achieved by a well-designed index on the CUST table.

Often a database consists of too many tables because it just seemed more appropriate, or maybe the objects just became tables without any performance considerations. Such errors are difficult to correct when the application is in production. One case was recently encountered where even the ideal indexes were unable to make operational transactions, accessing more than a dozen tables, fast enough. The final solution that had to be developed was to synchronously replicate some of the data to a new combined table that was accessed by the critical transactions, while most of the programs continued to access the small tables. Yet another application performed so poorly that it just could not be used. The optimizer being used appeared to have problems with joins that accessed more than 5 tables (which was quite common; in fact some accessed more than 20!). The table structure could have been made very simple, using fewer tables, but that would have required a large rewrite effort.

When tables are designed without hardware-conscious performance awareness, the following problems are likely:

1. The number of random touches will be high, even with the best possible indexes.
2. Index design will become difficult because of complex joins.
3. The optimizer will make wrong choices because of complex joins.

There is a simple way to prevent these problems—any table design with a 1:1 or 1:C relationship should be justified by the person proposing it. There are cases where table splitting is a good idea, or even necessary, but they are not at all common with current hardware. The splitter should have the burden of proof.

EXERCISES

8.1. Estimate the elapsed time for the join shown in Figure 8.25, using the given filter factors.

8.2. Design the best possible indexes for the join without adding redundant table columns and estimate the elapsed time.

8.3. There are three foreign keys in table CUST pointing to small code tables. Estimate the local response time for the four-table join below with nested loop and the best table access order (Fig. 8.26).

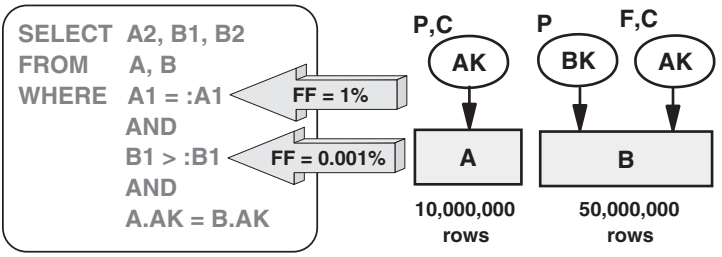


Figure 8.25 Slow table join.

SQL 8.19

```
SELECT      CNAME, C1TEXT, C2TEXT, C3TEXT
FROM        CUST, C1, C2, C3
WHERE       CUST.CNO = :CNO
            AND
            CUST.C1PK = C1.C1PK
            AND
            CUST.C2PK = C2.C2PK
            AND
            CUST.C3PK = C3.C3PK
```

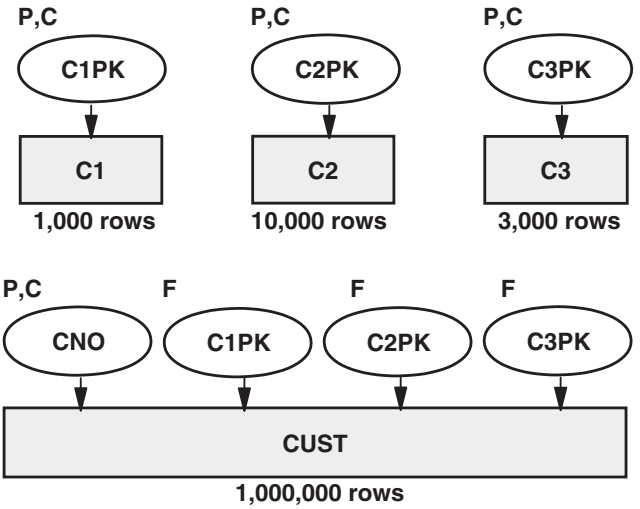


Figure 8.26 A four-table join.

8.4. Assume SQL 8.19 is executed a million times in a batch job. Should indexing be improved? What is the tuning potential? Are there other ways to improve the performance of this SELECT statement?

Chapter 9

Star Join Considerations

- Introduction to star joins by means of an example
- Role of Cartesian products, dimension tables, and fact tables
- Comparison of these issues with ordinary joins
- Index design for dimension and fact tables
- Huge impact of the table access order with regard to the two types of table and with Cartesian joins
- Comparisons made using the QUBE
- Limitations of star joins and the use of summary or query tables

INTRODUCTION

Star joins are different from traditional joins in two respects:

1. The central table in the starlike table structure such as the one shown in Figure 9.1, called the *fact table*, is much larger than the ones surrounding it, which are called *dimension tables*.
2. The best access path often contains *Cartesian products* of the dimension tables; this means they *do not have any common columns*; all combinations of the dimension table rows will be joined, as long as they satisfy their *local* predicates.

Kevin Viers (3, p. 611) puts this rather nicely when he talks of the Cartesian join as "This might sound strange, but it makes sense."

The fact table, SALES, in Figure 9.1 is huge, one billion rows, and contains one row for *every* sale; which *item* was sold at which *store* on what *date* and *to whom*! The four dimension tables provide the detail of this information. To understand what this means, let's take the example shown in SQL 9.1. We have simplified the situation a little by using three dimension tables only—it is assumed that no customer information is required.

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

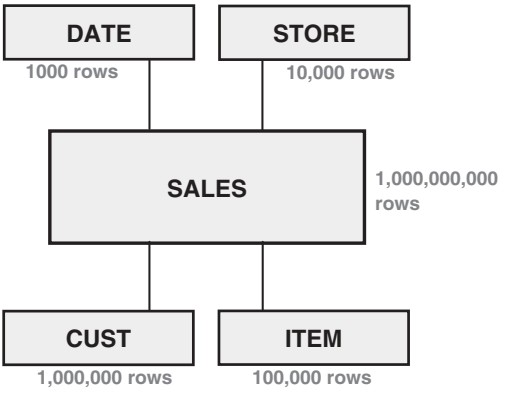


Figure 9.1 Star schema—a large fact table with small dimension tables.

SQL 9.1

```
SELECT   SUM (EUR)
FROM     SALES, DATE, ITEM, STORE
WHERE    ITEM.GROUP = 901
         AND
         DATE.WEEK = 327
         AND
         STORE.REGION = 101
         AND
         SALES.ITEMPK = ITEM.ITEMPK
         AND
         SALES.DATEPK = DATE.DATEPK
         AND
         SALES.STOREPK = STORE.STOREPK
```

This star join is extremely simple; it determines *the total sales (EUR) of one product group during one week in one geographical area*. The product group code is in table ITEM, the week number in table DATE, and the area code in table STORE.

The star schema (and the star join) has become common in data warehouses because it is easy to generate a single SELECT statement (a join), which produces the result table while the end users see the data as an *n*-dimensional cube.

The border between traditional joins and star joins is drawn in water. The four-table join shown in Figure 8.26 *may look like* a star join, especially if the code tables are drawn around the customer table. However, the SELECT statement is *not* a star join because the second characteristic does not apply; Cartesian joins of the code tables would make no sense at all. The local-row rule of thumb considered in Chapter 8 and, indeed, common sense tell us that the *outermost* table should be CUST. The order in which the other tables are accessed does not make any difference.

In a star join, the fact table would normally have a huge number of local rows. In this case, at least according to the rule of thumb, the fact table should be the *innermost* table in a nested loop join. This implies Cartesian joins as we will see in due course because the dimension tables typically do not have any common columns.

As we did in Chapter 8, we will now consider the two major issues affecting the performance of joins; the *indexes on the tables* and the *table access order*. As we do this, the concept of the Cartesian join should become clear. We will first discuss the indexes on the dimension tables, then the table access order, and finally the indexes on the fact table.

INDEXES ON DIMENSION TABLES

If the end users are able to generate predicates with *no restrictions*, the indexes on the dimension tables need to be adequate for *any* local predicate combination. Figure 9.2 shows just one of our four dimension tables, the Customer table. The first version could be one fat index per search column. With sufficient statistics, including histograms, the optimizer is likely to choose the most selective index, but in most cases there will only be *one* matching column.

The number of sequential touches to the index will be the product of the filter factors (FF) for the *matching* predicates multiplied by the number of rows in the dimension table, for example:

```
WHERE   DOB BETWEEN :BDATE1 AND :BDATE2    FF = 10%
        AND
        SEX = :SEX                         FF = 50%
```

These local predicates on the dimension table CUST would result in $0.1 \times 1{,}000{,}000 = 100{,}000$ sequential touches to index (DOB, ...) in Figure 9.2 because there is only *one* matching predicate; this is because the range predicate column, DOB, is the *last* matching column. According to the QUBE, this step would take one second.

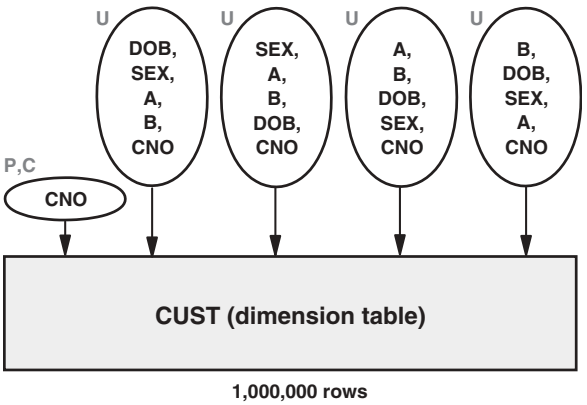


Figure 9.2 Starting point: at least one matching column.

To maximize the number of matching columns for *any* WHERE clause, $4 \times 3 \times 2 \times 1 = 24$ *fat indexes* would be needed in this case with the four search columns, DOB, SEX, A, and B. By interviewing end users or by tracing their queries, it may be possible to achieve *more than one* matching column for many queries, by using a much smaller number of indexes. For example, if the users are often interested in the sales per age group (FF = 10%) and sex, index (SEX, DOB, A, B, CNO) would be better than any of the indexes shown in Figure 9.2 because men in their thirties, for instance, are next to each other. The QUBE for scanning this particular index slice would be only 0.5 s because we would have two matching columns and a filter factor of 0.05 instead of 0.1.

HUGE IMPACT OF THE TABLE ACCESS ORDER

The table access order in a star join is even more important than with a traditional join. Suppose we started with the DATE dimension table, to identify the dates for the week number required. We have about 3 years of entries in the table, one row per day; as we are interested in only one week, the filter factor for the DATE access is 7/1000 = 0.7%—only 7 rows will be accessed in the table. It seems reasonable to assume that 0.7% of the SALES rows will be required also, as shown in Figure 9.3. This results in 7,000,000 touches to the table SALES and its index—or preferably just to the index.

The corresponding figures for starting with the STORE or ITEM table would be even greater, so let's consider the possibility of accessing *all* the dimension tables *before* reading the relevant rows in the fact table, SALES, as shown in Figure 9.4. Of course, although each dimension table has a *local* predicate, there are no *join* predicates between them.

If the fact table is the *innermost* table in the star join, the DBMS would first have to build *intermediate tables* that would contain *every* valid foreign key combination for the query as shown. These tables are *Cartesian products* of the

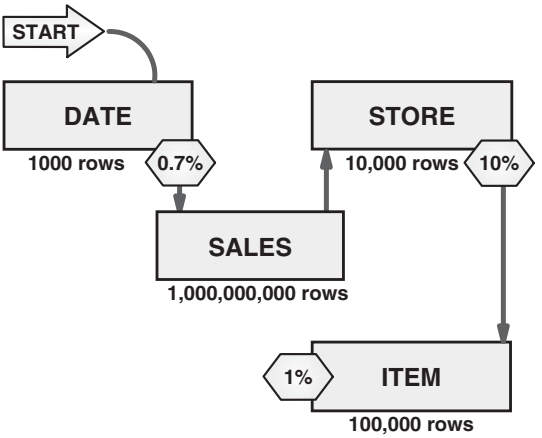


Figure 9.3 Possible table access order.

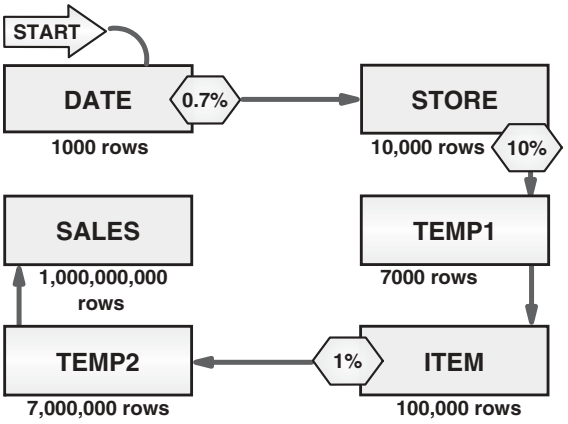


Figure 9.4 Probably a better table access order.

primary keys of those rows in the three dimension tables that satisfy their local predicates. If the primary keys are short, the size of the intermediate tables in the example might be less than 100 MB, probably small enough to fit in a database buffer pool.

The number of rows created in the *first* temporary table will be

$$(0.7\% \times 1000) \times (10\% \times 10{,}000) = 7 \times 1000 = 7000$$

The number of rows created in the *second* temporary table will be

$$7000 \times (1\% \times 100{,}000) = 7000 \times 1000 = 7{,}000{,}000$$

Figure 9.5 shows part of the *second* intermediate table that would be built, the primary keys of the Cartesian product of the three dimension tables, DATE, STORE, and ITEM. The intermediate table, as we have just calculated, would contain 7,000,000 rows.

DATENO	STORENO	ITEMNO
.....	.....	.....
764	235	1002
764	235	2890
764	235	4553
764	235	6179
.....	.....	.....

7,000,000 short rows

Figure 9.5 Cartesian product—intermediate table 2.

INDEXES ON FACT TABLES

When the second intermediate table in Figure 9.4 has been created, the DBMS must read all the fact rows that have the foreign key value combinations. Sometimes there will be several fact rows that contain the same combination of keys (a store sells a particular item several times during a day); sometimes there will be no fact rows relating to a row in the intermediate table. Consequently, it will not be easy to estimate the number of touches to the fact table, but this could be as high as one million.

A good fat index, such as the one shown in Figure 9.6, starting with the most selective column, would be necessary to achieve an acceptable elapsed time. Figure 9.7 makes it much easier to provide a quick estimate for the time required by the fat index scan. It shows *two* possible ways in which the scan could theoretically take place.

First, the optimizer might choose to scan a *single* slice of the index, where the thickness of the scan was determined by the DATENO column alone; only one matching column. The reason for this is that essentially we are searching for a *range* of dates—the seven days in the week requested, DATENO values 764 to 770, for instance. We have already seen that this slice consists of $0.7\% \times 1{,}000{,}000{,}000 = 7{,}000{,}000$ rows.

Second, a "smart" optimizer could decide to subdivide this into what we might call subscans. For each day (which has $1{,}000{,}000{,}000/1000 = 1$ million entries), we would have 1000 subslices based on the ITEMNO column (1% of 100,000 items). For the week, we would then have 7000 individual subslices (two matching columns). The average thickness of each of these slices would be $1{,}000{,}000/100{,}000 = 10$ rows. In this case, 7000 subslices each of thickness 10 rows would mean 70,000 rows in total.

A comparison of these two options would then be

1 MC $\quad 1 \times 10 \text{ ms} + 7{,}000{,}000 \times 0.01 \text{ ms} = 70 \text{ s}$

2 MC $\quad 7000 \times 10 \text{ ms} + (7000 \times 10 \times 0.01) \text{ ms} = 70 \text{ s} + 0.7 \text{ s} = 71 \text{ s}$

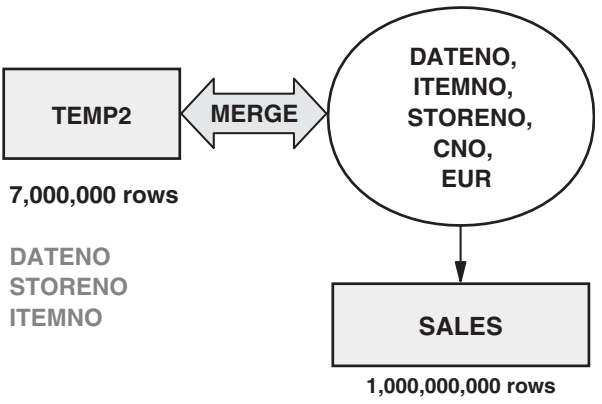


Figure 9.6 Fat index on the fact table.

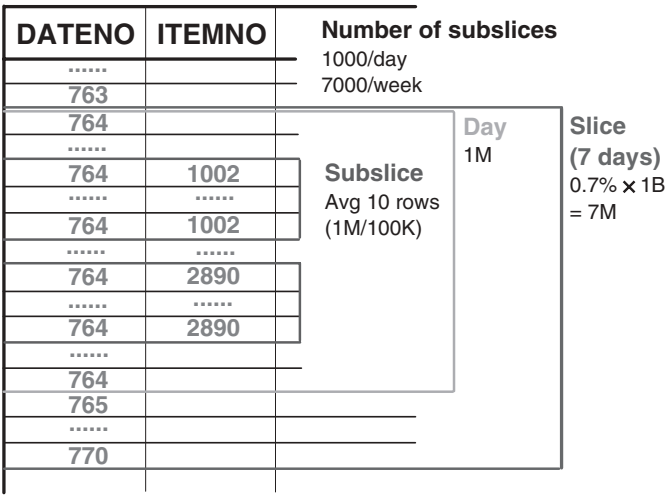


Figure 9.7 Fact table scans.

Before accessing the fact table, the intermediate table would have been sorted by the fat index columns. This is a relatively fast operation, assuming there is enough room for the work files in memory.

With the given filter factors and the assumed number of qualifying fact table rows, the local response time for the star join will probably be a few minutes, as long as the optimizer chooses the same access path. This estimate includes accessing the dimension tables as well as building and reading the two intermediate tables.

If there are only four dimension tables, we are reasonably well prepared for any star join with four fat indexes on the fact table, each one starting with the foreign key of a dimension table, as shown in Figure 9.8.

The four fat indexes on a billion-row table would, however, be quite large. The total disk space requirement for these four indexes would probably be hundreds of gigabytes, perhaps

$$4 \times 1.5 \times 1{,}000{,}000{,}000 \times 40 \text{ bytes} = 240 \text{ GB}$$

Assuming a monthly rent for disk space of $50 per gigabyte, the cost of the four indexes would then be U.S.$12,000 per month. In actual fact, a fat index on a table is often larger than the fact table itself for two reasons:

1. Tables are normally compressed, whereas indexes are not.
2. As the inserts to a fact table normally go at the end, a fact table may not need any distributed free space. The inserts to all indexes other than the clustering index (often starting with a time stamp) are random. To avoid frequent index reorganization, which could take several hours with one billion rows in the table even with current hardware, most of the indexes

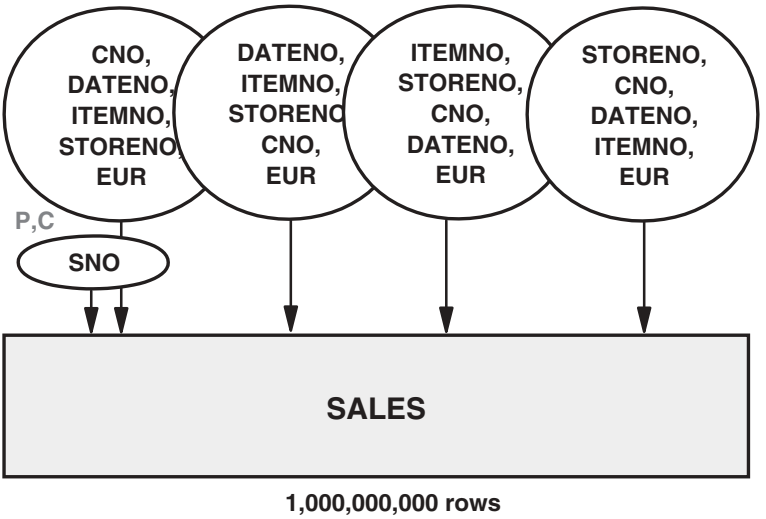


Figure 9.8 Four fat indexes on the fact table—one per dimension.

on a fact table should have ample free space on the leaf pages, perhaps 30 or 40%.

Nevertheless, the greatest performance issue is the unpredictability of the filter factors and the number of qualifying rows. If the query refers to an item group that covers not 1% but 10% of the sales, the response time growth may become unacceptable, as the final intermediate table would have 70 million rows. If the number of qualifying fact table rows is 100 million instead of 10 million, the enormous number of sequential touches makes the response time very long even if the index rows were in a small number of slices.

SUMMARY TABLES

Even with ideal indexes, some queries to a fact table of one billion rows will result in far too many touches. The only way to improve performance dramatically then is by using *summary* tables (*query* tables). These are denormalized fact tables. They are feasible when the number of rows is not very high, perhaps only a few million. Because of the denormalization, no joins are needed; all the required columns are in one table.

If weekly sales queries are common, it naturally makes sense to build a summary table for the sales by week. A good summary table for the example we have discussed would have a row per sales by week, item, and store, as shown in Figure 9.9. With three fat indexes, the typical response time would probably be less than a minute. The summary table could be made significantly smaller by storing sales by week and item group, as shown in Figure 9.10. In this case the typical response time might be less than a second.

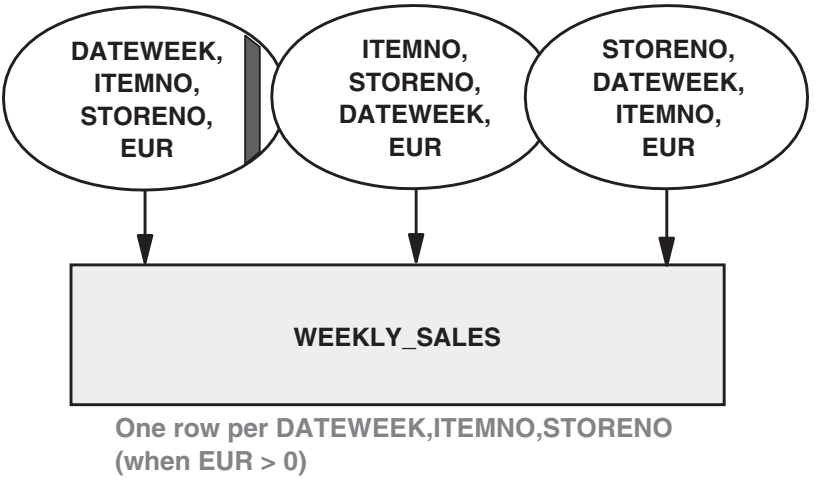


Figure 9.9 Summary table for Weekly_Sales.

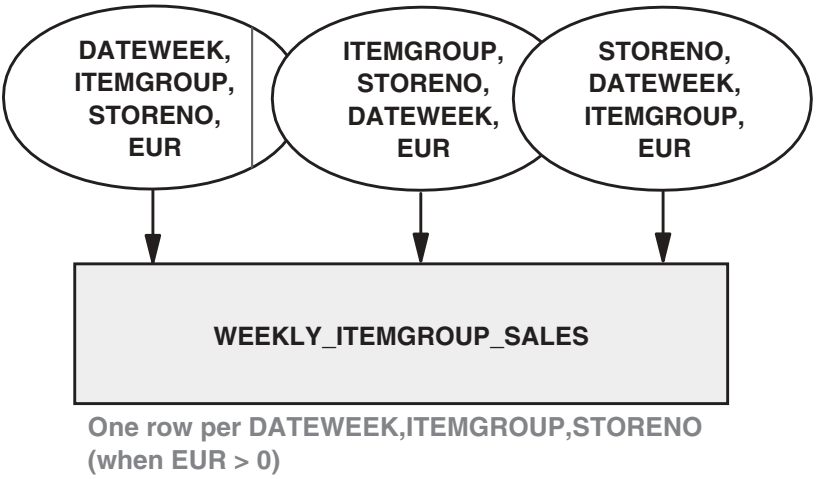


Figure 9.10 Summary table for Weekly_Itemgroup_Sales.

The indexes on summary tables tend to be quite small; the only limiting factor is normally the time it takes to refresh the tables.

As with most things though, a new solution brings with it new problems:

1. If the users generate many different queries, designing summary *tables* may be more difficult than designing the *indexes*. Fortunately, tools are already available that propose summary tables based on traced queries.
2. Summary tables are of little value if the optimizer is not able to choose the correct one; it is unrealistic to expect the end user to specify the

appropriate table in a query. The best optimizers are already learning to access the right summary table, although the SELECT statement actually refers to the fact table. If an optimizer does not have this capability, or if its hit ratio is not good enough, each end user may be forced to access one summary table only. Summary tables that an optimizer is aware of are also known as *automatic summary tables* or materialized views.

Chapter 10

Multiple Index Access

- Index ANDing
- Pitfalls and comparison with the use of a single index
- Use with query tables and fact tables
- Bitmap indexes
- Index ORing
- Boolean predicates
- Comparison with the use of a single index
- SQL Server index joins.

INTRODUCTION

Many DBMSs are able to collect pointers from several indexes on a table, or indeed from several slices of a single index, compare these sets of pointers and then access the rows that satisfy all the predicates in a WHERE clause. This capability is called *multiple index access* or *index ANDing* (index *intersection*) and *index ORing* (index *union*). In this chapter we will consider when these features are useful and how they affect index design.

INDEX ANDING

SQL 10.1

```
SELECT      B, C
FROM        TX
WHERE       A = :A
            AND
            B BETWEEN :B1 AND :B2
ORDER BY    B
```

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

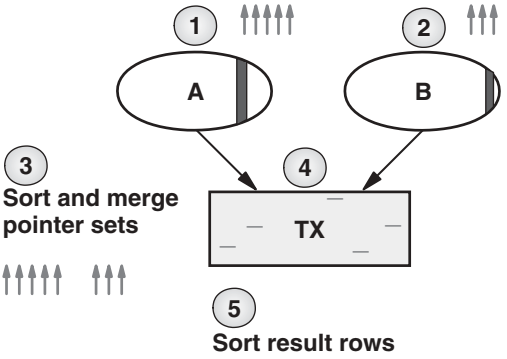


Figure 10.1 Two thin indexes and multiple index access.

Let us consider how the indexes shown in Figure 10.1 could be used to satisfy SQL 10.1.

Index ANDing was important in the days when fat indexes were not very common because of the high cost of disk space. Its introduction had provided an alternative on those occasions when the optimizer had been forced to make a choice between index A and index B, followed by reads to the table rows to evaluate the other predicate. This could imply heavy access to the table.

It would, of course, be more efficient to do the following (the numbers refer to those in Fig. 10.1):

- Collect all the pointers for those index rows that satisfy the corresponding predicate, from both index slices (1) and (2).
- Sort the two sets of pointers into page number sequence (3).
- Merge the two sorted sets of pointers (3).
- Access the table (4) *only* for those rows that satisfy *both* WHERE clauses—one table touch per result row; these table touches will be *faster* than they would have been with random reads because, as the pointers have been sorted by page number, the scan of the table pages will be done using skip-sequential read.

How would this access path compare with that for a single composite index (A, B)? The latter would be preferable to the multiple index access because the number of index touches would be less. The optimizer may choose skip-sequential read anyway if it appears to be a good idea. A fat index (A, B, C) is naturally the most efficient solution for this query.

Index ANDing is not a common access path in operational applications today. If one is detected with EXPLAIN, consideration should be given to eliminating it by using a fat index because index ANDing, using the mechanism described above, has three *severe pitfalls*:

1. The number of sequential touches may be excessive when a simple predicate has a high filter factor.

2. An ORDER BY results in a sort even when several slices are read from an index in which the index rows are in the right order; this is because the sort of the set of pointers destroys any inherent sequence provided by the index.
3. An index only access path is not possible *if only the pointers* to the table rows are collected from the index slices. This issue, a DBMS-specific implementation, will be discussed later.

Let us return to the SELECT statement that looks for large males from a million-row customer table as shown again in SQL 10.2.

If table CUST has *three* single-column indexes (one with column SEX, one with column WEIGHT, and one with column HEIGHT) and if the optimizer decides to collect the matching pointers from these three indexes, the DBMS must touch half a million SEX index rows. In addition, every large male causes a touch to the table. If the DBMS performs multiple index access in the manner depicted above, even making the index SEX fat cannot eliminate the table touches.

SQL 10.2

```
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
            AND
            (WEIGHT > 90
            OR
            HEIGHT > 190)
ORDER BY    LNAME, FNAME
```

Index ANDing with Query Tables

Index ANDing is a valuable facility when SELECT statements with many *different* and *unpredictable* WHERE clauses are generated for a *query table*.

As we saw in Chapter 9, *query tables* are denormalized fact tables. They are feasible when the number of rows is not very high, perhaps only a few million. Because of the denormalization, no joins are needed, all the required columns being in one table. Table CUST in Figure 10.2 is a typical example. It may have tens of columns, having a row length of several hundred bytes, even after compression.

Figure 10.2 shows five search columns, eventual WHERE columns, assuming the customer number is not used as a search column. The six thin indexes together with multiple index access may provide adequate performance for any compound predicate referring to the five search columns. Collecting and sorting the pointers does not take an unacceptable period of time if the index slices to be scanned contain no more than a few hundred thousand rows. Accessing the

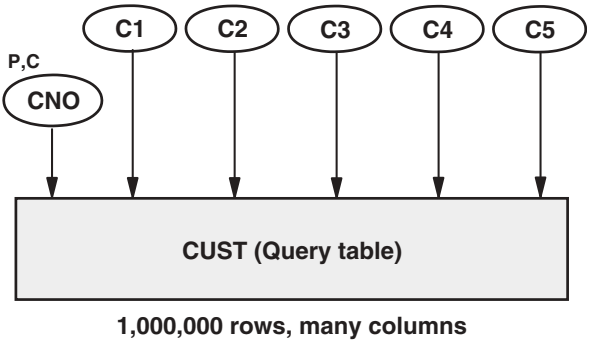


Figure 10.2 Six thin indexes—adequate performance for any WHERE clause.

table rows tends to be the longest component of the response time: The DBMS may have to read thousands of table rows that could be some distance from each other. Fat indexes eliminate these table touches. Designing the fat indexes is analogous to indexing the dimension tables—which we considered in Chapter 9 (the minimum number is one per search column), but the length of the index rows may be an issue.

Multiple Index Access and Fact Tables

It may seem tempting to build several thin indexes, the primary key and the foreign keys, on a large fact table. They would consume much less disk space than the fat indexes recommended in Chapter 9. Unfortunately, however, a multiple index scan with B-tree indexes, together with the scan and sort of a huge number of pointers, would probably be too slow with current hardware if the table has, say, a billion rows.

Multiple Index Access with Bitmap Indexes

Index ANDing and index ORing are much faster with bitmap indexes. Bitmap indexes are more efficient than traditional B-tree indexes for unpredictable queries against very large tables for which there are *no table touches,* for example, SELECT COUNT, or *a small number of table touches*. According to Sasha and Bonnet (7, p. 281), the CIA was among the first users of bitmap indexes:

> *The comparative advantage of bitmaps is that it is easy to use multiple bitmaps to answer a single query. Consider a query on several predicates, each of which is indexed. A conventional database would use just one of the indexes, though some systems might attempt to intersect the record identifiers of multiple indexes. Bitmaps work better because they are more compact, and intersecting several bitmaps is much faster than intersecting several collections of record identifiers. In the best case the improvement is proportional to the word size of the machine because two bitmaps can be intersected word-sized bits at a time. That best case occurs when each predicate is unselective, but all the predicates*

together are quite selective. Consider, for example, the query, "Find people who have brown hair, glasses, between 30 and 40, blue eyes, in the computer industry, live in California." This would entail intersecting the brown hair bitmap, the glasses bitmap, the union of the ages between 30 and 40, and so on.

With current disks, a *semifat B-tree index* is the fastest solution even for CIA-like applications, provided there are not too many range predicates. For the example above, an index starting with columns HAIRCOLOUR, GLASSES, EYECOLOUR, INDUSTRY, and STATE in any order, together with DATE_OF_BIRTH as the sixth column, would provide excellent performance because there would be *six matching columns*: The index slice containing the suspects would be very thin. If the result table contains a hundred rows, the QUBE for the local response time would be

$$101 \times 10 \text{ ms} + 100 \times 0.01 \text{ ms} + 100 \times 0.1 \text{ ms} = 1 \text{ s}$$

Reading long and efficiently compressed bitmap vectors would probably take more CPU time. In addition, single-row inserts, updates, and deletes are slow and disruptive because of locking with bitmap indexes. On the other hand, when the number of potential WHERE clauses is large, it becomes impossible to have good B-tree indexes for every WHERE clause.

INDEX ORING

Index ORing is a more significant feature than *index ANDing*.

Before multiple index access paths were introduced as part of the optimizers' capabilities, which access path would be chosen for the SELECT statement shown in SQL 10.3? If the table had single-column indexes A and B (refer to Fig. 10.3), the optimizer had only one reasonable choice, namely a full table scan because both predicates are now non-BT—the DBMS cannot reject a row when a predicate is evaluated false, as we discussed in Chapter 6.

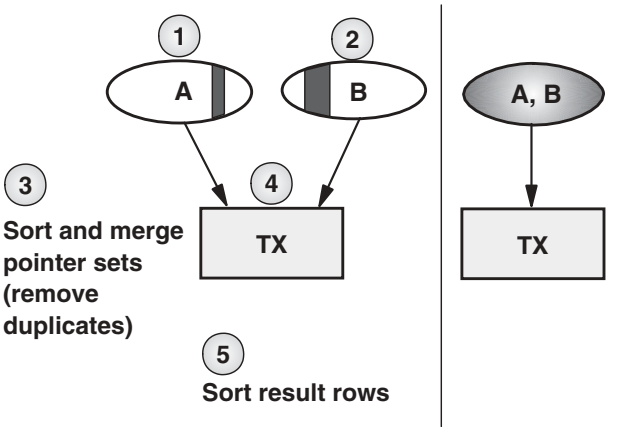


Figure 10.3 Index ORing is good news.

SQL 10.3

```
SELECT     B, C
FROM       TX                          1,000,000 rows
WHERE      A = :A                      FF = 0 ... 0.01%
           OR
           B BETWEEN :B1 AND :B2       FF = 0 ... 0.05%
ORDER BY   B
```

Even a fat index (A, B, C) would not help very much because the DBMS would have to scan the whole index, taking one million sequential touches.

With index ORing, the following takes place:

- Two sets of pointers are collected (1) and (2)—100 sequential touches to A, 500 to B, making 600 in all with the worst input.
- The pointers are sorted (3)—very fast.
- Duplicates are eliminated (3)—very fast.
- The qualifying table rows are read (4)—with the worst input and no duplicates, 600 random touches; 600×10 ms $= 6$ s; this compares with $1{,}000{,}000 \times 0.01$ ms $= 10$ s for a scan of the whole index.

Some time ago, the programming standards at a bank *ruled out the use of OR in a WHERE clause*, even though the optimizer was capable of using multiple index access. Of course, a programmer may not be aware of the effects of non-BT predicates, but there are many cases where a WHERE clause with multiple ANDs and ORs is convenient and performs *well enough* with good indexes: no multiple index access, no sort, index only. During the quick EXPLAIN review, SELECT statements should be checked for multiple index access; inadequate indexes or harmful ORs may be identified, which have to be replaced by UNION or by splitting the cursor.

INDEX JOIN

The SQL Server optimizer is able to generate an access path called an *index join*, which differs from the implementation described above.

This is how Kevin Viers (3, p. 611) describes the SQL Server index join:

> *Another way of using multiple indexes on a single table is to join two or more indexes to create a covering index. In case you don't remember, a covering index is an index that contains all of the columns required for a given query. Consider the following example:*

```
SELECT OrderDate, ShippedDate, count(*)
FROM Orders
GROUP BY OrderDate, ShippedDate
```

Again, the Orders table contains indexes on both the OrderDate and the ShippedDate columns. In this instance, the optimizer will join two indexes using a merge join to return the appropriate result set. By joining the two indexes, SQL Server created the same effect as having one covering index on the table.

By collecting more than just the pointers from the index slices, this implementation avoids the third pitfall of multiple index access, *unnecessary table touches*. However, the first one, *thick index slices*, and possibly also the second one, *unnecessary sorting of the result rows*, remain. To avoid them altogether, a *real* fat index (covering index) is needed; for this query it would be (OrderDate, ShippedDate).

EXERCISES

10.1. Assume the CIA table described in the section Multiple Index Access with Bitmap Indexes has 200,000,000 rows. Estimate the disk space required for (a) bitmap indexes and (b) a semifat B-tree index.
Assume 8 bits per byte. Convert the difference in disk space into dollars per month.

Chapter 11

Indexes and Reorganization

- Physical structure of indexes
- How index rows are accessed by the database management system
- Effect of table row inserts on an index
- Index leaf page splits
- Reorganization
- Serious effect of splits on index scans
- Leaf page split ratio and its prediction using binomial distribution
- Insert patterns
- End of index, random, and hot spots
- Free space recommendations and how to determine reorganization frequencies
- Special cases
- Volatile index columns and long index rows
- Effect of a disorganized index and table on large batch programs
- Effect of leaf page splits on table rows stored in leaf pages with SQL Server and Oracle
- Cost of index reorganization
- Monitoring splits

PHYSICAL STRUCTURE OF A B-TREE INDEX

In previous chapters, the indexes have been described as tables containing a subset of the columns from the tables on which the indexes are built, together with pointers to the table rows. We have assumed that the DBMS *goes directly to the first row of the index slice* defined by the matching columns and stops as soon as it finds an index row that does not have the right values of the matching columns. The diagrams have always shown one index row per table row, even when the index key is not unique. Furthermore, the index rows have always been shown in the sequence defined by the index columns. This mental image is very

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

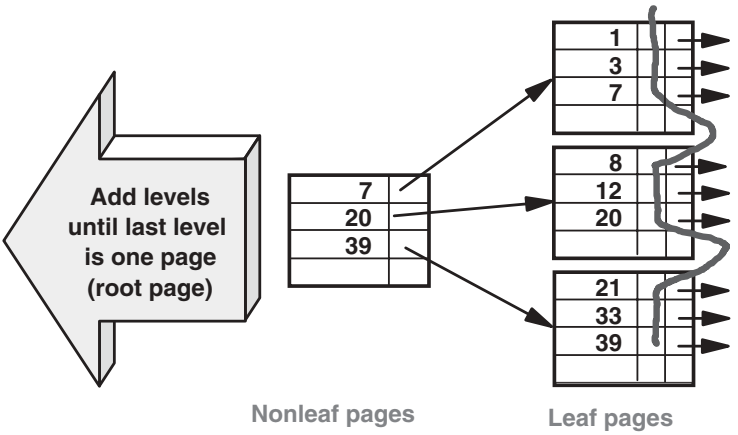


Figure 11.1 B-tree index with two levels.

useful when evaluating an index by counting touches. However, now is the time to discuss the *actual physical structure* of a B-tree index.

As far back as the sixties, the indexes of file and database systems were built as *balanced trees*. The small tree in Figure 11.1 has been chopped down, the root being placed on the left, the leaves on the right.

Index rows are stored in leaf pages. Today, the typical index page size is 4 or 8 kb. The length of an index row is usually the sum of the length of the index columns plus about 10 bytes of control information. If the total length of an index row is 200 bytes, about 40 index rows will fit in an 8K leaf page if the index is unique. In a nonunique index, several pointers may follow each key value; in many DBMS products, these pointers are stored in pointer value sequence. This is to enable the DBMS to quickly find a pointer to be deleted, even if there are a million pointers chained to one key value. We mention this because for historical reasons, some DBAs are worried about the impact of the maximum pointer chain length on delete performance.

HOW THE DBMS FINDS AN INDEX ROW

Assume the optimizer decides to use the index shown in Figure 11.1 for the SELECT shown in SQL 11.1.

SQL 11.1

```
SELECT   COLX
FROM     TABLE
WHERE    COLI = 12
```

Using internal system tables, the DBMS finds a pointer to the root page. After reading the root page, the DBMS knows that it needs to read the second leaf page to find the pointers that relate to key value 12.

One random read brings one page from the disk drive. If the system table pages needed to locate the root page are already in the database buffer pool, the SELECT *may* take *three* random reads: the root page, a leaf page, and a table page. However, with current hardware, the nonleaf pages are likely to stay in the database buffer pool, or at least in the read cache, because they are frequently referenced. Therefore, it is reasonable to assume that only *two* random reads take place; the synchronous I/O time for this is about 2×10 ms $= 20$ ms. The CPU time is much less than this with current processors, so the elapsed time for the SELECT is likely to be about 20 ms also.

When the DBMS needs to read an index *slice*, it reads the first index row as described above. The first index row has a pointer to the next index row. The DBMS follows this pointer chain until it comes to an index row that does not satisfy the matching predicates that define the slice. All index rows are chained together according to the index key.

In SQL 11.2, the SELECT statement is looking for customers who have an LNAME starting with, say, the character string 'JO' and a CITY starting with, say, the character string 'LO'. In this case the DBMS first tries to find a row in the index (LNAME, CITY) with the key value JO padded to the maximum index length with character "A"s as in JOAA ... AA.

When it finds the position for this key value, it reads the following index rows using the chain, until it finds an index row whose key value does not start with 'JO'.

SQL 11.2

```
DECLARE CURSOR112 CURSOR FOR
SELECT     CNO, FNAME
FROM       CUST
WHERE      LNAME BETWEEN :LNAME1 AND :LNAME2
           AND
           CITY BETWEEN :CITY1 AND :CITY2
ORDER BY   FNAME
```

WHAT HAPPENS WHEN A ROW IS INSERTED?

If a table has a clustering index, the DBMS tries to place an *inserted row* in the *table* page to which it belongs, according to the clustering index key value (the home page). If the row does not fit in the home page, or the home page is locked, the DBMS will check the pages close to the home page. In the worst case, the new table row is placed in the last page of the table. Existing table rows are usually left where they are, depending on the DBMS and the type of table

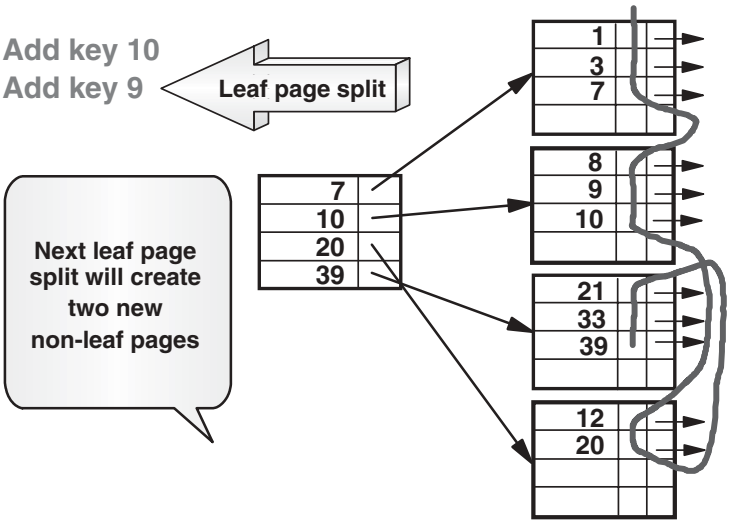


Figure 11.2 Leaf page split is a fast operation.

because otherwise this would mean updating the pointers in *all* the indexes that have been built on the table. When many table rows have been misplaced, the table needs to be reorganized if the table row order is important, which is often the case for massive batch jobs that process several large tables.

As shown in Figure 11.2, the DBMS tries to add the *index row* for an inserted table row in the leaf page to which it belongs, according to the index key value. This index page may not have sufficient free space, however, in which case the DBMS will *split* the leaf page; half of the index rows will be moved to a new leaf page, as close as possible to the page being split, but in the worst case at the end of the index. In addition to the percentage of free space left in *each* leaf page, it may be possible to leave every *n*th page empty when the index is created or reorganized, a good idea if leaf page splits are unavoidable.

When an index has an *ever-increasing* key value, new index rows are added to the last leaf page, which is probably never split. Such an index may not need any free space.

Figure 11.2 shows the index after two additional table rows have been added. The first of these, key 10, was placed in the appropriate leaf page, free space being available. The second, key 9, caused a leaf page split. Any further leaf page splits would require a second nonleaf page to be created, which in turn would require a third-level page to be built.

ARE LEAF PAGE SPLITS SERIOUS?

Splitting a leaf page requires only one *extra* synchronous read, 10 ms. In addition to the two leaf pages, the DBMS must normally update only one nonleaf page, which is probably already in memory or in the read cache.

After a split, retrieving any *single* index row is probably just as fast as before. In the worst case, a split creates *a new level* in the index, but that adds only an insignificant amount of CPU time if nonleaf pages are not read from the disk drive.

Reading an *index slice*, however, *does get slower*. In all our calculations so far, we have assumed that the chain connecting index rows *always points* to the same or the next page, which may be prefetched by the DBMS. This is close to the truth after the index has been created or reorganized, but each leaf page split encountered in a slice may add *an extra two random touches*—one to find the page to which the index rows have been moved, the second to return to the original position in the scan. The first random touch is likely to result in a random read from a disk drive (10 ms).

Let us assume that we need to scan 100,000 index rows. According to the QUBE, this takes

$$1 \times 10 \text{ ms} + 100{,}000 \times 0.01 \text{ ms} = 1 \text{ s}$$

When there are 20 index rows per leaf page (and no leaf pages splits), the DBMS must read 5000 leaf pages. If these pages are adjacent and sequential read speed is 0.1 ms per page (40 MB/s, page size 4K), the I/O time is

$$10 \text{ ms} + 5000 \times 0.1 \text{ ms} = 510 \text{ ms}$$

When 1% of leaf pages have been split, scanning 100,000 index rows is probably degraded by 50 random I/Os (1% of 5000). Even with such an *apparently* small percentage split, the I/O time has doubled:

$$51 \times 10 \text{ ms} + 5000 \times 0.1 \text{ ms} = 1010 \text{ ms}$$

The general formula for the index slice scan I/O time after leaf page splits is

```
(1 + (LPSR × A/B)) × ORIG
```

Where LPSR = leaf page split ratio (number of splits/number of leaf pages)
A = I/O time per random read
B = I/O time per sequentially read page
ORIG = I/O time before leaf page splits

This formula is based on two assumptions:

1. Sequential I/O time is NLP × B (the random I/O to the first leaf page is ignored).
2. Random I/O time is LPSR × NLP × A (one random I/O per leaf page split).

As the ratio A/B increases (which is the current technology trend), the alarm limit for the leaf page split ratio decreases. When A/B = 100 (our assumption), the I/O time *doubles* as LPSR reaches 1%.

Applying the general formula to our example above:

$$\begin{aligned} \text{LPSR} &= 0.01 \\ \text{A} &= 10 \text{ ms} \\ \text{B} &= 0.1 \text{ ms} \\ \text{ORIG} &= 0.5 \text{ s} \end{aligned}$$

The index slice scan I/O time after the leaf page splits becomes

$$(1 + (0.01 \times 10 \text{ ms}/0.1 \text{ ms})) \times 0.5 \text{ s} = 2 \times 0.5 \text{ s} = 1 \text{ s}$$

WHEN SHOULD AN INDEX BE REORGANIZED?

Insert Patterns

Indexes are reorganized to restore the correct physical order, which is important for the performance of *index slice scans* and *full index scans*. Adding index rows may create disorder in a way that depends on the *insert pattern*. It is important to remember that updating a key column means *removing* an index row and *adding* a new one in the position determined by the new index key value. We will see in Chapter 13 that some products support *nonkey index columns*; updating these columns does *not* affect the position of the index row.

The following guidelines for the three *basic insert patterns* discussed below are based on two assumptions:

1. The index is unique.
2. The space for a deleted index row can be used for new index rows *before* reorganization.

1. New Index Rows Added to End of Index (ever-increasing keys)

No free space or reorganization is needed, assuming the DBMS does not split the last leaf page when a new index row has a higher index key value than any existing index row. However, if the index rows *at the front* of the index are *periodically deleted*, the index may have to be reorganized in order to reclaim the empty space (a *creeping index*).

An important exception: If the index rows are variable length, free space is needed to accommodate any increase in the length of the index rows.

2. Random Inserts

We will see later that, as far as free space and reorganization considerations are concerned, short, medium, and long index rows should be treated differently; this is shown in Figure 11.3. The latter is the most difficult case to handle, the former the easiest; the discussion on the medium case will serve to explain why.

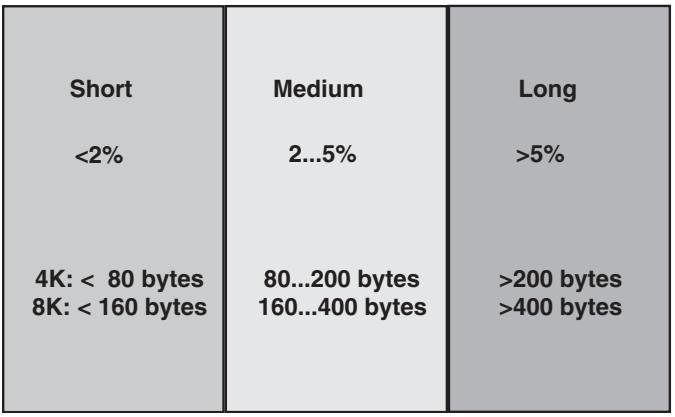


Figure 11.3 Three different free-space treatments.

Short-Cut for Short Index Rows

When the index rows are short, refer to Figure 11.3, choosing P for an index with random inserts is simple:

Basic recommendation: P = 10%; reorganize the index when it has grown by 5%. Thus, P = twice the predicted growth.

With short index rows, this rule keeps the number of leaf page splits to a reasonably low level; LPSR will be less than 1%. The discussion that follows justifies the factor of 2. We will also see why leaf page splits will be more likely and P more difficult to choose, when the index rows are longer.

If a longer reorganization interval is required, P should be set to 20% and the index reorganized when it has grown by 12%. We will discuss at a later stage what would be required if the interval needed to be extended even further.

Difficult Choice for Long Index Rows

When the index rows are long, refer to Figure 11.3, choosing P for an index with random inserts can be quite tricky. This will be addressed later, after the following discussion on medium-sized index rows has been understood.

Choices for Medium-Sized Index Rows

The number of leaf page splits can be predicted when two values, P and X, are known about the state of an index after reorganization:

P% = amount of free space requested in CREATE INDEX

To request 25% free space, a value of 25 would be used for PCTFREE in Oracle and DB2, and a value of 75 for FILLFACTOR in SQL Server. During reorganization (rebuild) and load, index rows are added to a leaf page until the next row would not totally fit above the P% line shown in Figure 11.4.

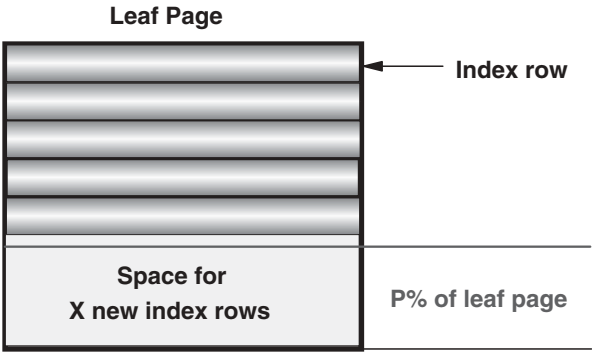


Figure 11.4 Leaf page split—P and X.

X = number of new index rows that can be added to a leaf page after reorganization; our recommendation, as we will see later, will be X = 5.

If P = 10%, leaf page size = 4K and the index rows are 200 bytes in length, 18 index rows will fit above the P% line (90% of 4K) and so X will be 2.

In addition to P and X, we need two variables to describe the growth of an index:

Y = number of index rows added since index reorganization

Z = number of leaf pages immediately after reorganization

The number of leaf page splits can now be predicted by a binomial distribution, assuming the inserts are random. The EXCEL function

```
BINOMDIST(X,Y,1/Z,TRUE)
```

determines the probability of a leaf page *not* being split.

For example, let us assume the following: An index, after reorganization, has 1,000,000 rows and 50,000 leaf pages (Z) with P = 20%; 10,000 new index rows are added per day (a growth rate of 1% per day); the insert pattern is random; old rows are not deleted in the foreseeable future.

As there are 20 index rows per leaf page when 80% full, there is room for five new index rows per leaf page. Therefore X = 5.

Will there be any leaf page splits during the first 5 days following a reorganization [50,000 inserts (Y)]?

```
BINOMDIST(5,50000,1/50000,TRUE) =  0.9994
```

This means that 0.06% of leaf pages will be split during the 5-day period. The number of leaf page splits will be 0.06% of 50,000 leaf pages = 30. The I/O time for an index slice scan will now become (1 + 0.06) × ORIG, with the assumptions listed above. This appears to be acceptable.

After 10 days (100,000 inserts), the proportion of leaf pages that have *not* been split becomes

```
BINOMDIST(5,100000,1/50000,TRUE) = 0.9834
```

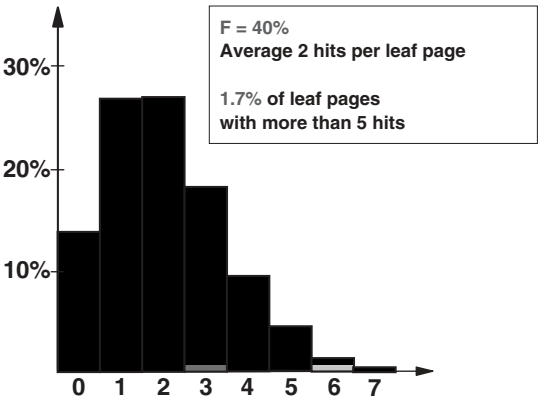


Figure 11.5 Effect of random inserts.

Now, 1.7% of the leaf pages will be split, and the I/O time for an index slice scan will become $(1 + 1.7) \times \text{ORIG} = 2.7 \times \text{ORIG}$. At this point there will be, on average, 2 new index rows per leaf page (100,000/50,000), so 40% (2/5) of the distributed free space will have been used.

It may seem very odd that so many leaf pages have been split ($1.7\% \times 50{,}000 = 850$) when only 40% of the free space has been used. Figure 11.5 shows why this will be so, by showing how the new index rows are likely to hit the leaf pages when the inserts are random.

- Nearly 15% of the leaf pages will not be hit at all
- Over 25% will be hit only once
- 1.2% will be hit 6 times, a split occurring for each leaf page

We will see that the filling factor of the free space (40% above) plays an important role in LPSR prediction. We will call this value F:

$$\text{F} = \text{percentage of the distributed free space that has been used at a particular point in time}$$

When Y new rows have been added in Z leaf pages (each of which had free space for X new rows at reorganization),

$$\text{F} = 100 \times \text{Y}/(\text{X} \times \text{Z})\%$$

In the example above,

$$\text{F, after 10 days} = 100{,}000 \text{ rows}/(5 \text{ rows per page} \times 50{,}000 \text{ pages}) = 40\%$$

The LPSR can be expressed as a function of X and F as shown in Figure 11.6.

This model gives an accurate prediction for leaf page splits when the LPSR is low; after a lot of leaf page splits, say LPSR >10%, the prediction becomes pessimistic because of the increase in the number of leaf pages—each split creates two leaf pages, each having almost 50% free space.

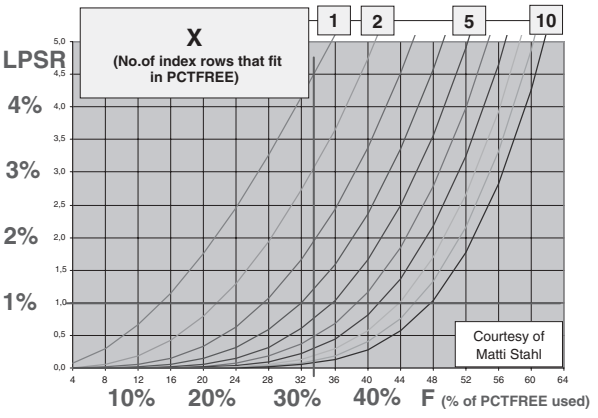


Figure 11.6 LPSR as a function of used free space.

So when should an index be reorganized? Let us assume that we want *the I/O time for an index slice scan to remain below 2 × ORIG*. To achieve this, if A/B = 100 the *LPSR must remain below 1%*: $1 + (0.01 \times 100) = 2$.

Referring to Figure 11.6, when X = 1, LPSR becomes 1% when F is as low as 15%, that is, when only 15% of the distributed free space has been used. This could mean frequent reorganizations.

When X = 10, the LPSR stays below 1% until F becomes 48%. This appears to be much better. However, if the index row length was 5% of the leaf page size (about 200 bytes with 4K pages, 400 bytes with 8K pages), P would have to be 50% to make X = 10. Sequential read would be relatively slow if all leaf pages were only half full.

With the common leaf page sizes currently in use (4K and 8K), X = 5 seems a reasonable compromise between reorganization frequency and the amount of distributed free space required. *We will base our general recommendations on X = 5*. When the index rows are exceptionally long, we will have to consider smaller X values, as we shall see in due course.

Figure 11.6 shows that when X = 5, we should reorganize an index when no more than 36% of the distributed free space has been used, in order to achieve an LPSR below 1%; *for simplicity, we will take a figure of one third,* 33%. But when will this occur? How long will it be before one third of the distributed free space has been used, at which point the index will need to be reorganized to achieve our stated objectives?

As F represents a percentage of the *distributed free space*, we will need to convert this to a percentage of the *actual data*; this is, of course, the non-free-space area. With the figures from our example (P = 20%, F = 33%), the point at which reorganization should take place will be reached when further growth (we will call this the Growth before REORG, G) of:

$$\begin{aligned} F \times (P/(100 - P)) &= 33 \times (20/(100 - 20)) \\ &= 8.3\% \end{aligned}$$

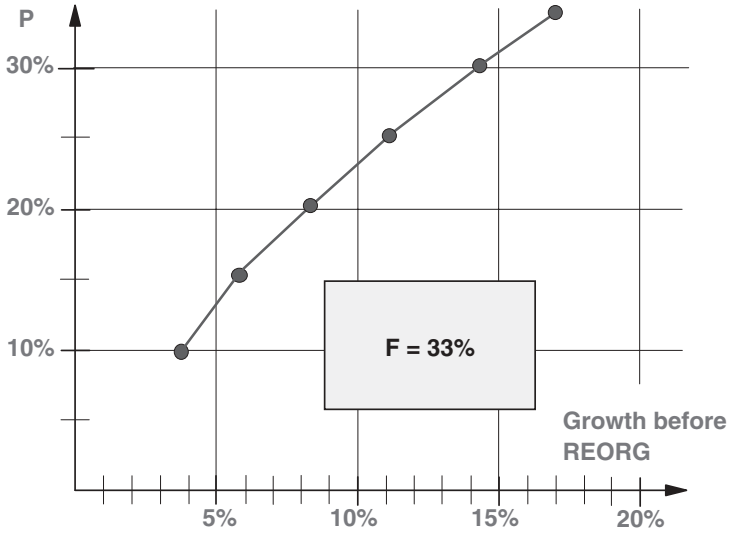


Figure 11.7 Reorganization guidelines.

has occurred. This value, 8.3%, is the growth permissible before the index should be reorganized (to meet the LPSR objective of 1%, which, remember, will keep the index slice I/O time down to no more than twice the ORIG). Values of G for different values of P are shown on the graph in Figure 11.7.

Thus, the Growth before REORG (G) formula or Figure 11.7 can be used, for a given P, to predict how frequently an index should be reorganized. Alternatively, if the reorganization interval has been specified, they may be used to choose a value for P. Let us assume that the index in our example (daily growth 1%) can be reorganized every 2 weeks. With G = 14% (14 days × 1% per day), Figure 11.7 suggests that P should be set to 30%.

Many tables are empty to begin with and then keep growing; the growth rate initially may be very high. Indexes on such tables *need special treatment during the first few weeks:* very high P (even 99%) and a reorganization possibly required every night. Figure 11.7 can be used to predict how the reorganization interval can be extended as the table grows and P is reduced. If the index discussed above grows by 10,000 rows per day, it will have more than 10,000,000 rows in 3 years. Then, with P = 20%, the index reorganization interval can be as long as 80 days: From Figure 11.7, for P = 20%, G = 8%; 8% of 10,000,000 = 800,000 rows, equivalent to 80 days.

In this example, the length of an index row is 4% of the leaf page size (25 rows in total per page), about 160 bytes with 4K pages or 320 bytes with 8K pages. In this case, P must be at least 20% to satisfy the five-row rule. This is not a problem, but supposing the index rows were three times larger (12% of the leaf page)? Leaving 60% of each leaf page empty after reorganization is not an attractive thought; having only 3 index rows per leaf page would make index

slice I/O time quite long, even before any splits. This problem, together with its solution, will be discussed later.

Justification of our Recommendations

Short Index Rows *Basic recommendation:*

$$P = 10\%$$

reorganize the index when it has grown by 5%.

To justify the recommendation, we will consider the borderline case: The length of the index rows is 80 bytes and the leaf page size 4K.

If $4096 - 10 = 4086$ bytes are available for index rows, 51 index rows fit in a leaf page. With $P = 10\%$, each leaf page has 45 index rows after reorganization. Thus, $X = 6$.

When the index has grown by 5%, there are 2.25 new index rows (5% of 45) per leaf page, on average. Thus $F = 2.25/6 = 38\%$.

The curve for $X = 6$ (Fig. 11.6) gives LPSR = 1% when $F = 38\%$.

This is the worst case; when index rows are shorter than 2% of the leaf page size, the LPSR will stay lower.

If a longer reorganization interval is required, P should be set to 20% and the index reorganized when it has grown by 12%. The justification for this is as follows:

With $P = 20\%$, each leaf page has 40 index rows after reorganization. Thus, $X = 11$. When the index has grown by 12%, there are 4.8 new index rows (12% of 40) per leaf page, on average. Thus $F = 4.8/11 = 47\%$. The curve for $X = 11$ (Fig. 11.6) would give LPSR = 1% when $F = 47\%$.

For even longer reorganization intervals, assume LPSR = 1% when $F = 50\%$. Then

$$G = 50 \times (P/(100 - P))$$

Medium Index Rows Basic recommendation: Choose P to make $X = 5$, reorganize when $G = 33 \times (P/(100 - P))$.

We will again consider the borderline case: The length of index rows is 200 bytes and the leaf page size 4K.

P must be $(5 \times 200$ bytes$)/4086$ bytes $= 25\%$ to make $X = 5$. At reorganization, 15 index rows will be placed in each leaf page.

When the index has grown by 11% ($G = 33 \times 25/75$), there are 1.65 new index rows (11% of 15) per leaf page on average. Thus $F = 1.65/5 = 33\%$.

The curve for $X = 5$ (Fig. 11.6) gives LPSR = 0.7% when $F = 33\%$.

This is again the worst case; when index rows are shorter then 5% of the leaf page size, LPSR will stay lower.

Long Index Rows Choose reasonable values for X and P (try to make X at least 2); then find F and G as usual. This will be considered in more detail in the section on Long Index Rows.

Summary of Steps to Determine the Reorganization Frequency

1. For short index rows, P = 10%; reorganize the index when it has grown by 5% (G = 5%). Alternatively, use P = 20% and G = 12%. To make the reorganization interval longer, choose P > 20%; then G = 50 × (P/(100 − P))%.
2. For long index rows, refer to the later section.
3. For medium-sized index rows,
   a. Choose P such that X = 5: P = 5 × index row length/leaf page size.
   b. Using Figure 11.7, determine the value of G and hence the reorganization interval.

Summary of Steps If the Reorganization Frequency Is Specified

1. For short index rows, choose one of the following: (G = 5%, P = 10%), (G = 12%, P = 20%), or (G = 33% and P = 40%). Use the formula P = 100 × G/(G + 50) when G > 33%.
2. For long index rows, refer to the later section.
3. For medium-sized index rows,
   a. Assume LPSR of 1%, X = 5, F = 33%.
   b. Given the value of G, determine the value of P using Figure 11.7.

3. Large Portion of New Index Rows Goes to Small Area (hot spots — not to the end of the index)

Unfortunately, this *hard to manage* insert pattern is fairly common. In index (BO, DATE), for instance (BO = branch office, DATE is ever increasing), the new index rows go to the end of each BO slice. The reorganization interval must be tailored index by index according to the number of leaf page splits and their impact on index slice scan performance.

In index (BO, DATE), the page containing the last row of the largest BO is the hottest page. It splits soon after reorganization. After the first split, the hottest page is half full. For a while, the hottest page contains only rows of the largest BO. The most recent rows for that BO can still be found quite quickly. The situation gets worse when the last pages of many branch offices have been split. In the end, each new index row per BO may go to a different leaf page. It may then take 20 random index touches to retrieve the last 20 index rows per BO.

In a case such as this, a simple solution can provide significant relief. If old rows (low values of DATE) are periodically deleted, the index should *not* be reorganized after the deletes. It is convenient to have empty leaf pages close to each hot spot; cutting the tails makes room for the growing heads.

Making an index, which has a high insert rate and hot spots, *resident* (pinned) in memory is becoming more and more feasible as memory prices are falling and 64-bit addressing makes large buffer pools possible. In addition to the obvious benefit of having no read I/Os, a resident index is not very sensitive to leaf page

splits. As random touches to resident leaf pages take about 0.1 ms instead of 10 ms, a resident index may need no reorganization, even if it has a high insert rate to a small area. If a fat resident index is too expensive, a semifat resident index may be a good alternative solution.

Special Cases

There will be occasions when one or both of the assumptions shown earlier do not apply; the following comments should then be considered.

1. If the index is nonunique, the index keys will be followed by several pointers, one per duplicate key. Many physical index rows may then be as short as 10 bytes or so (a pointer and a few control characters).

   If *all* index key values have a large number of pointers (e.g., SEX index), the free-space requirements are reduced. The index may be in a good shape when 67% (vs. 33%) of the free space per leaf page has been used. If some index key values have only a single pointer, the guidelines for unique indexes should be used to be on the safe side.
2. If a DBMS doesn't free the space of a deleted index row until index reorganization takes place, the gross growth (inserts) should be used instead of the net growth (inserts – deletes) in the formula for determining the reorganization frequency.

VOLATILE INDEX COLUMNS

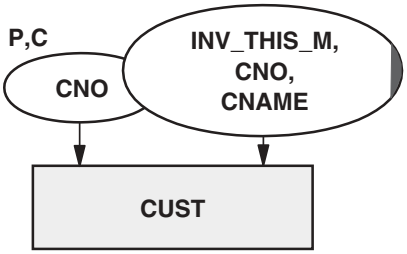
Let us consider an index that starts with an extremely volatile column (INV_THIS_M, CNO, CNAME), where column INV_THIS_M is the amount invoiced this month. With this index, the following SELECT, for instance, would be extremely fast, assuming that the DBMS is able to read the index backwards.

```
TR = 1, TS = 500, F = 500, LRT = 65 ms
```

Figure 11.8 shows the index at the beginning of the month, when the index rows are in CNO sequence. When the first invoice is created, the DBMS must move the related index row to the end of the index. The index row for the customer associated with the second invoice is then moved, according to the invoice amount, to either before or after the previously moved index row. With every new invoice, an index row may move—and the move is always forward. The index will have a lot of leaf page splits close to the end. This is a hot spot problem.

SQL 11.3

```
SELECT     CNAME, CNO, INV_THIS_M
FROM       CUST
ORDER BY   INV_THIS_M DESC
WE WANT 500 ROWS PLEASE
```



INV_THIS_M	CNO
0	4
0	10
0	14
0	15
0	19
0	22
.....	.....

Figure 11.8 A split-bound index—index row moves not random.

If the index is reorganized only once a month, LPSR will get quite high. In theory, the TOP 500 query could require as many as 500 random touches (if every index touch was random), although the actual number of TRs would probably be much less. Real problems would come with SELECTs for which an index starting with INV_THIS_M was *not* a three-star index. Then, one FETCH might need thousands of index touches, many of which would be random when the LPSR is high.

There are at least three feasible solutions for the hot spot problem:

1. Make the index resident; P = 0, no reorganization. If there are 1,000,000 customers, the size of the index may be 60 MB. If the cost of memory is U.S.$1000 per gigabyte a month, the cost of the resident index is $60 per month. However, in 32-bit systems virtual storage constraints may make this solution unattractive.
2. Create a resident semifat index (INV_THIS_MONTH); P = 0, no reorganization. This index does not consume much memory—probably less than 10 MB because it is not unique. As with a normal semifat index, each FETCH requires a table touch. Therefore, this alternative makes sense only if there are many index touches per FETCH (and not many FETCHes): the index touches remain fast no matter how high the LPSR.
3. Reorganize the fat index often enough to keep the LPSR low; perhaps every night. If there are 1,000,000 customers, the elapsed time for index reorganization may be less than a minute.

Volatile index columns *do not* cause split problems if the index row movements are *random*. Consider the following: An index starting with ZIPCODE on table CUST with 1,000,000 rows; 100,000 customers move location to a new ZIPCODE per year while the number of customers grows by only 2%. The LPSR

is likely to stay very low for more than a year when distributed free space is specified, according to the basic recommendations (P at least 10% and large enough for at least five new index rows per leaf page). Reorganization frequency will depend solely on the net growth.

In general, it is important to place the volatile index column as far to the right as possible. For WHERE STATUS = :STATUS AND ITEMNO = ITEMNO, the index on the ORDER table should be (ITEMNO, STATUS ...) and not (STATUS, ITEMNO ...). The moves caused by the volatile column STATUS will be shorter with the first candidate index. LPSR may stay low even if the moves are not random.

Volatile fixed-length index columns following the primary key, for example, INV_THIS_M in index (CNAME, CNO, INV_THIS_M), *do not* create splits because updating such columns does not cause the index row to be moved.

LONG INDEX ROWS

An index row may be considered long if making P large enough for five new rows (X = 5) results in an excessive amount of distributed free space. When P is *not* large enough for five rows,

```
G = 33 × P / (100 - P)%
```

is no longer valid because LPSR >1% when F = 33%. For instance, LPSR = 4.5% if only one row fits in the free space—see the curve for X = 1 in Figure 11.6.

Let us assume that only eight index rows fit in a leaf page. To allow for five new rows per leaf page, P should be 63% (5/8). This would make an index slice read quite slow even without splits. P = 25% (six rows after reorganization, room for two new rows) seems more reasonable. Let us see how the LPSR would grow then.

Following the curve for X = 2 in Figure 11.6, we see that the LPSR reaches 1% when F = 22%. At that point, the average number of new index rows per leaf page is $0.22 \times 2 = 0.44$, so the index has grown by 7% ($100 \times 0.44/6$). If we are able to reorganize the index this often, the LPSR will stay below 1% with P = 25%.

Let us assume that the index rows are even longer, perhaps so long that only four rows fit in a leaf page. The reasonable alternatives are P = 25% (X = 1) and P = 50% (X = 2). In the first case, to keep the LPSR below 1%, the index must be reorganized when F = 15% (Fig. 11.6), that is, when the average number of new index rows per leaf page is $0.15 \times 1 = 0.15$; the index has grown by 5% ($100 \times 0.15/3$). This solution is better than one with P = 50% because there are three, instead of two, index rows per leaf page after reorganization; at this point, the I/O time will be 0.03 ms per index row as opposed to 0.05 ms with P = 50%. If the index cannot be reorganized this often, one of the following alternatives must be chosen:

1. Set P = 50% and reorganize when the index has grown by 22%. At this point, 22% of the free space has been used and the average number of new index rows per leaf page is $0.22 \times 2 = 0.44$; the index has grown by $100 \times 0.44/2$, LPSR = 1%.
2. Leave every *N*th leaf page empty at reorganization if that option is available (as with FREEPAGE in DB2 for z/OS). Now, most leaf page splits are not very harmful; when the other half of a leaf page moves only a few pages away, the split probably does not cause a random I/O. The disk system at least, if not the DBMS, normally reads ahead, so a nearby page is often found in a pool or cache. If P = 25% (enough for one new index row) and every fourth page is left totally empty, there is room for 56 new rows $[(32 \times 1) + (8 \times 3)]$ per 32-page group. The probability of a long-distance split is then quite low even when F = 50%. The reorganization interval required to avoid the index slice read time from doubling is difficult to predict, but it may be more than twice that of the first alternative (in which the amount of distributed free space was slightly higher: 50% vs. $56/128 = 44\%$).

The second alternative seems better in this case. In general, however, when the index rows are not this long, it is probably better to avoid leaf page splits by using a high P value, rather than making the leaf page splits less harmful by using empty pages. The empty leaf pages make sequential read slower, just like the empty space per leaf page.

The two examples may seem unrealistic; are there indexes with rows so long that only eight or four rows fit in a leaf page? Such indexes are rare today, but as hardware evolves and the DBMS limits are increased, they will probably become more common. It is sensible to have some idea of when an index row may be too long from the leaf page split point of view. Also, some products provide the option of storing table rows in leaf pages. Table rows are often long—more than 500 bytes is quite normal.

The benefits derived from fat and semifat indexes rely on the high speed of index slice scans with current hardware, 0.01 ms per TS, according to the QUBE. This benefit is reduced when the index rows are very long. If there are only two rows per 4K leaf page, the I/O time per row may be 0.1 ms/2 = 0.05 ms. More seriously, the number of random index touches begins to increase steeply with inserts when $X < 5$.

EXAMPLE: ORDER-SENSITIVE BATCH JOB

The 12 million touches in the batch program joining tables CUST and POLICY, shown in Figure 11.9 and SQL 11.4, take only 10 min according to the QUBE. At least, that is the figure if the assumptions on which the QUBE is based apply—in this case that the indexes and tables are in perfect order, as they would be following a reorganization. (Note: Because of WE WANT 500 ROWS PLEASE, the optimizer will choose nested loop; this is essential to avoid a sort.

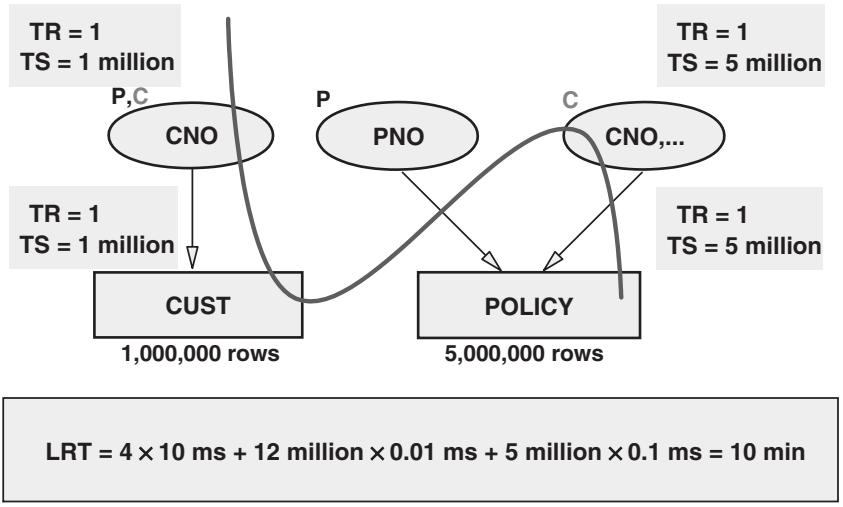


Figure 11.9 Inserts may slow down massive batch jobs.

The program creates a commit point after roughly 500 FETCHes. For the same reason, reading index rows in physical sequence—as with Oracle FAST FULL INDEX SCAN—is not a viable option.)

We will assume that the number of policies increases by 10% in 6 months, perhaps due to a successful marketing campaign. The free space per leaf page and for the table has been set at 10%, quite adequate for a 6-month period when business was slow. As a result, many leaf page splits will occur because while the *average* growth rate per leaf page is 10%, *half of the leaf pages* have a growth rate *exceeding* 10%.

SQL 11.4

```
DECLARE CURSOR113 CURSOR FOR
SELECT     many columns
FROM       CUST, POLICY
WHERE      CUST.CNO = POLICY.CNO
ORDER BY   CNO

OPEN CURSOR113
FETCH while...
CLOSE CURSOR113
WE WANT 500 ROWS PLEASE

commit point roughly every 500 FETCHes
```

Let us take a closer look at the state of index (CNO, ...) on table POLICY after 500,000 inserts. Assume the index rows have a fixed length of 150 bytes and that 4086 bytes (4096 − 10) are available for index rows per leaf page. With P = 10%, 3677 bytes (0.9 × 4,086) can be used at reorganization, enough for 24 index rows (24 × 150 bytes = 3600 bytes) but not for 25 index rows. As 27 index rows fit in 4086 bytes (27 × 150 bytes = 4050 bytes), there is space for 3 newcomers per leaf page. The actual amount of distributed free space is thus 3/27 = 11%; we almost always get a little more free space than we request. Now, the number of leaf pages needed to store 5,000,000 index rows is 5,000,000 rows/24 rows/page = 208,000. From this we can predict the effect of the inserts.

After 500,000 inserts, the average number of new index rows per leaf page is

```
500,000 rows / 208,000 pages = 2.4
```

Therefore, 80% (2.4/3) of the distributed free space has been consumed. Assuming the inserts are random, 22% of the leaf pages will have been split because

```
BINOMDIST(3,500000,1/208000, TRUE) = 0.78
```

If the normal assumptions (A = 10 ms, B = 0.1 ms, etc.) are valid, the I/O time for a full index scan will now be 23 times as long as immediately after reorganization (208,000 × 0.1 ms = 21 s):

```
(1 + (LPSR ×  A/B)) × ORIG = (1 + (0.22 × 100)) × 21 s
= 23 × 21 s
= 8 min
```

The actual I/O time would be less than 8 min because the binomial distribution model becomes pessimistic after a large number of splits. In any case, the high LPSR should have been revealed by a weekly exception report (more about this later in this chapter) long before the index becomes so badly disorganized.

Let us now apply the guidelines, assuming that we expect 10% growth before reorganization. Figure 11.7 proposes that P should be set to 23% when G = 10%. Now, no more than 20 index rows fit in 0.77 × 4086 bytes = 3146 bytes (21 × 150 bytes = 3150 bytes), leaving space for 7 new rows. With X = 7 and F = 29% (2/7), Figure 11.6 predicts LPSR = 0.1%, so the I/O time for a full index scan will be quite short after 10% index growth:

```
(1 + ( LPSR ×  A/B)) × ORIG = (1 + (0.001 × 100)) ×
   25 s
= 1.1 × 25 s
= 28 s
```

This result may seem too good to be true. Weren't the recommendations based on LPSR = 1%? Well, the starting point was LPSR <1% and the actual LPSR will often stay well below 1% for two reasons:

1. The basic recommendation was derived assuming that *5* new index rows fit in the free space defined by P (X = 5). In our example X = 7.

2. At reorganization, the P is rounded up. In our example, 21 index rows *almost* fit in a leaf page when P = 23%.

Reorganizing the index at this point is not necessary, but it is not a bad idea either; the curve for X = 7 in Figure 11.6 shows that performance is beginning to deteriorate—the curve is becoming steeper at this point.

This example illustrates how dramatically the I/O time for a full index scan may grow when index reorganization is long overdue, which may happen if a table grows faster than expected and adequate exception monitoring is not in place. The elapsed time of the batch program does not grow in the same proportion because of the high CPU time (QUBE: 500 s) required by the 5 million FETCH calls. It is the SELECT calls that choose a few index rows out of a thick index slice that suffer most from a high LPSR; the elapsed time of such calls consists mostly of I/O time.

Table Disorganization (with a Clustering Index)

The POLICY table, with P = 10% grew by 10%, just like its indexes. How does that affect the elapsed time of the batch job?

Almost all the POLICY *table pages* become full after 500,000 inserts (growth 10%) because P being 10% means that the distributed free space is used up when the table has grown by

```
P/(100 - P) = 11%
```

and because (real) table pages are never split; inserted table rows that do not fit in the home page (where they should be inserted) are normally placed in a near-by page. Table rows that are stored in *leaf pages* are discussed in the next section. If the DBMS places the table rows close to the home page when the latter is full, the performance of the table access does not at first deteriorate dramatically. This is because the recently accessed pages will be in the buffer pool or disk cache. However, when all the table pages close to the home page are full, the new row will probably go at the end of the POLICY table. Each row that is placed a long way from its home page will probably add a random I/O. If only one new table row fits in the free space per page (10%), the table will be in a terrible state after 500,000 inserts. The number of additional random touches will probably be more than 100,000 (assuming more than 100,000 inserted table rows are a long way from the home page); the elapsed time of the batch program will increase by more than 15 min. The table should have more free space, at least 20%, to survive growth of 10%.

If the DBMS does not support a clustering index, the POLICY table must be reorganized frequently because new POLICY rows go to the end of the table and each INSERT adds 10 ms (one random touch) to the elapsed time of the batch program. After 10,000 new rows, the POLICY table will have grown by only 0.2%, but the time for a table scan via the foreign key index will have grown from 50 s (1×10 ms $+ 5{,}000{,}000 \times 0.01$ ms) to 150 s ($10{,}000 \times$

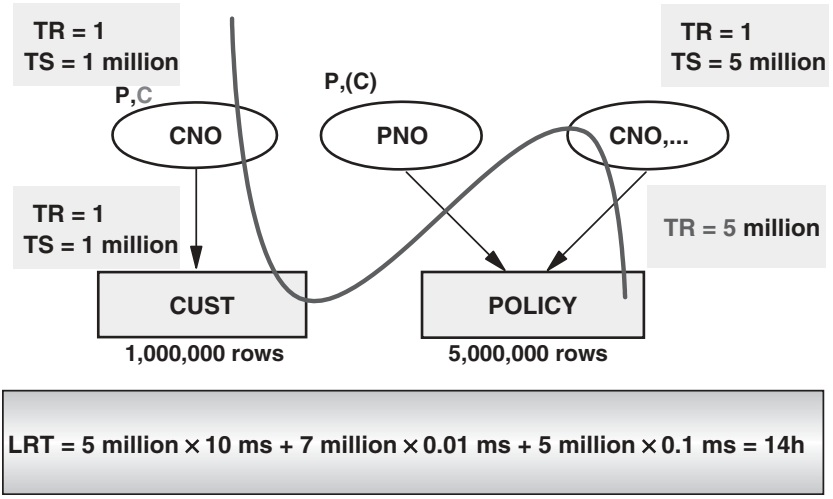


Figure 11.10 Massive batch job slow already after reorganization.

10 ms + 5,000,000 × 0.01 ms). To avoid such frequent reorganizations of the POLICY table, columns should be added to index (CNO, ...) to prevent table touches, or the table rows should be stored in that index if the DBMS supports the option. More about this later.

Table Disorganization (Without Clustering Index Starting with CNO)

Figure 11.10 shows what would happen if the POLICY table rows are *not* in CNO sequence; either PNO is the clustering index as shown or perhaps there is *no* clustering index, and the table rows have not been sorted in CNO sequence at reload time.

The 12 million touches now take 14 h even *before* the marketing campaign. Now, making the foreign key index (CNO, ...) fat for the batch program makes a *huge* difference. In general, the order of the table rows does not matter as long as the access path is index only.

TABLE ROWS STORED IN LEAF PAGES

Real table pages never split, but table rows stored in leaf pages do move when the leaf pages split. Do we need to worry about this?

SQL Server

In SQL Server 2000, the table rows are stored in the leaf pages of the clustered index if a table has a clustered index. Let us assume that the primary key index

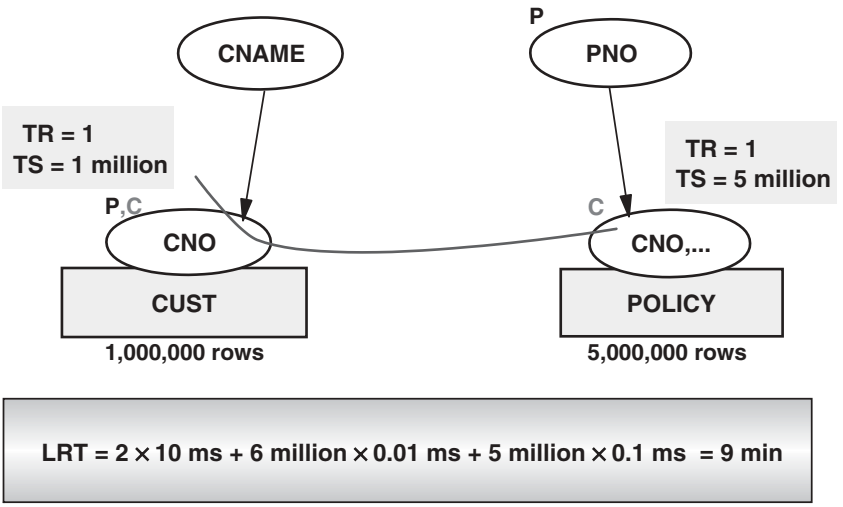


Figure 11.11 Table rows in leaf pages—SQL Server.

(CNO) is the clustered index of the CUST table and the foreign key index (CNO, . . .) is the clustered index of the POLICY table—refer to Figure 11.11. This eliminates 6,000,000 touches—just like fat indexes for the batch program—and saves disk space too. INSERTs to CUST and POLICY are about 10 ms faster because there is no table page to update.

The nonclustered indexes point to the table rows using the *clustered index key*. The indirect pointers add a small overhead (SELECT. . .WHERE PNO = :PNO, for instance, must go through two sets of nonleaf pages), but they prevent a massive pointer update operation when a leaf page split moves table rows. The leaf page splits in the clustered index have the same effect as with any index: Index slice read time becomes longer. LPSR may grow steeply when table rows are long—if the clustered index key is not ever increasing.

Oracle

In Oracle 9i, the implementation of storing table rows in leaf pages (Index-Organized Table) differs in three ways:

1. Only the primary key index can be chosen for the table rows.
2. There are two pointers in the other indexes: a direct pointer and the primary key. The direct pointer is not updated if a table row moves as a result of a split. The direct pointer is used as a first guess; if the table row is no longer at that address, the nonleaf index pages of the primary key index are used to access the table row.
3. There is an option to store only N first bytes of the table rows in the leaf page; the rest is stored in an overflow area. This reduces the need for

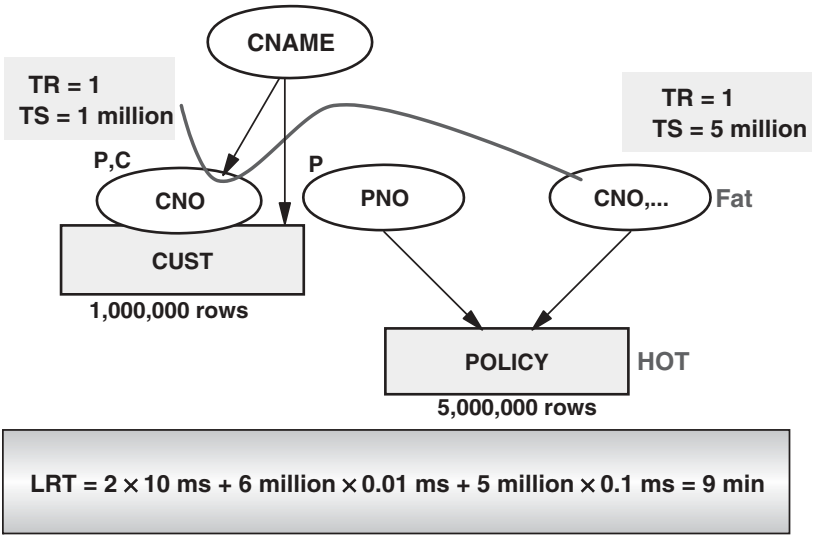


Figure 11.12 Table rows in leaf pages—Oracle.

free space per leaf page but access to the columns in the overflow area becomes slow.

Figure 11.12 shows how the CUST table could be implemented as an Index-Organized Table. This saves one million touches in our batch program, and the index is fat for any query with WHERE CNO = :CNO. The foreign key index on table POLICY should be made fat for the batch program to prevent table touches and very frequent organizations of the POLICY table. Because index (CNO, ...) is not the primary key index, the table rows cannot be stored in it.

Let us assume 8K pages (blocks) and random inserts to the CUST table. Without using an overflow area, 6 CUST rows could be stored per leaf page, with free space for 2 new rows (X = 2). To keep LPSR below 1%, the CNO index should then be reorganized when 20% of the free space has been used, F = 20% in Figure 11.6. That point is reached when, on average, 0.4 new CUST rows (F × X = 0.2 × 2) have been added per page, which corresponds to a 7% growth (0.4/6) of the CUST table. More frequent reorganizations may be justified to keep the success ratio of the direct pointer (the first guess) close to 100%.

COST OF INDEX REORGANIZATION

An index can be reorganized in many ways:

1. Full index scan (random touches and no sort *or* sequential touches and sort)
2. Full table scan (like CREATE INDEX; sequential touches and a sort)

According to the QUBE, the elapsed time per one million rows is *4 min* for the second alternative (2 million sequential touches, 2 million SQL calls, 1 million rows sorted), but many utilities are faster because they bypass the application programming interface. The sort phase is often the longest component, and it consumes a lot of CPU time (e.g., 10 s for one million rows).

The lock waits created by index reorganization are product and option dependent. With a simple utility, the whole table may be S locked (updaters have to wait) while the table or the index is being scanned. The lock wait time is much shorter if the utility saves the updates performed during the scan and applies them before the sort phase.

Large indexes may have to be reorganized at inconvenient hours. In the worst case, locking problems make frequent reorganizations impossible. In this case, some volatile indexes may have to be made resident (pinned in memory).

SPLIT MONITORING

The reorganization characteristics of certain selected indexes, at least those on large tables that have a lot of inserts and updates, should be predicted as we have shown earlier. For the majority, however, the decision to reorganize an index can be based on monitoring.

The number of leaf page splits per index can be monitored explicitly or simply determined by observing the number of leaf pages. When the inserts are random and pages are not left empty at reorganization, the LPSR (the number of leaf page splits divided by the number of leaf pages) is adequate to identify reorganization candidates. The number of *long* leaf page splits divided by the number of leaf pages—let us call this ratio LLPSR—is a better indicator (fewer false alarms) because *short* leaf page splits seldom create random touches for an index slice scan. The LLPSR is reported by DB2 for z/OS V7 Real Time Statistics, for instance:

```
(100 × REORGLEAFFAR / NACTIVE)
```

If the DBMS splits the last leaf page when a new index row is added at the end of the index, indexes with ever-increasing keys should be manually excluded from the list of reorganization candidates if the LLSPR is not available.

Let us assume a scenario with 2000 tables and 5000 indexes. Most indexes are created with $P = 10\%$ (the default in Oracle and DB2); the indexes of read-only tables have $P = 0$, and selected indexes of fast growing tables have higher, tailored P values and reorganization schedules. Other indexes are reorganized once a year.

If there are many inserts and updates, the state of the indexes should probably be checked weekly. The most important report should show all indexes with LPSR $>1\%$ (or LLPSR $>1\%$ if this facility is available); but supposing this report contains hundreds of indexes? The reported indexes are only suspects; they *may* need reorganization, but it is quite possible that most of them do not; it depends on how they are used, for example:

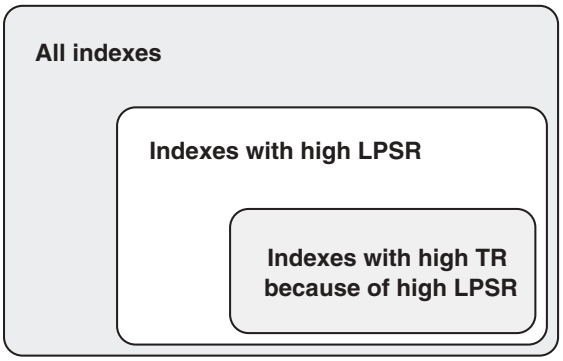


Figure 11.13 Candidates for index reorganization.

- If the index is always used to find a single index row, a high LPSR does not matter.
- If the scanned index slices are always thin, say less than 1000 rows, a high LPSR may not matter too much.

If index reorganization is disruptive, we may have to consider an index to be innocent until it is proven guilty. Split-sensitive indexes can be found with the help of an exception report that shows the profile of slow transactions or SQL calls. If a SELECT takes a long time because of a large number of *random index page reads,* the reason is probably one of the following:

1. Non-BJQ join with nested loop
2. High LPSR and index slice scan

Thus, we may either reorganize all indexes that may benefit the larger subset in Figure 11.13 or only those that are *known* to benefit the smaller subset. The first approach may be reasonable if the reorganizations do not cause significant contention (lock waits, CPU queuing, disk queuing) and with a company-owned computer so that there are no external charges for CPU time.

The weekly split report should include, if the information is available, the number of inserts since the last reorganization; again DB2 for z/OS V7 RTS: REORGINSERTS, is a fine example; this helps to improve the reorganization schedule. If the LPSR growth is high compared to the number of inserts, higher than that predicted by the binomial distribution, the index may have hot spots. Tailored solutions, such as pinning the index in memory, may then be necessary.

SUMMARY

Index reorganization management is a controversial topic. Donald K. Burleson (9) concludes a recent article on the topic with the following thoughts:

The Debate Continues

Today, a battle is raging between the "academics" who do not believe that indexes should be re-built without expensive studies and the "pragmatists" who

rebuild indexes on a schedule because their end-users report faster response times.

To date, none of the world's Oracle experts has determined a reliable rule for index rebuilding, and no expert has proven that index re-builds "rarely" help. Getting statistically valid "proof" from a volatile production system would be a phenomenal challenge. In a large production system, it would be a massive effort to trace LIO from specific queries to specific index, before and after the rebuild.

- **Academic approach**—*Many Oracle experts claim that indexes rarely benefit from rebuilding, yet none have ever proffered empirical evidence that this is the case, or what logical I/O conditions arise in those "rare" cases where indexes benefit from re-building.*
- **Pragmatic approach**—*Many IT managers force their Oracle DBAs to periodically re-build indexes because the end-user community reports faster response times following the re-build. The pragmatists are not interested in "proving" anything, they are just happy that the end-users are happy. Even if index re-building were to be proven as a useless activity, the Placebo effect on the end-users is enough to justify the task.*

It is clear that all 70 of the index metrics interact together in a predictable way, and some scientist should be able to take this data and reverse-engineer the internal rules for index rebuilding, if any actually exist. For now, the most any Oracle professional can do is to explore their indexes and learn how the software manages the b-tree structures.

While the placebo theory is hard to deny, we believe that there are many indexes in the world that should be reorganized (rebuilt) periodically to realize the full benefit of fast sequential read. Many indexes grow; some indexes start with a volatile and ever-increasing column. Reading thick index slices is becoming more and more common, as we have seen throughout this book.

The static metrics like LPSR and LLPSR can only divide the indexes into two sets: those that certainly do *not* need a reorganization and those that *may* need it. The real need can only be determined by estimating or monitoring *the number of random leaf page reads per SQL call.* Then, when an SQL call that reads a thick index slice is found, the effect of a reorganization is easy to determine.

Index growth should be predicted when an index is designed. Furthermore, it is important to remember that when index rows are long compared to the leaf page size, leaf page splits begin to occur when only a small part of the distributed free space has been used. This may set an upper limit to the number of columns that should be added to an index; sometimes it is better to create a new index. The sensitivity of the leaf page split ratio to the length of the index row also reduces the attraction of storing table rows in an index.

Figure 11.14 summarizes the most important issues with regard to free space and reorganization for indexes; our recommendations for both short and medium-sized index rows are shown, assuming the rows are inserted randomly. We have already seen that specific recommendations cannot be provided for long index rows.

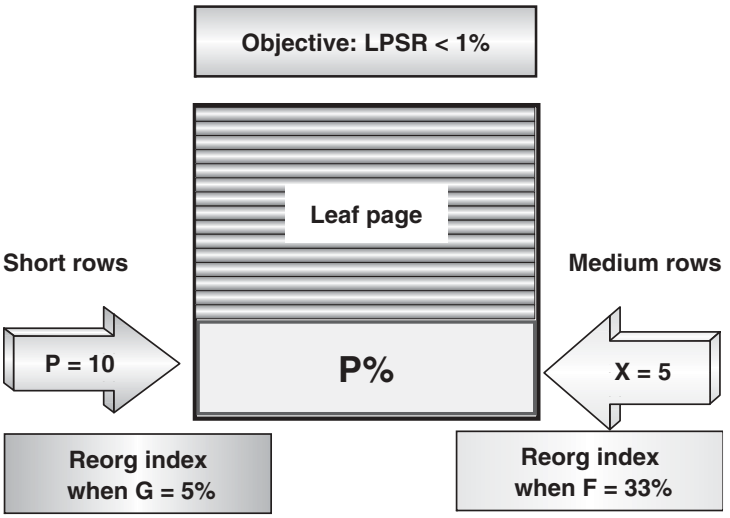


Figure 11.14 Free space and reorganization summary.

Included also are perhaps the two most important characteristics that influence these recommendations. It is crucial that the role of the LPSR is fully understood and that very small values will begin to affect index scan performance; our 1% objective will no doubt come as a surprise, even to many experienced DBAs. That this limit will be breached when relatively small amounts of the free space has been used will also cause some surprise; as will suggestions of reorganization when perhaps only one third of the free space has been used.

Chapter 12

DBMS-Specific Indexing Restrictions

- Restrictions applied to indexes by relational database management systems
- Maximum number and length of the index columns
- Variable-length columns
- Maximum number of indexes per table
- Maximum size of an index
- Index locking
- Index row suppression with null values

INTRODUCTION

Index design is, to a large extent, independent of the DBMS; by far the most important decision is which table columns should be copied into the index. There are, however, a few restrictions that are DBMS and version specific.

NUMBER OF INDEX COLUMNS

The maximum number of columns that may be copied to an index varies between 16 and 64. Not everybody sees this as a problem. Gulutzan and Pelzer (1, p. 231) presents a surprising recommendation:

> *The general recommendations that apply to all DBMSs, then, are:*
>
> - *Use up to 5 columns in a compound index. You can be sure that the DBMS will allow at least 16, but 5 is regarded as the reasonable maximum by some experts.*

Nobody should drink more than five cups of coffee a day because some doctors consider this a reasonable maximum!

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

TOTAL LENGTH OF THE INDEX COLUMNS

The total length of the columns that may be copied to an index has an upper limit that depends on the DBMS. As fat indexes are becoming more popular, this limit seems to be increasing in new versions. For instance, in DB2 for z/OS V7 the upper limit was 255 bytes, but in V8, which became available in 2004, the limit was increased to 2000 bytes. The next practical limit is the size of the index pages. As discussed in Chapter 11, leaf page splits may start to happen soon after reorganization if the free space per leaf page is not large enough for at least two new index rows. If inserts do not go the end of the index, index rows that are longer than 20% of leaf page size (800 bytes with 4K pages and 1600 bytes with 8K pages) may imply frequent reorganization.

VARIABLE-LENGTH COLUMNS

If the DBMS pads variable-length columns to the maximum length, index rows in a fat index can easily become much too long. In DB2 for z/OS V7, for instance, fat indexes with VARCHAR columns are not at all common for this reason, but V8 has an option NOT PADDED in CREATE INDEX.

Many products use VARCHAR columns for nonnumeric columns; SAP is a well-known example. The index tuning potential with such products is fundamentally increased if variable-length columns stay variable length when copied to an index.

NUMBER OF INDEXES PER TABLE

Many products either have no upper limit or the limit is so high that it does not matter. SQL Server 2000, for instance, allows up to 249 unique nonclustered indexes or constraints per table and 1 clustered index. The time taken by the access path selection process increases according to the number of indexes, but this is significant only if the cost estimates are done at each execution.

MAXIMUM INDEX SIZE

Typical upper limits are several gigabytes, and these are being increased on a continuing basis. Large indexes are normally partitioned, as are large tables, in order to minimize the cost of running maintenance utilities and to spread the index over several disk drives or RAID arrays.

INDEX LOCKING

If the DBMS locks an index page or a part of an index page, for example, a subpage, from the time of the update to the commit point, the index page or

subpage can easily become a bottleneck as the inserts will be serialized. SQL Server 2000, for instance, does this, but it is probably not a problem if the size of the lock acquired is a row.

DB2 for z/OS uses latches to ensure physical integrity of index pages. This is essentially a bit set in the database buffer pool when a page in the pool is latched and reset when it is unlatched. A page is latched only for the time taken to read or modify it, a fraction of a microsecond with current processors. Data integrity is ensured by locking the table page or the table row that the index row is pointing to (*data only locking*). If the program is making a modification to a table row or a table page, these locks are not released until the commit point.

Several years ago DB2 locked the modified *index* page or subpage until the commit point. To avoid the resulting long lock waits, it was common practice to avoid ever-increasing keys, by changing a time stamp value from hh.mm.ss to ss.mm.hh, for instance. With latches (or index key locking), inserts at the end of an index are not likely to cause queuing unless the insert transaction rate is several thousand per second.

Avoiding ever-increasing index keys is another myth that is fading away very slowly. With current implementations, an index with an ever-increasing key is actually *good* for performance: Often there will be no synchronous reads because the last leaf page is likely to stay in the database buffer pool. Furthermore, if the index rows do not move or grow, there is no need for any distributed free space or a reorganization.

INDEX ROW SUPPRESSION

Oracle indexes do not always have an index row for each table row. Rows with nulls in *all* key columns are not indexed. As this is a sensitive issue, we will again quote from Gulutzan and Pelzer (1, p. 251):

> *The first and most notorious exception to the rule is that Oracle—and Oracle alone—refuses to store NULLS. Oh, the DBMS will put Nulls in compound keys if other columns are NOT NULL, but Oracle does not have much use for NULL keys. So if you have a table (Table1), containing two columns (column1.column2), and column 2 is indexed, then*
>
> - *Oracle can do:*
>   `INSERT INTO Table1 VALUES (5, NULL)`
>   *much more quickly, because the DBMS does not have to update the index at all.*
> - *Oracle can do:*
>   `SELECT * FROM Table1 WHERE column2 < 5`
>   *a bit more quickly, because the index on column2 has fewer keys and therefore might have fewer levels.*
> - *Oracle cannot do:*
>   `SELECT * FROM Table1 WHERE column2 IS NULL`

> *quickly because the DBMS can't use an index for any case where column2 IS NULL might be true.*
>
> *That is a good trade off. Unfortunately, it causes an attitude difference between Oracle programmers and everybody else: Oracle programmers will think nullable columns are good while programmers for other DBMSs will think that NOT NULL leads to performance improvements.*

The Oracle implementation indeed enables very small indexes with low maintenance cost in some cases; we will discuss an example in the next chapter. However, it also creates situations where an index is not utilized as we would expect (IS NULL, sort avoidance, index only). Fortunately, index row *suppression* can now be *suppressed* by creating a function-based index using the null value built-in SQL function NVL:

```
create index cust_3 on cust (nvl(fname, 'null'),
   lname, cno, city)
```

This means that the index column FNAME will never have null values—when the table column is NULL the corresponding index column will contain a character string 'null'; the predicate FNAME = 'null' will now read an index slice.

DBMS INDEX CREATION EXAMPLES

It is interesting to compare the index characteristics available across the various database management systems commonly in use. The permissible values and restrictions, together with other indexing options available, which are discussed in the following chapter, follow by means of examples of their creation.

DB2 for z/OS

CREATE [UNIQUE] INDEX index_name [CLUSTER]
ON table (column1 [DESC], column2 [DESC]...)
PCTFREE n, FREEPAGE n

Pagesize 4K
Max 64 index columns
Max 2000 data bytes (V7: 255 bytes)

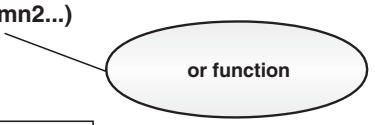
DB2 for LUW

```
CREATE [UNIQUE] INDEX index_name [CLUSTER]
ON table (column1 [DESC], column2 [DESC]...)
[INCLUDE (columna, columnb...)]
PCTFREE n
```

Pagesize 4K, 8K, 16K or 32K
Max 16 index columns
Max 1024 data bytes

Oracle

```
CREATE [UNIQUE] INDEX index_name
ON table (column1, column2...)
PCTFREE n
```

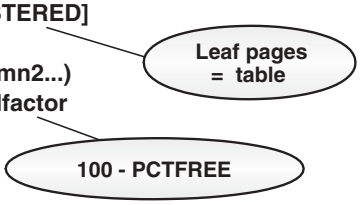


Option: Index-organized table
Leaf pages = table

Pagesize (DB_BLOCK_SIZE) 2K, 4K, 8K, 16K, 32K or 64K
Max 32 index columns
Index row max 40% of DB_BLOCK_SIZE

Microsoft SQL Server

```
CREATE [UNIQUE]
[CLUSTERED|NONCLUSTERED]
INDEX index_name
ON table (column1, column2...)
WITH  FILLFACTOR = fillfactor
```



Pagesize 8K
Max 16 index columns
Max 900 data bytes

Chapter 13

DBMS-Specific Indexing Options

- Index row suppression and exceptional conditions
- Additional index columns after the index key
- Constraints to enforce uniqueness
- Reading indexes in both directions
- Index key truncation
- Function-based indexes
- Index skip scan
- Block indexes
- Data-partitioned secondary indexes

INTRODUCTION

The number of indexing options is steadily increasing over time; this chapter covers the more important options for traditional indexes. In addition, there is an increasing number of application-specific index structures, such as those for geographic databases.

The indexing options available for several DBMS products were discussed in the previous chapter.

INDEX ROW SUPPRESSION

In nonrelational DBMSs index row suppression was a common, and commonly used, option. The creation of an index segment or an index record could be avoided if the indexed field had a given value, typically, blank or 0; furthermore, exit programs could be written that would make more complex decisions. Perhaps the pressure to reduce the size of indexes is less today because this feature is not implemented in many relational products.

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

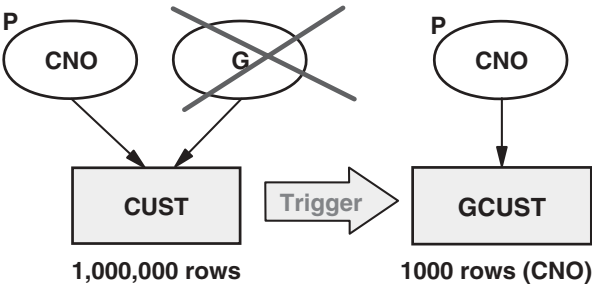


Figure 13.1 When the DBMS does not support index row suppression.

In fact, the feature *is* useful in indexes that are created to find *exceptional* conditions, not because of the disk space savings but to eliminate unnecessary index maintenance. A common circumvention is to build a small index-like table maintained by a trigger, as shown in Figure 13.1.

Gold customers, one customer out of every thousand, have a value 1 in column G, whereas ordinary customers have a value 0. If the DBMS supports index row suppression, it would probably be used for index G. Otherwise, to reduce the maintenance overhead of such an index, a table, GCUST, could be created containing the primary keys of all the gold customers.

This index-like table, just as the index for customers with G = 1 that would have been created if index row suppression had been allowed, is maintained only in the following events:

- A gold customer is added to table CUST.
- A gold customer is removed from table CUST.
- An ordinary customer is promoted to a gold customer.
- A gold customer is downgraded to an ordinary customer.

With Oracle, index suppression can be implemented with NULL values, as we saw in Chapter 12. In the example above, ordinary customers could have a NULL value in column G instead of 0. Performance could be better than with an index-like table (no join), and there is no need to create a trigger to maintain the table GCUST. However, using NULL to represent a specific value is not a recommended practice; it may lead to errors in the long run.

ADDITIONAL INDEX COLUMNS AFTER THE INDEX KEY

Index keys determine the position of an index row in the chain that connects all the rows of an index. When the key value of an index row is updated, the DBMS removes the index row and replaces it in the new position. In the worst case, the index row is moved to another leaf page.

DB2 for LUW allows a split of the columns listed in the CREATE INDEX into two groups: *key columns* and *nonkey columns* (option INCLUDE in CREATE INDEX, as seen in Chapter 12). For instance, in index (A, B, C, D), the index

key may be A, B while columns C and D are nonkey columns. Now, updates to columns C and D will not move the index row. This may save one random read from the disk drive (10 ms) per updated table row. Note, however, that if columns A and B make the index unique, updating columns C or D would not move the index row to another leaf page even if these were key columns.

Another important benefit is the reduction in the number of nonleaf pages: only the key columns are stored in the levels above the leaf pages. The database buffer pool size required to try to keep nonleaf pages in memory is reduced.

While nonkey index columns may make updates faster, SELECT calls that refer to the nonkey columns in the WHERE clause or in the ORDER BY clause may suffer. An example of this is shown in SQL 13.1 and Figure 13.2. As the position of columns C and D in the index rows are now no longer guaranteed to be in sequence, only two matching columns together with the necessity of a sort are less satisfactory.

SQL 13.1

```
SELECT      D
FROM        TABLE
WHERE       A = :A
              AND
            B = :B
              AND
            C = :C
ORDER BY    D
```

The biggest benefit of nonkey index columns comes from the ability to add index columns to the primary key index.

If the DBMS does not support nonkey index columns (or the enforcement of uniqueness for a partial index key by means of a constraint, as described below),

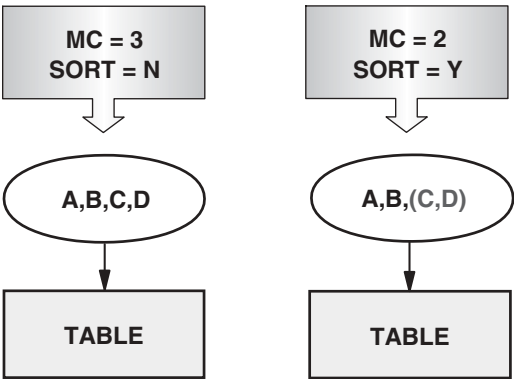


Figure 13.2 Nonkey index columns may make SELECT calls slower.

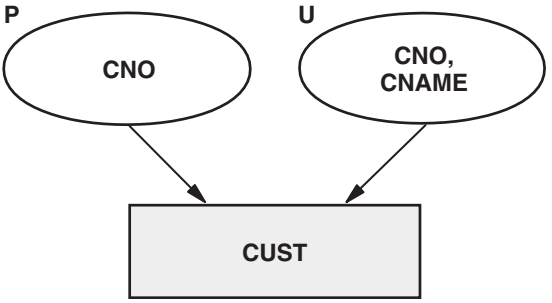


Figure 13.3 When the DBMS does not support nonkey index columns.

columns should *never* be added to the primary key index because the DBMS only ensures that the *whole index key* is unique. If column CNAME is added to the primary key index CNO as a key column, it becomes possible to have two rows in the table CUST having the same value in column CNO. To produce a fat index for the SELECT shown in SQL 13.2 with a DBMS that does not support nonkey index columns, an additional index (CNO, CNAME) may have to be created, as shown in Figure 13.3.

SQL 13.2

```
SELECT      CNAME
FROM        CUST
WHERE       CNO = :CNO
```

CONSTRAINTS TO ENFORCE UNIQUENESS

Oracle provides a different solution for the problem illustrated in Figure 13.3. Index CNO is not needed because an index (CNO, CNAME) can be used to enforce the uniqueness of the primary key CNO. This is done with a PRIMARY KEY constraint. When a candidate key must be unique, UNIQUE constraints can be used with an index that begins with the candidate key.

Note that index (CNO, CNAME) must be created without the keyword UNIQUE in Oracle, although it *is* actually unique because it contains the primary key.

DBMS ABLE TO READ AN INDEX IN BOTH DIRECTIONS

If a DBMS can read an index backwards (which is becoming common; Oracle, SQL Server, and DB2 for LUW have had this capability for some time; DB2 for

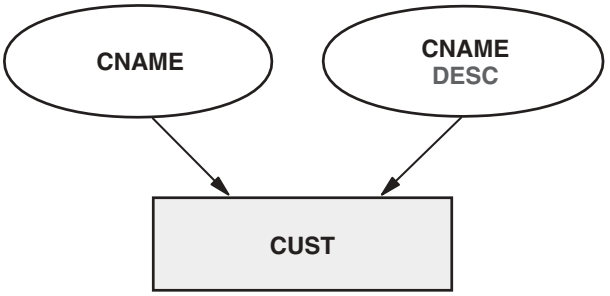


Figure 13.4 When the DBMS cannot read an index backward.

z/OS introduced this facility in V8), *browsing in both directions* can be implemented with a single index without a sort being required. Otherwise, two indexes must be created to avoid a sort of the result rows, as in Figure 13.4.

A commonly used trick to avoid mirror indexes like this is to save the index key values of the displayed screens in an auxiliary table. There is a limitation in this, however, in as far as the user is not able to go backward beyond the start point.

Note: Enabling the read backward facility may require an option to be requested; in DB2 for LUW, for instance, ALLOW REVERSE SCANS must be specified in the CREATE INDEX.

INDEX KEY TRUNCATION

Index key truncation (e.g., implemented in DB2 for z/OS) refers to the DBMS storing only that part of an index key in the nonleaf pages that is necessary to determine the page at the next lower level. As with the nonkey index columns discussed above, it reduces the number of nonleaf pages.

FUNCTION-BASED INDEXES

Let us assume that column CNAME is stored in table CUST as *mixed case*, but value CNAME is moved in *upper-case* form into a SELECT host variable (Fig. 13.5). A common solution would be to store CNAME twice in the CUST table, once in mixed case, and a second time in upper case. The latter column would be copied to the index. The two table columns would then have to be synchronized, either with triggers or by a generated column.

With Oracle, a function-based index may be created:

```
CREATE INDEX .... ON CUST (UPPER (CNAME), CNO, CITY)
```

When a predicate contains the same function, as in SQL 13.3, the optimizer will consider the function-based index. This eliminates the need for the redundant column in table CUST.

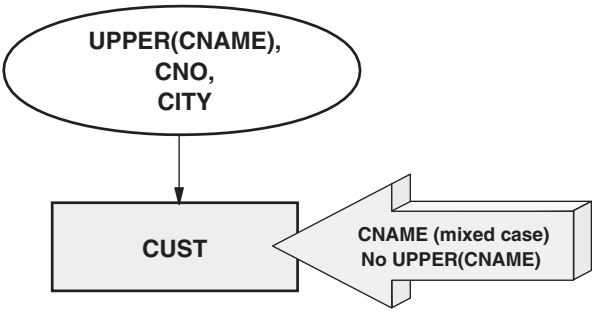


Figure 13.5
Function-based index avoids the need for a redundant table column.

SQL 13.3

```
SELECT     UPPER (CNAME), CITY, CNO
FROM       CUST
WHERE      UPPER (CNAME) BETWEEN :hv1 AND :hv2
ORDER BY UPPER (CNAME)
```

Note: If the result must show CNAME in mixed case, column CNAME must be added to the index to provide index-only access:

```
CREATE INDEX .... ON CUST (UPPER (CNAME), CNAME, CNO,
   CITY)
```

Oracle 9i supports expressions and functions (even user-written ones) in the CREATE INDEX, while SQL Server 2000 supports indexes on computed columns in a table.

INDEX SKIP SCAN

Let us assume an index (CCTRY, BDATE, CNAME) on table CUST, but no index starting with BDATE. Most optimizers would probably choose a full index scan for SQL 13.4, but Oracle would be able to consider an *index skip scan* whereby multiple slices, one slice per country as defined by CCTRY and BDATE, would be read in parallel; skip scan skips the first index column. An index starting with columns BDATE and CCTRY would then be superfluous if column CCTRY had a low cardinality; consider one million customers in 20 different countries, for instance.

SQL 13.4

```
SELECT CCTRY, CNAME
FROM   CUST
WHERE  BDATE = :BDATE              FF =  0.01%
```

Full index scan (CCTRY, BDATE, CNAME):

$$1 \times 10 \text{ ms} + 1{,}000{,}000 \times 0.01 \text{ ms} = 10 \text{ s}$$

Ideal index (BDATE, CCTRY, CNAME):

$$1 \times 10 \text{ ms} + 100 \times 0.01 \text{ ms} = 11 \text{ ms}$$

Index skip scan (CCTRY, BDATE, CNAME):

$$20 \times 10 \text{ ms} + 100 \times 0.01 \text{ ms} = 201 \text{ ms}$$

BLOCK INDEXES

DB2 for LUW V8 provides an option of multidimensional clustering: Related table rows are stored in blocks. Although this is a table design option, it is interesting from the index point of view since it can sometimes be used as an alternative to a fat index.

Let us assume an ORDER table for a company that sells 100 products in 50 countries. To make this table multidimensional, the following statement is added to CREATE TABLE:

```
ORGANIZE BY DIMENSIONS PRODUCT AND COUNTRY
```

The table then consists of *blocks*, groups of consecutive pages (2–256 pages); each block only contains rows that have the same values for all the dimensions: Some blocks contain rows that relate to country 1 and product 1. A block index on PRODUCT has pointers to all the blocks that contain rows for a particular product; likewise a block index on COUNTRY has pointers to all the blocks that contain rows that relate to a particular country. A compound block index is used for INSERTs and normal indexes are used for other single-row operations.

When a SELECT contains WHERE PRODUCT = :PRODUCT AND COUNTRY = :COUNTRY, the optimizer will probably choose an ANDing operation on the two block indexes. Then the blocks containing the qualifying rows are read. Almost all the touches are sequential.

The traditional alternative would be to have a clustered index, or a fat index, beginning with PRODUCT and COUNTRY. After reorganization, practically all touches are sequential, but the number of random touches increases after inserts. The advantage of multidimensional clustering is that inserts do not degrade clustering; there is no need to reorganize the table after a lot of inserts. The obvious price is poor disk space utilization when the dimensions have a high cardinality and some values are rare; there may then be many blocks that contain only one row.

DATA-PARTITIONED SECONDARY INDEXES

When a very large table is partitioned it may be desirable, for availability reasons, to make all indexes on the table data partitioned. The ability to reorganize partitions and their indexes in parallel is among the main benefits. DB2 for z/OS V8 provides this option.

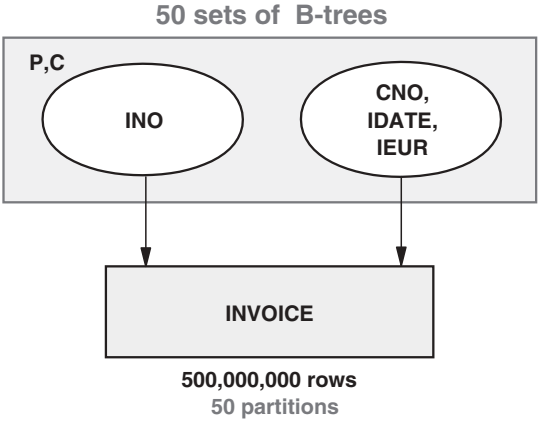


Figure 13.6 Data-partitioned secondary indexes.

Let us assume that table INVOICE has 50 partitions as shown in Figure 13.6.

```
CREATE TABLE INVOICE
             PARTITION BY (INO)
             (PARTITION1 ENDING AT (10000000),
             PARTITION2 ENDING AT (20000000)...
```

If index (CNO, IDATE, IEUR) is a data-partitioned index, 50 B-trees (index partitions) are created, one for each table partition. Programmers need to be aware of this because the minimum number of random index touches will be 50 if there is no predicate for INO. The following SELECT would search all 50 B-trees.

SQL 13.5

```
SELECT     INO, IEUR
FROM       INVOICE
WHERE      IDATE > :IDATE
ORDER BY IDATE
```

To reduce the number of B-trees searched, a predicate that specifies the relevant INO range should be added:

SQL 13.6

```
SELECT     INO, IEUR
FROM       INVOICE
WHERE      IDATE > :IDATE AND INO > :INO
ORDER BY IDATE
```

EXERCISES

8.1. Identify the index restrictions and advanced options of your current DBMS version.

Chapter 14

Optimizers Are Not Perfect

- Optimizers do not always see the best alternative
- Matching and screening problems
- Non-BT predicates
- Unnecessary sorts
- Unnecessary table touches
- Optimizers' cost estimates are sometimes wrong, perhaps disastrously so (and how we can help)
- Range predicates
- Skewed distribution
- Correlated columns
- Properties of partial index keys
- Optimizers' cost estimation formulas considerations
- I/O estimation
- CPU estimation
- Impact of pools and cache
- Helping the optimizer with estimation-related problems
- Optimize every time with actual values
- Access path hints
- Redundant predicates
- Falsifying statistics
- Modifying indexes
- Effect of optimizer problems on index design

INTRODUCTION

Sometimes a SELECT call can be unbelievably slow even though suitable indexes have been created for it. If the dominant component is service time and not queuing time, the optimizer has probably chosen a poor access path. The optimizer of a relational DBMS has a difficult task; it must choose the right access path based solely on the statistics collected for the characteristics of the tables and

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

indexes without making any measurements; the current optimizers do not issue any SELECT calls to the database when choosing the access path. In a limited way, the current optimizers validate their assumptions at execution time; they may, for instance, decide to turn on sequential prefetch for an index or a table after observing the page access pattern. In the future, the best optimizers may also ascertain the *real* filter factors at execution time.

With a simple SELECT, the most important decisions to be made are the choice of index (or indexes in the case of multiple index access) and the way it is used (matching columns, screening columns, sequential prefetch). For joins, the join method and table access order are also critical.

All serious relational DBMSs now have a *cost-based* optimizer; they identify a number of reasonable alternatives and make cost estimates for them. The cost estimate is largely a prediction of the local response time—a weighted average of the CPU time and the synchronous I/O time. The estimate is based on statistics collected by a utility, for example, the size of the tables and their indexes together with the distribution of the values of selected columns. Many of these value distributions are optional. The maximum set may include the following data per column or column combination:

- Number of distinct values (*cardinality*)
- Second largest and the second smallest value
- N most common values and their frequencies (the number of rows with a given value)
- Histogram with N bars (2% below 10, 5% below 20, etc., or the percentile values 2%, 4%, 6%, etc.)

The cost formulas of the best optimizers contain tens of variables (compared with the two variables used in the QUBE—TR and TS). These variables may include hardware information as well, such as processor speed, disk drive speed, and the size of the database buffer pool. The oldest cost-based optimizers have already had their 21st birthday, and the developers have improved them continually. Yet they sometimes choose a totally wrong access path, even for an innocent looking SQL statement. We will now discuss why this is so.

Essentially, there are *two basic problems* that we have to learn to live with; we will deal with each of them in turn:

1. Optimizers do not always see the best alternative.
2. Optimizers' cost estimates may be very wrong.

OPTIMIZERS DO NOT ALWAYS SEE THE BEST ALTERNATIVE

Matching and Screening Problems

Difficult predicates, defined in Chapter 6 as those that *cannot participate in defining the index slice*, used to be the most common reason for optimizer problems. This type of problem has become less frequent over the years for two reasons:

- Over a long period of time, the optimizers have learned to handle more predicates in an efficient way, often by transforming them into a more optimizer-friendly form before access path selection; separate tools are also available that suggest a rewrite of a given query for a given optimizer.
- In enlightened organizations, every SQL programmer will have a pitfall list, containing the most common difficult predicates for the current optimizer version, and a suggestion for circumvention.

Each DBMS seems to have a different name for difficult predicates. In SQL Server books they are called *nonsearch arguments*; some Oracle books talk about *index suppression*, DB2 for z/OS books describe them as *nonindexable predicates*.

In addition to matching problems, an optimizer may also have screening problems. With *really* difficult predicates, the DBMS is *not even able to do index screening*. This means that the DBMS has to read a table row to evaluate a really difficult predicate even though the predicate column or columns have been copied to the index. In DB2 for z/OS, really difficult predicates are called *stage 2 predicates*.

Probably one of the best known examples of these is the predicate :hv BETWEEN COL1 AND COL2; often convenient, for instance, when the host variable is CURRENT DATE and columns COL1 and COL2 are the start and end dates for a period. In some environments, the programmer has to rewrite this really difficult predicate as COL1 <= :hv AND COL2 >= :hv. Some optimizers may already do the transformation automatically.

Non-BT

The Boolean operator OR in a WHERE clause often causes unpleasant surprises because it may make a compound predicate too difficult for the optimizer. As described in Chapter 6, non-BT predicates cannot be used to reject a row when the predicate is evaluated false; they therefore impose a very serious restriction in that they *can participate in defining an index slice only with multiple index access*. A predicate is BT if a row *can* be rejected when the predicate is false. Therefore, if a WHERE clause contains only ANDs (no ORs), all predicates will be BT.

In many browsing transactions, as well as in batch job repositioning, it would be convenient to use a cursor such as the one used in SQL 14.1 to express the requirement to *display the result rows that have not yet been displayed*. At least one program generator produces cursors like this for browsing transactions.

The user enters the three first characters of LNAME, for instance, JON. The application program moves the following values to the host variables:

```
LNAMEPREV
```

In the first transaction JONAAA..., in the subsequent transactions the last LNAME value shown:

```
CNOPREV
```

SQL 14.1

```
DECLARE CURSOR141 CURSOR FOR
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       (LNAME = :LNAMEPREV
            AND
            CNO > :CNOPREV)
            OR
            (LNAME > :LNAMEPREV
            AND
            LNAME <= :LNAMEMAX)
ORDER BY    LNAME, CNO
WE WANT 20 ROWS PLEASE
```

In the first transaction 0, in the following transactions the last CNO value shown:

```
LNAMEMAX
JONZZZ...
```

When index (LNAME, CNO, FNAME) is available, the best access path is naturally MC = 2, index only, no sort. The number of touches per transaction then is minimal, TR = 1, TS = 19 for a browsing transaction displaying 20 rows per screen.

Unfortunately, the WHERE clause has *no* BT predicates. If the optimizer cannot remove the OR by rewriting, it must choose between a *full index scan* (*MC = 0*) and *multiple index access*. In the latter case, if the DBMS collects only pointers from the index slices, the access path will not be index only. Furthermore, the whole result table will be materialized because the sort of the pointers implies a sort for the ORDER BY. If the result table has 1000 rows, this means 1000 unnecessary *random* touches to the table. Exercise 14.1, at the end of this chapter, considers rewriting SQL 14.1 to eliminate the OR.

The next WHERE clause is not as bad as the one shown in SQL 14.1. Predicates P1 and P2 *are* BT, so the access path at least has MC = 2, assuming predicates P1 and P2 are not difficult.

```
WHERE P1 AND P2 AND (P3 OR P4).
```

How can we help the optimizer if it does not see the best alternative? Unfortunately, the SQL statement would have to be reformulated. It may even be necessary to split a complex cursor into several cursors. Optimizers are learning to handle transformations such as the following:

```
COLX = :hv1 OR COLX = :hv2      into
COLX IN (:hv1, :hv2)
```

Splitting a cursor, however, is a much more difficult task because application code is required between the SQL calls.

SQL 14.1 is a typical case where the ultimate solution is to split the cursor. Replacing the OR with a UNION ALL would probably lead to an unnecessary sort for an ORDER BY but omitting the ORDER BY would be quite risky; many things may change in the future. To avoid the sort and all the redundant touches, two cursors would have to be written as shown in SQL 14.2.

SQL 14.2

```
DECLARE CURSOR142A CURSOR FOR
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       LNAME = :LNAMEPREV
            AND
            CNO > :CNOPREV
ORDER BY    LNAME, CNO
WE WANT 20 ROWS PLEASE

 EXPLAIN: MC = 2, index only, no sort

DECLARE CURSOR142B CURSOR FOR
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       LNAME > :LNAMEPREV
            AND
            LNAME <= :LNAMEMAX
ORDER BY    LNAME, CNO
WE WANT 20 ROWS PLEASE

 EXPLAIN: MC = 1, index only, no sort
```

The application program first opens CURSOR142A and issues a maximum of 20 FETCH calls. If this cursor runs out of result rows, it is closed and cursor CURSOR142B is opened. The program keeps FETCHing until it runs out of result rows or has filled a screen. Before terminating, the program saves the last LNAME and CNO values for the next transaction.

This program does no unnecessary touches; it always starts at the index position at which the previous transaction exited. According to the QUBE, the local response time will be:

$$1 \times 10 \text{ ms} + 19 \times 0.01 \text{ ms} = 10 \text{ ms}$$

Let us return one last time to the search for large men.

In this case, replacing the OR with a UNION would be a good solution if the program FETCHes the whole result in one transaction, which seems to be the case as there is no WE WANT n ROWS PLEASE. Now it is easy to derive the ideal indexes for the two SELECTs:

SQL 14.3

```
SELECT    LNAME, FNAME, CNO
FROM      CUST
WHERE     SEX = 'M'
          AND
          (WEIGHT > 90 OR HEIGHT > 190)
ORDER BY  LNAME, FNAME
```

SQL 14.4

```
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
            AND
            WEIGHT > 90

UNION

SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
            AND
            HEIGHT > 190
ORDER BY    LNAME, FNAME
```

Unnecessary Sort

For many optimizers, some cursors with ORDER BY are too difficult in the sense that the optimizer sorts the result rows even though they are accessed in the right order using the selected indexes. The following two situations are in the pitfall list for DB2 for z/OS V7:

1. A cursor with UNION and ORDER BY always results in a sort, no matter how good the indexes.
2. A join SELECT with ORDER BY results in a sort if the ORDER BY refers to a column in a table that is *not* the outermost table (the table the optimizer chooses as the *first* table to access).

Some respected database gurus recommend leaving out the ORDER BY in these cases, if the result rows are in the right order even without ORDER BY. They are then assuming that the DBA will always notice any change of access path. The safest way to eliminate the unnecessary sort is to split the cursor, which may be time consuming or, if using a purchased package, impossible. A

tolerable compromise may be once again fat or even ideal indexes, to make the materialization of the whole result table fast enough.

Unnecessary Table Touches

A recently published book (1) by Gulutzan and Pelzer has an interesting approach: It describes many commonly used SQL tuning tricks and then tries them out with eight different DBMSs. The trick is considered successful if the improvement in local response time is more than 5%.

A fat index (*covering index*) is, of course, one of the tricks. Surprisingly the result is only 6/8; two of the eight products *did not* choose an index only access path. The authors then warn about the following restriction related to index only:

> *Ordinarily, using a covering index will save one disk read (GAIN 6/8). The gain is automatic but only if the columns in the select list exactly match the columns in the covering index. You lose the gain if you select functions or literals, or you put the column list in a different order.*
>
> ...

```
SELECT name FROM Table1
ORDER BY name
```

> *Will the DBMS do the "smart" thing and scan the covering index, instead of scanning the table and then sorting. Perhaps. But Cloudscape won't use covering indexes unless it needs them for the WHERE-clause anyway, and Oracle won't use an index if the result might involve Nulls. So there is a gain if you assume that NULL names don't matter and you change the search to*

```
SELECT     name
FROM       Table1
WHERE      name > ''
ORDER BY name

GAIN 7/8
```

Literals in SELECT lists are a well-known pitfall, as well as the NULL issue with Oracle, but the claim about the order of columns in the SELECT list is astonishing. The following sentence in the summary of the quoted paragraph is even more surprising:

> *DBMSs never use covering indexes when there are joins or groupings.*

The claim about joins excluding index only does not agree with our experience with DB2, Oracle, and SQL Server, nor with the recommendations in Chapter 8. It is possible that some products do have such a strange limitation, but it could also be a misinterpretation of the EXPLAIN. If the EXPLAIN does not show the expected access path, it is easy to jump to the conclusion that the optimizer *did not see* the best alternative. It is possible, and is not uncommon, that the optimizer did in fact see the "best" alternative, but the estimate differed so much from the actual value, that the optimizer chose the wrong access path simply

because it was the fastest one according to its estimates. The second part of this chapter discusses this type of problem.

Nevertheless, we recommend the book (1) from which the above quote was taken, especially if the DBMS manuals or performance guides do not adequately cover the limitations of their current optimizer.

OPTIMIZERS' COST ESTIMATES MAY BE VERY WRONG

Range Predicates with Host Variables

```
WHERE COL1 > :COL1
```

If the optimizer does not estimate the filter factor at execution, when the value of the host variable :COL1 is known, it must resort to a *default value.* The default value may be constant, 1/3, for instance, or it may depend in some way on the cardinality of the column; for example, DB2 for z/OS V7 assumes that the filter factor for a range predicate becomes smaller the higher the cardinality of the column. In any case, it can be *very* wrong. This may result in it choosing, among other index problems, the wrong index or the wrong table access order.

Making estimates every time an SQL call is executed consumes a large amount of CPU time, but the default FF values often lead to very poor cost estimates. Many products now offer a compromise option: The optimizer may be asked to make the cost estimates the first time an SQL call is executed and to store the chosen execution plan for reuse. This technique may well result in better cost estimates than with the default filter factors, but it is somewhat unpredictable; the values moved into the host variables at the first execution may be atypical.

Figure 14.1 shows a scenario that may lead to very poor filter factor estimates. Both predicates refer to host variables. For the range predicate A < :A, the optimizer has to use a default filter factor that might depend on the cardinality of the column; for example, if column A has a *high* cardinality, the optimizer could assume a *low* filter factor, such as 0.01%.

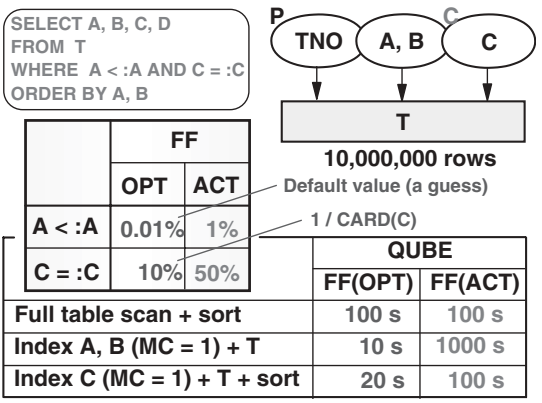


Figure 14.1 Scenario that may lead to very poor filter factor estimate.

The equal predicate C = :C is much easier for the optimizer. If the cardinality of column C is 10, the optimizer assumes the filter factor is 10%, which is the correct value for the average case. However, the worst input filter factor for C = :C might be 50%.

In Figure 14.1 the first FF index estimate is *very low* and the second one *low*. Based on these poor estimates, the optimizer would make the wrong choice.

We may help the optimizer by forcing it to choose the access path at each execution when the actual host variable values are known. This would probably solve the problem, at least if the optimizer had a histogram available that showed the distribution of the values for column A. Knowing the minimum and maximum values may not be sufficient if the distribution is skewed as shown in Figure 14.2.

An access path hint would avoid the overhead involved in estimating the cost at each execution. This would be sensible if index C was indeed the best choice with all input.

The best solution for this example though, as with many real-life situations, is to make the best index *even better* as shown in Figure 14.3. This index enhancement not only improves the LRT; it also eliminates the need to use the tricks described above. A really good index is hard to resist with a cost-based optimizer.

Skewed Distribution

In the example shown in SQL 14.5, we are assuming a single-column index, SEX, and that 99.9% of Sumo wrestlers are men.

SQL 14.5

```
SELECT      AVG (WEIGHT)
FROM        SUMO
WHERE       SEX = :SEX
```

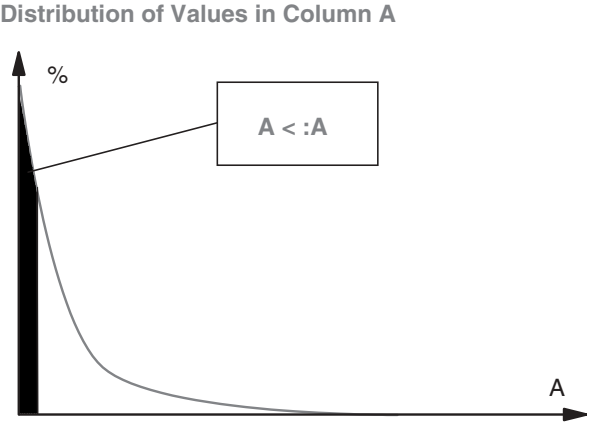


Figure 14.2 Histogram showing the distribution of the values for column A.

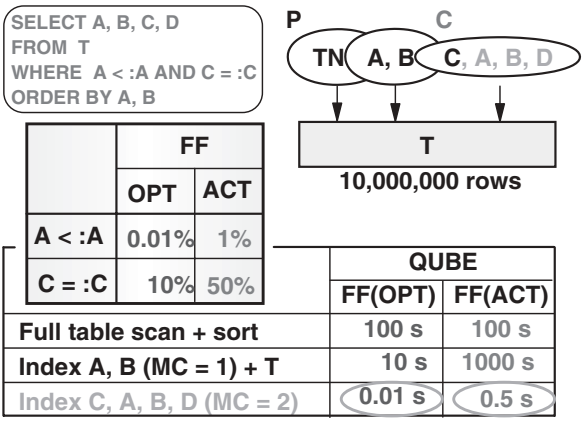


Figure 14.3 Making the best index even better.

Even if the distribution of the values in column SEX is stored in the system tables, the optimizer cannot make use of it if the access path selection is not done repeatedly at every execution time. When the access path is selected with a host variable in the predicate, the optimizer has to assume that the filter factor will be 1/cardinality = 0.5. It will then select a full table scan.

If the only purpose of the index is to find the exceptions, the solution is simple—a literal should be used instead of a host variable. For predicate SEX = 'F', the optimizer would assume the correct filter factor 0.1%. Then it would probably decide to use the index.

If the user sometimes asks for the average weight of female Sumo wrestlers and sometimes that of males, there are solutions that enable the best access path to be chosen for both cases, without repeated cost estimates:

- Two SELECT statements are written, one with SEX = 'F', the other with SEX = 'M' and the application program allowed to choose the appropriate SELECT depending on the user input.
- If static SQL with bind is not available, use an option that stores the statements and their access paths in a pool for reuse. Be careful that the optimizer does not convert the literals into variables, otherwise it will use the same access path for both cursors. This option is called *cursor sharing* in Oracle. If the INIT.ORA parameter CURSOR_SHARING = FORCE, all statements that differ only in the values of the literals, use the same execution plan. CURSOR_SHARING = EXACT disables cursor sharing; there will be a separate execution plan for each literal. With CURSOR_SHARING = SIMILAR, the optimizer will examine the statistics and *may* use different execution plans.

Thus, another old recommendation has become a myth:

Do not create an index for a column with a low cardinality.

By the way—you might like to consider the following question:

Should a single-column index be dropped if the cardinality of the index column is one?

For example, assume the system tables are being scanned to investigate the current indexes. A single-column index is found with a cardinality of one. Should this index be dropped?

It is certainly worth while spending some time considering this situation. Our comments will be found on the ftp site for this book. See Preface for details.

Finding an access path in a pool is much faster than repeating the access path selection process with cost estimates. Current hardware makes an SQL pool of several gigabytes feasible. If each SQL statement together with its access path takes a few kilobytes, the pool may have space for a million access paths. This is of critical importance when using a system like SAP or Peoplesoft, not because of skewed distributions but because without reuse of saved access paths, *every* SELECT would incur the overhead of estimating the cost of alternative access paths.

Correlated Columns

```
WHERE MAKE = :MAKE AND MODEL = :MODEL
```

The optimizer is able to produce a good estimate for this compound predicate only if it knows the *cardinality of the column combination* MAKE, MODEL. Some products (e.g., DB2 for z/OS) provide an option to determine the cardinality of N first index columns such as CARD(MAKE, MODEL) in index (MAKE, MODEL, YEAR) in the utility which updates the optimizer's statistics. Other products (e.g., SQL Server 2000) collect this information (SQL Server term: density) automatically. Therefore, if columns MAKE and MODEL are the *first two* columns in a multicolumn index, the optimizer may be able to ascertain the correct filter factor for the compound predicate with highly correlated column value distributions; remember that only Ford made Model Ts!

If the optimizer *does not know* CARD(MAKE, MODEL), the cardinality of the concatenation of the first two index columns, it will assume a very low cardinality of

$$1 \;/\; \text{CARD (MAKE)} \times 1 \;/\; \text{CARD (MODEL)}$$

If the optimizer does not know CARD(MODEL), it may use a general default cardinality (e.g., 25) or, if it knows CARD(fullkey), the cardinality of the *whole* index key, it may interpolate between CARD(MAKE) and CARD(fullkey); for example, if CARD(MAKE) = 50 and CARD(MAKE, MODEL,YEAR) = 2000, an optimizer might use 1025 (the average of 50 and 2000) as an estimate for CARD(MAKE,MODEL). Both of these approaches can obviously lead to a filter factor estimate that is very wrong.

Mark Gurry (2 p. 36) says that this is a common problem, according to his Oracle experience:

In summary, why does the cost based optimizer make such poor decisions? First of all, I must point out that poor decision-making is the exception rather the rule. The examples in this section indicate that columns are looked at individually rather than as a group. If they were looked at as a group, the cost based optimizer would have realized in the first example that each row looked at was unique without the DBA having to rebuild the index as unique. The second example illustrates that if several of the columns have a low number of distinct values, and the SQL is requesting most of those values, the cost based optimizer will often bypass the index. This happens despite the fact that collectively, the columns are very specific and will return very few rows.

Cautionary Tale of Partial Index Keys

The following cautionary tale, albeit somewhat simplified and containing a few hypothetical numbers, is based on an experience encountered by a Scandinavian company.

This company had bought a monitoring tool to identify exceptionally slow SQL. In the very first report produced, the DBAs found that one frequently executed SELECT was quite slow and consumed a large amount of CPU time. The SELECT was a simple one, and the problem was obvious—there was no adequate index. The SELECT, SQLA shown in SQL 14.6, was using the primary key index, which was not even semifat for the call. Furthermore, there was only one matching column (C1), and the index slice defined by that column was quite thick with the worst input. A new index with three matching columns, shown in Figure 14.4, was created and, not surprisingly, the performance of the slow and expensive SELECT improved dramatically. This was the good news.

Now for the bad news. Soon after the new index was created, a sales manager, due to leave for a meeting in Germany early the following morning, was trying to create a report with an SQL reporting tool. Normally, this report was produced quite quickly, but now it took a *very* long time. The sales manager, fortunately, did not miss his flight, but he complained mightily to the IT department. The

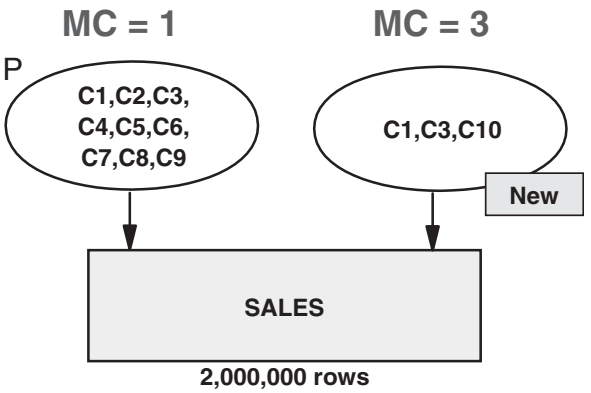


Figure 14.4 New index makes SQLA much faster.

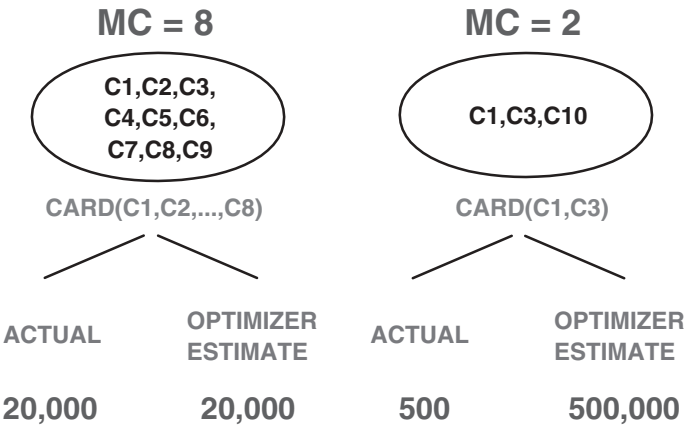


Figure 14.5 Optimizer does not know everything.

DBAs were shocked when they discovered that this tool generated SELECT, SQLB, shown in SQL 14.7, now used the new index instead of the primary key index; the latter appeared to be much better suited, using eight matching columns instead of only two with the new index, as shown in Figure 14.5. The DBAs were forced to drop the new index while they investigated the behavior of the optimizer.

SQL 14.6

SQLA

```
SELECT   many columns
FROM     SALES
WHERE    C1  = :C1
         AND
         C3  = :C3
         AND
         C10 = :C10
```

SQL 14.7

SQLB

```
SELECT    many columns and sums
FROM      SALES
WHERE     C1 = :C1 AND C2 = :C2 AND C3 = :C3 AND C4 = :C4 AND
          C5 = :C5 AND C6 = :C6 AND C7 = :C7 AND C8 = :C8
ORDER BY...
GROUP BY...
```

A cost-based optimizer does not make its decision based on the number of matching columns. It makes cost estimates for several feasible alternatives. In this case, when it considers the new index for SQLB, it needs to know the filter factor for the compound predicate C1 = :C1 AND C3 = :C3.

If the actual cardinality for column group (C1,C3) is 500, the *average* filter factor of the matching predicates will be $1/500 = 0.2\%$. The number of index and table touches is then $0.002 \times 2{,}000{,}000 = 4000$ for each. The *worst* input may result in many more touches. Likewise, if the actual cardinality for column group (C1,C2,C3,C4,C5,C6,C7,C8), the first eight columns of the primary key, is 20,000, the average filter factor for the whole WHERE clause is 1/20,000, and the number of touches (because all eight simple predicates are matching) is only $100 + 100$. If the optimizer knew all this, it would not use the new index.

What does the optimizer know? It depends on the product and the options chosen; but, often, by default, the optimizer only knows the cardinality of the *first* index key (new index: C1), together with the cardinality of the whole index key (new index: C1,C3,C10). When the optimizer needs to know the cardinality of (C1,C3), it must use interpolation between these two values. The interpolation formulas tend to be confidential, but it is not necessary to know the details in order to appreciate that the optimizer's estimates can be very wrong in a case like this. The optimizer needs help.

Many DBMS products provide an option in the utility that builds the statistics for the optimizer (RUNSTATS, ANALYZE, or equivalent) to collect information about partial index keys, such as cardinality, histograms, or the most common and most rare values. It may be wise to choose the cardinality option for all partial keys of all indexes, at the very least. Requesting statistics options only for some indexes, columns, or column groups sometimes makes the situation worse; it may be preferable for the optimizer to be wrong in a consistent way, rather than knowing a lot about some objects while using the default values for others. We encountered one case, for example, where an optimizer chose a wrong index because, even though it knew that the cardinality of CNO, a foreign key, was 5,000,000, it did not know the cardinality of CNAME, and therefore used the default cardinality, 25. The optimizer would have made the right choice if it had used the default cardinality for both CNO and CNAME.

But then, in real life, it is not easy to follow the rather obvious recommendation of equal treatment for optional statistics. Detailed statistics are often created for a column because the default filter factors result in a wrong access path. Is it possible, then, to do the same for all related columns? Maybe not. It would be preferable for the DBMS to make intelligent recommendations about optional statistics. After all, the optimizer knows when it must resort to a default filter factor. The optimizer could also monitor the real filter factor and compare it with its assumption. The most advanced products are already taking the first steps in this direction (e.g., DB2 for LUW 8.2 and Oracle 10 g). Any improvement in this area would be very welcome because the need for hints would be reduced. It is better to address the cause than the symptoms.

Returning to our example, when the cause of the optimizer's strange behavior was identified, it was quite easy to see how the optimizer could be persuaded to use the new index for SQLA, but not for SQLB. In this specific case, an easy solution would be to make the new index (C10,C1,C3) instead of (C1,C3,C10). Now with SQLB, the optimizer perhaps does not even bother to make an estimate for the new index because there are no matching columns. If it were to make an estimate, the new index would probably appear to be much more expensive than the good old index.

COST ESTIMATE FORMULAS

The optimizer's FF estimates, discussed above, are the weakest link in the cost estimation process. If an FF estimate is very wrong, and it could be wrong by an enormous factor, the cost estimate will be very wrong. If these erroneous cost estimates do not happen to occur in a consistent manner, the optimizer will probably make a poor choice. However, there will be many cases when the optimizer makes perfect FF estimates and knows exactly how many rows need to be touched in each step of the execution plan. Then, the quality of the cost estimate will depend on the accuracy of the cost estimate formula.

What exactly *is* the *cost* that the optimizer estimates? In simple cases (synchronous reads only, no parallelism) it may be the elapsed time (excluding contention), *the service time*; but there are many reasons why the cost estimates may be very different from the actual service time, even with perfect FF estimates. Maybe this is why the optimizer developers insist on the term "cost." Anyway, would we really want the cost to be the response time? Consider the following alternatives for a SELECT:

- Response time 3 s, 100,000 pages read asynchronously, in parallel from six disk drives, CPU time 2 s.
- Response time 5 s, 1000 pages read synchronously, CPU time 0.1 s.

Are these fair cost estimates for the two alternatives? Would you want the optimizer to choose the first alternative in an operational environment with 1000 concurrent users?

The optimizers used by the major products (at least the current versions) estimate both I/O cost and CPU cost. Predicted parallelism is sometimes taken into account, reducing the cost estimate, but sometimes not. The total cost is then a weighted sum of these components. Thus, it does relate to the service time although the correlation is not always very high.

Estimating I/O Time

This would seem to be quite easy (if parallelism is ignored); even we know how to do that, and we can do it quickly. If the optimizer has good FF estimates and the statistics are up-to-date, it is able to predict the number of random reads and the

number of pages read sequentially exactly. Some optimizers do have problems in estimating the effect of prefetch, however, especially when parameters are used to limit the number of prefetched pages.

The optimizer then simply needs the coefficients for the two types of I/O, such as 10 ms per random read and 40 MB/s for sequential read. The first value is an adequate estimate for the drives currently in use, but the speed of sequential read varies widely. It may be 2 MB/s for really old disk servers and 80 MB/s for the latest ones. Now there are three alternatives:

1. The optimizer developers decide on a typical disk system for each DBMS version and choose the coefficients accordingly. DB2 for z/OS (only used on mainframes) and SQL Server 2000 (only used with Windows) currently seem to have this approach.
2. The optimizer provides external parameters that can be used to specify the characteristics of the disk system. DB2 UDB for LUW (used on many different platforms) is built in this way.
3. The utility that collects the statistics may measure the I/O speed and store the values in the System Statistics. This approach is implemented in Oracle 10 g (also used on many different platforms).

This is not a minor point. If the optimizer assumes 4 MB/s for sequential read while the actual speed is 40 MB/s, the I/O time estimate to scan a 400-MB table will be 100 s, whereas the actual time will only be 10 s. Consequently, assuming the CPU time is ignored, the optimizer would consider the break-even point for choosing between a full table scan and a nonfat nonclustering index to be FF = 1% (10,000 random reads) while the actual break-even point would be FF = 0.1% (1000 random reads).

Buffer pool and cache hits are also important issues that make it difficult to estimate I/O time. Our quick estimates (the QUBE) are often pessimistic because every random touch is assumed to take 10 ms. As memory becomes ever cheaper, a requested table or leaf page is more likely to be in the database pool or in the read cache. For applications with which one is familiar, it may be possible to estimate which pages are accessed so frequently that they are likely to remain in the pool. It would then be possible to add a third parameter type of touch to the QUBE formula—*a cheap random touch*, costing 0.1 ms.

Evaluating the impact of pool or cache hits though is tedious for us and very difficult for the optimizers. The current optimizers may assume that really small indexes and tables stay in the buffer pool, but in general their estimates about pool behaviour are almost as rough as those of the QUBE. The exact algorithms are confidential and they are sometimes quietly changed, but the main assumption seems to be that only the root page remains in memory. This explains a fairly common problem, namely, that the optimizer may choose a wrong index if the right index, often a fat one, happens to have one more level than the chosen index.

Estimating CPU Time

This requires a prediction of the number of instructions or cycles required by an SQL call, and then multiplying this value with a processor-dependent coefficient. Again, if the optimizer knows the number of random touches, the number of sequential touches, the number of rows sorted, and the number of result rows, it could be expected to make a fairly decent CPU time estimate. Actually the optimizer may take many other factors into account, such as the number and complexity of predicates, but there is one additional factor that reduces the accuracy of the CPU time estimates: the high-speed caches in memory. If an instruction or a row is not in this cache, it must be moved from the main memory. This may involve a wait that is quite long compared to the processor cycle time itself. The gap between processor speed and memory speed is constantly growing. Therefore, the high-speed cache hit ratio has an exceedingly significant impact on the actual CPU time. This hit ratio depends on the complexity of an SQL call and on the level of contention. This is why CPU times may be longer at peak time than in a stand-alone benchmark. It should be noted that the CPU time, as measured by the DBMS monitors, includes this *memory wait time*, yet we normally consider the measured CPU time to be the service time, as opposed to CPU queuing time (waiting for an available processor).

How do the optimizers choose the processor-dependent coefficient to estimate the CPU time? DB2 for z/OS is aware of the CPU model used for doing the cost estimates and uses an appropriate value. Oracle 8 did not estimate the CPU cost at all; Oracle 9 has external parameters while Oracle 10 g uses the System Statistics described above. DB2 for LUW also has external parameters.

Helping the Optimizer with Estimate-Related Problems

Optimize Every Time—with Actual Values Moved to Predicate Variables

The optimizer may be made to choose the access path every time, when the values of the host variables in the predicates are known. However, with operational applications, the CPU time overhead tends to be too high—an overhead that is increasing not decreasing, as the cost formulas become more sophisticated. Therefore, this alternative should be chosen only when the optimal access path depends on the values moved into the host variables (as in SEX = :SEX in the Sumo wrestler case).

The implementation of this option is product dependent.

- In DB2 for z/OS, the package containing SELECT... WHERE SEX = :SEX must be bound with parameter REOPT(VARS). Then the optimizer will recalculate the cost estimate every time, using the actual value moved to host variable :SEX.
- To use two different access paths with Oracle, two SELECTs must be coded, one with WHERE SEX = 'M' and another with WHERE

SEX = 'F'. The appropriate SELECT must be chosen by the application program. Furthermore, the INIT.ORA parameter must be either EXACT or SIMILAR.

- In some environments, dynamic SQL may have to be used, with SQL caching disabled.

Optimize every time should not be confused with *optimize the first time*. Oracle 9i makes the cost estimate the first time an SQL call is executed and then stores the SQL and the execution plan for reuse; this is called *bind variable peeking*. DB2 for z/OS V8 provides the same facility by means of REOPT(FIRST) for dynamic SQL. With these options, if the first execution of SELECT AVG (WEIGHT) FROM SUMO WHERE SEX = :SEX refers to men, the optimizer will always use a full table scan.

When filter factors are estimated with values, either with constants or values moved to a variable, the optimizer, of course, needs adequate information about the column (or column group) value distributions: a histogram or TOP/BOTTOM N. How else would the optimizer know, for instance, that 99.9% of Sumo wrestlers are men?

Access Path Hints

Access path hints provide a more effective solution than *optimize every time* when the best access path is independent of the input—unaffected by, for example, skewed distributions. The hint informs the optimizer that the indicated access path is the cheapest one. If the optimizer is able to generate this access path, it will not even estimate the costs.

There are three kinds of hints:

1. Specific (the whole access path is specified)
2. Restrictive (some alternatives are ruled out)
3. Informative (the optimizer is given information that enables better cost estimates)

Oracle now has more than 100 different hints and they are widely used, partly for historical reasons. SQL Server 2000 has a limited number of hints (join method, choose index, etc.). DB2 for z/OS has the facility described below for specifying any access path the optimizer is able to build. DB2 for LUW V8 has no hint facility.

To provide some understanding of the scope of access path hints, we will show a few simple Oracle examples, but please understand that this is but a small sample:

```
SELECT  /*+ ORDERED */ C1,C2,C3
FROM    A,B,C
WHERE ...
```

specifies that the table access order must be A, B, C. Replacing this hint by LEADING would specify that the outermost table, the driving table, in a join must be the first table in the FROM clause. The optimizer is to choose the order of the other tables:

```
/*+FULL(table)*/                Choose full table scan
/*+INDEX(table index)*/         Choose index scan
/*+NO_INDEX(table index)*/      Do not use specified indexes
/*+INDEX_FFS*/                  Choose FFS (1)
/*+USE_NL(table)*/              Choose nested loop join (2)
/*+USE_HASH(table1 table2) */   Choose hash join
/*+USE_MERGE(table1 table2)*/   Choose sort-merge join
/*+CARDINALITY(card)*/          The size of result table (3)
```

1. Fast full index scan (FFS) reads all leaf pages in physical sequence, parallelism enabled. The result is not in index key order.
2. The specified table is the inner table in a nested loop.
3. A table name can be added to the hint to specify an intermediate result.

In SQL Server 2000, the query hints are usually expressed by using the OPTION clause, such as the following example, which means we want 20 rows please:

```
SELECT...FROM...WHERE...ORDER BY...OPTION(FAST 20)
```

or by an addition to the JOIN clause, such as a request for a hash join to take place:

```
SELECT...FROM table1 INNER HASH JOIN table2 ON...
```

The SQL Server hints include the following; the first one specifies an index or a set of indexes with ANDing, and the rest for specifying the join method and table access order:

```
INDEX =
HASH
MERGE
LOOP
FORCE ORDER
```

In DB2 for z/OS, the access path hints are implemented externally to the SQL code. The characteristics of the access path chosen by the optimizer are stored in PLAN_TABLE, which has dozens of columns. Many of these columns can be updated to describe the alternative we would like the optimizer to choose. The SQL call is then rebound with parameters that refer to the specified access path. If that access path is one of the alternatives seen by the optimizer, it will be chosen without any cost estimate being made. Thus we may, for instance, specify another index, another join method, or a different table access order. However, if

one of the predicates is too difficult for the optimizer, resulting in MC = 1 (one matching column), changing the MC value to 2 would not help!

Compared to the Oracle and SQL Server implementations, the DB2 hint is somewhat tedious to use. The developers recommend it mainly for fast fallback to a previous (better) access path. This is easy to do if the old access plans are stored in the PLAN_TABLE. Furthermore, influencing generalized programs (such as SAP and Peoplesoft) is easier with the DB2 approach because the source programs are not changed. When the SQL calls cannot be changed, the alternative approach requires the use of VIEWs with hints or storing the SQL call with its hints in system tables (*stored outlines* in Oracle).

Redundant Predicates

This is a clever, but dirty, way to influence the optimizer by making an unwanted alternative look expensive. Consider WHERE A < :A AND C = :C as shown in Figure 14.1, and assume the optimizer chooses index A because of poor FF estimates. A redundant predicate may be added, such as 0 = 1, that does not affect the result but makes index A unattractive:

```
WHERE (A < :A OR 0 = 1) AND C = :C
```

The redundant predicate makes predicate A < :A non-BT; now index A has no matching columns. Even with the poor FF estimates, the optimizer makes a lower cost estimate for the better alternative, index C.

The dangers of using this approach are obvious:

1. The optimizer may one day be intelligent enough to remove the redundant predicate.
2. There will be only one matching column when A is added to index C (Fig. 14.3).
3. What will our grandchildren think if they discover we have written code like this?

Falsifying Statistics

The statistics for the optimizer are normally held in tables that can be updated with SQL. This is useful for making test databases look like production databases to catch optimizer problems early. It is also possible to influence the optimizer by falsifying the statistics in the production system. The table access order in a join, for instance, could be influenced by changing the number of rows in a table from perhaps 10,000 to 100,000. This is, of course, even more risky than using redundant predicates because this sort of change may affect many SQL calls.

This trick has been used by several companies whose optimizer had chosen a wrong index, perhaps because a better index happened to have one more level (the optimizer assuming that only the root page would stay in the buffer pool). One installation affected by this changed the number of levels for a particular index in the statistics, so the optimizer would choose the right index. This seemed a

rather harmless lie—until one day the company had to do a complicated recovery procedure. Following this, one critical transaction ran very slowly for a couple of hours because the system tables had reverted to the correct number of levels for the index. The DBAs decided never to falsify statistics again.

Modify Index

The right index can be made more attractive, or the wrong index can be made unattractive, for a SELECT. We have seen examples of both in this chapter. This is often a good approach.

DO OPTIMIZER PROBLEMS AFFECT INDEX DESIGN?

The short answer is a qualified no. It is difficult for a cost-based optimizer to resist a really good index. Therefore, when the optimizer chooses the wrong index because of poor filter factor estimates, the best solution may be to make the good index even better. The qualification that must always be borne in mind is the one we discussed earlier in the section on partial index keys; a SELECT statement may mistakenly begin to use a newly created index, even with cost-based optimizers, although the previously chosen index was quite suitable. We have to accept that this risk exists whenever we add a new index taking whatever steps were appropriate to monitor inappropriate use. The risk is likely to be much smaller when columns are added to an existing index.

The only way to totally avoid performance degradation due to index improvements would be to use specific hints with *every* SQL call. This would be extremely tedious and defeats the whole idea of leaving the access path choice to the optimizer; it would also be rather risky as we are not perfect either. Likewise, it would negate any positive side effects of a cost-based optimizer; when an index is improved to make one SQL call faster, many other calls tend to become faster as well. The preferred approach is to design very good indexes as early as possible, and provide the optimizer with ample information about column and column group value distributions, at least *the cardinalities of all partial index keys*. The cardinalities, unlike histograms and Top N/Bottom N, are useful for equal predicates, even when the estimates are made with host variables. Consider index (A,B,C) and WHERE A = :A AND B = :B.

The risk of negative side effects when improving indexing is analogous to the dilemma that optimizer developers face—whenever they improve the cost estimate formulas, many SQL calls around the world will become faster; unfortunately a few will become slower.

EXERCISES

14.1. Rewrite the cursor shown in SQL 14.8 to a cursor whose access path is
- MC = 1
- Index only
- No sort

SQL 14.8

```
DECLARE CURSOR141 CURSOR FOR
SELECT   LNAME, FNAME, CNO
FROM     CUST
WHERE    (LNAME = :LNAMEPREV
         AND
         CNO > :CNOPREV)
         OR
         (LNAME > :LNAMEPREV
         AND
         LNAME <= :LNAMEMAX)
ORDER BY LNAME, CNO
WE WANT 20 ROWS PLEASE
```

Restriction: You may not remove the ORDER BY.
Hint: The WHERE clause may contain the operator NOT, but NOT makes the predicate difficult for the optimizer (no matching).

14.2. List the most common pitfalls for your current optimizer.

Chapter 15

Additional Estimation Considerations

- Detailed look at the assumptions behind the QUBE
- Nonleaf pages and leaf pages with regard to buffer pools, subpools, and disk read cache
- Occasions when the response time can be much less than the QUBE
- Leaf and table pages remaining in memory
- Small tables, hotspots, and comebacks
- Skip-sequential
- Calculating the CPU time
- Default CPU coefficients
- Measurements and considerations
- CPU estimates used in making index design comparisons

ASSUMPTIONS BEHIND THE QUBE FORMULA

The simple QUBE formula is based on the following main assumptions:

1. CPU queuing time is insignificant—several processors, reasonable processor load.
2. Disk drive utilization (drive busy) is between 15 and 35%.
3. Other queuing is insignificant.
4. *CPU time is the dominant component of sort time.*
5. *Index nonleaf pages are in memory or in the disk server read cache, but index leaf pages and table pages are always read from disk.*
6. Processor and disk drive speeds:
   - At least 200 mips or 1 GHz per processor.
   - Disk drives 10,000 or 15,000 rpm, average seek time less than 5 ms.
   - *Sequential read from disk 40 MB/s.*

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

7. *Row length not much more than 500 bytes.*
8. *Nonsequential touches are considered to be random.*

Although we have discussed these assumptions throughout this book, we will now consider the ones *shown in italics* in further detail.

NONLEAF INDEX PAGES IN MEMORY

So far, we have been concerned only with the lowest level of a B-tree index, the leaf pages. We have assumed that the read time for nonleaf pages can be ignored with current hardware, but is this a realistic assumption? Let us consider an example with a typical fat index.

Example

Figure 15.1 shows a fat index in an operational database. The table contains information about the policy claims made on an insurance company. How many levels will this index have and how much space will the nonleaf pages require?

Assumptions

- The total length of an index row, 10 columns plus system overhead, is 80 bytes.
- All index pages, leaf as well as nonleaf, contain 10% free space after reorganization.
- No leaf pages are left empty at reorganization.
- The index page size is 4 kb; if the pages are larger than this, the calculation below can be redone—the number of levels may be less but the *space* taken by the nonleaf pages will not be essentially different.
- The DBMS does not truncate the keys stored in nonleaf pages.

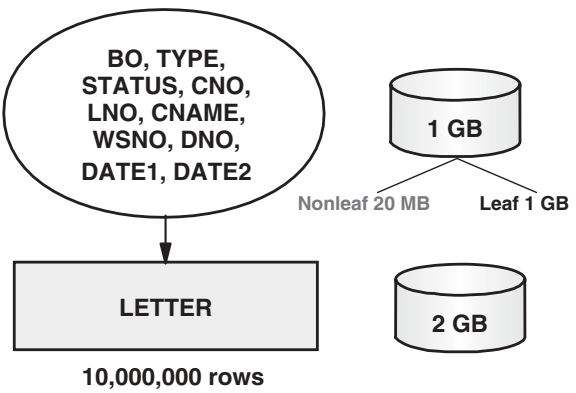


Figure 15.1 There are many nonleaf pages in a fat index.

Number of Index Pages

After an index reorganization, there are about 45 index rows in a leaf page. Thus, the number of leaf pages is $10{,}000{,}000/45 = 220{,}000$. Level two consists of $220{,}000/45 = 4900$ pages, level three $4900/45 = 110$ pages, level four $110/45 = 3$ pages. There is only one page in level five, the root page.

Thus, there are about 5000 nonleaf pages, and their total space requirement is about 20 MB. The total number of pages in the index is about 225,000. Thus, the disk space requirement is 1 GB.

Note About Key Truncation

If only that part of the key needed to choose the correct page at the next lower level is stored in the nonleaf pages, and if the keys are truncation-friendly, the number of nonleaf pages would be significantly lower. In our example, LNO is the primary key. As the five first index columns make an index key unique, the index columns following LNO would not be stored in the nonleaf pages. Because the longest column in the index, CNAME, follows LNO, the number of nonleaf pages could be as low as 2000 instead of 5000. Unfortunately, not all index keys truncate as well as this.

The disk space requirement is probably not an issue (according to our rule of thumb, the cost is only about U.S.$50 per month), but will the nonleaf pages fit in memory, as we have assumed? In this index, the number of nonleaf pages is 2% of the number of leaf pages, (5000/220,000). Let us assume this is a typical index.

The next question is the total size of all the operational indexes. In a medium-size insurance company, a few million customers, the total size of the operational tables may be 500 GB. The total size of the indexes for these tables may also be 500 GB. Using the same 2% coefficient derived above, the space required for all the nonleaf pages will then be 10 GB. *This is a little too much for current database servers*, which typically have 16 to 64 GB of memory. Furthermore, virtual storage considerations set a limit on the size of database buffers if the system does not yet support 64-bit addressing.

Impact of the Disk Server Read Cache

The rapidly growing disk server read cache provides a solution. At the time of writing, a disk server may typically have 64 GB of read cache, and this amount of semiconductor memory does not carry an inhibitive price tag. The essential factor is the transfer time from read cache to memory; with current disk servers and fiber channels, this will be less than 1 ms per 4-kb page. Although this time is much longer than the retrieval time from a pool in memory, a disk server read cache that is significantly larger than the database buffer pool helps to keep the average access time to nonleaf pages low.

If the size of the database buffer pools and read cache are only 10% of the values presented in Figure 15.2, while the operational database size remains one

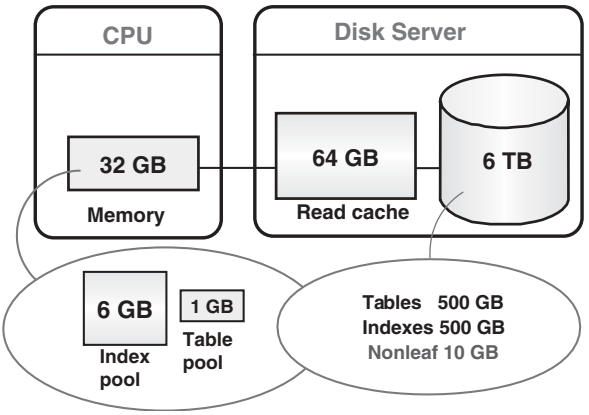


Figure 15.2 How often must a nonleaf page be read from a disk drive?

terabyte, retrieving an index row will often take two random disk reads, a level 2 nonleaf page and a leaf page; 20 ms. This may double the elapsed time of some update transactions that have to maintain many indexes.

Many operational databases are much smaller than those for this insurance company. If the whole database, tables and indexes, takes only, say, 32 GB, it *may* remain resident in the 64-GB read cache of a disk server—or at least the hit ratio may be close to 100%. In that case, the elapsed time for a random touch is about 1 ms instead of 10 ms. Unfortunately, though, this can be realized only if the other files sharing the 64-GB read cache are *friendly neighbors.*

A file is a friendly neighbor if it never occupies a large part of the read cache. If a file is always accessed sequentially, its pages do not stay long in the read cache because disk servers try to give preferential treatment to randomly read pages. If the accesses to a large file are totally random and the access frequency is not very high, each page that comes into the read cache stays there until it becomes the least recently used randomly read page. Such pages may occupy the read cache for 10 min, for instance. If a page is accessed randomly and frequently—more than once every 10 min, say—it stays in the read cache for a long time.

A disk server with a 64-GB read cache is likely to have 64 or 128 disk drives sharing the cache: several terabytes. The 32-GB operational database in our example is likely to have so many bad neighbors that a significant portion of its pages are at times overlaid by the workload sharing the read cache. Assuming a value of 1 ms for TR may be very optimistic unless the whole server is dedicated to the small database (unlikely because of cost reasons).

Buffer Subpools

Many installations divide the database pool into five subpools:

1. Application indexes

2. Application tables
3. Work files for sorting and temporary tables
4. System tables and their indexes
5. Dedicated pool for in-memory tables and indexes

There are two reasons for using subpools: (1) monitoring and (2) prioritization.

Monitoring

If one transaction causes 500 random disk reads, it would be useful to know how many of them apply to application tables and how many to application indexes. The overhead of monitoring by subpool is much lower than the overhead of monitoring by object.

Prioritization

It is now technically and economically feasible to hold selected tables and indexes in memory with a dedicated database buffer subpool; each table or index page is read into it only once from the disk server after the DBMS has been started. A realistic order of magnitude for such a subpool is a few gigabytes. In the near future, a third dimension may be required regarding billing for outsourced hardware, in addition to CPU time and allocated disk space—*data held in memory.* Currently, the cost might be up to U.S.$1000 per month per gigabyte, but this figure is still decreasing rapidly as time goes by.

Current memory (16 to 64 GB) is largely occupied by system and application programs. What is left for the operational database buffer pools is often only a few gigabytes. As time passes, with 64-bit addressing and even cheaper memory, database buffer pools of a 100 GB will become feasible.

When the DBMS is started, the subpools are empty. It may take half an hour before the largest subpool is filled with pages read from the disk server. From then on, new pages from disk must overlay older pages that, if updated, will have already been sent to the disk server by the DBMS. The main algorithm used for choosing which pages are to be overlaid is the *least recently used algorithm (LRU)*. Sequentially read pages stay in the pool for a shorter period of time than randomly read pages. Many DBMSs allow a systems administrator to influence the overlay algorithms at a subpool level.

Determining the size of the database buffer pools is one of the key tuning activities at the DBMS level. The basic objective is to increase the size of the pools until paging becomes imminent, when the unused memory is within a few percent of the total memory available. Allocating pool space to different instances of the DBMS, for example, operational, data warehouse, application test, system test purposes, is a complex trade-off. Pool size is a powerful priority mechanism.

Some platforms, for instance, IBM iSeries, previously AS/400, do not support pools; all pages are considered equal by the operating system. The main algorithm is the LRU. This eliminates a system tuning task, but also a means to set priorities.

The read cache of a disk server behaves much like memory without pools. The disk server decides which page is overlaid by a new page. The main algorithm used, unsurprisingly, is also the LRU, the main exception being a shorter cache holding time for sequentially read pages. Typically, the algorithms cannot be influenced by a systems administrator, but it may be possible to force a short cache hold time for the pages of selected objects.

This equal treatment in the use of the read cache may be an issue when the disk server contains pages from many systems. The number of 4K pages from an operational database may vary between 1 and 10 million, depending on the other load. A fairly large database buffer pool as in Figure 15.2 reduces the variation in the response times of the operational transactions.

Long Rows

For sequential read, the I/O time per row depends on the length of the row. If sequential read time is 40 MB/s, the I/O time per 4K page is 100 μs. Therefore, if there are 10 rows per page, the I/O time per row will be 10 μs. This is the default value we have used for the QUBE.

With rows longer than 400 bytes, the actual I/O time

```
I/O time per page / number of rows per page
```

can be significantly more than 10 μs, especially if there is a lot of distributed free space.

Slow Sequential Read

The coefficient for TS should be calibrated according to the measured peak hour read speed. If the measured speed is only 10 MB/s (a value quite possible with older disk drives) instead of the assumed 40 MB/s for the QUBE, the coefficient will be 0.4 ms instead of 0.1 ms.

WHEN THE ACTUAL RESPONSE TIME CAN BE MUCH SHORTER THAN THE QUBE

The purpose of the QUBE is to find potentially slow access paths with *minimal effort*. Because of the simplicity of the formula (only two *essential* variables are used—the number of random touches and the number of sequential touches), it is feasible to check every SQL call at an early stage.

When designing indexes using the QUBE, we need to keep a sense of proportion with regard to the changes being suggested for the indexes; for example, when choosing between a semifat and a fat index (perhaps because the additional disk space would be more than 10 GB) or between a fat index and an ideal index (perhaps because more than 10,000 rows are added to the table during peak

hour), it may be necessary to know how pessimistic the QUBE is for the cheaper alternative.

We will now discuss the *three most important occasions* when the *actual response time* can be *much shorter* than the *QUBE*.

Leaf Pages and Table Pages Remain in the Buffer Pool

The cost of a random touch is 10 ms, according to the QUBE. This would be close to the truth if *all* leaf and table pages were accessed randomly. With 100 million leaf and table pages in a database, 400 GB if the page size is 4K, and an access rate to these pages of 10,000 per second, the *average time between accesses to each page* would be

$$100{,}000{,}000/10{,}000 = 10{,}000 \text{ s} = 3 \text{ h}$$

In real applications, of course, some pages will be accessed much more often than the average page.

To simplify the discussion, let us call the combination of the database buffer pool in memory and the read cache in the disk server, the *pool*. A randomly read page may stay in the pool, say, for 10 min if it is not accessed. This means that a page that is accessed at least *once every 10 min* stays in the pool.

When is a leaf page or a table page accessed this often?

Small Tables

If a table has 100 equally popular pages and the table is accessed once a second, the average time between accesses *per page* is 100 s. If the pool residency time is 10 min, it is reasonable to assume that the table pages, as well as the leaf pages of the active indexes, will stay in the pool.

Hot Spots

A train seat reservation system held data for a period of 90 days. The main table was clustered by departure date. As most people reserved a seat on the day of departure, the last 1% of the table pages, as well as the last 1% of the clustering index, tended to stay in the pool. This eliminated many random reads.

When is a hot spot hot enough? If the pool residency time is *10 min*, the average time between accesses to a page should be *no more than 5 min* to take any variation into account. Thus, if there are 10 page accesses per second to the train seat table, the *hot spot area*, trains leaving today, should not be greater than 300 s $\times$ 10 pages/s = 3000 pages.

Comebacks

A program may do a random touch to a page several times. Is it likely that the page will still be in the pool? The answer will be yes if the time between accesses

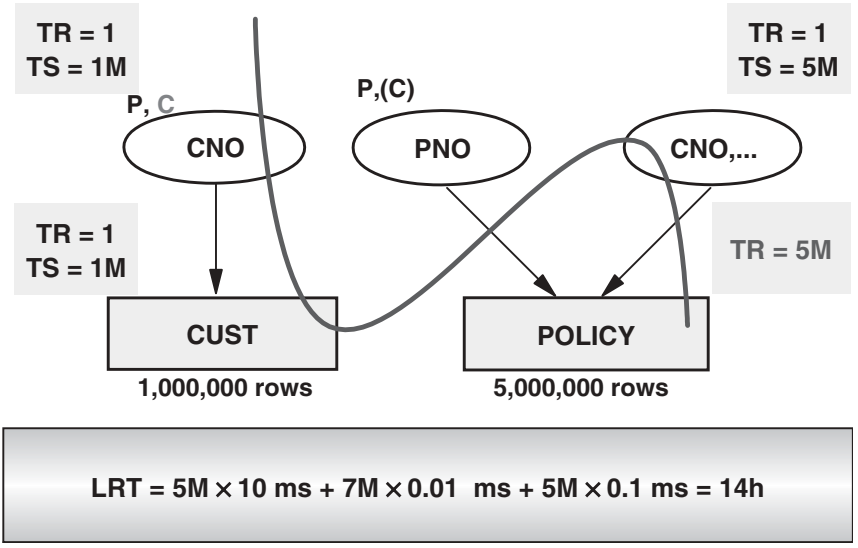


Figure 15.3 Batch program— comebacks.

is less than the pool residency time. This is why the pool residency time is very critical for massive batch jobs with many random touches.

Let's consider the batch job we discussed in Chapter 11. If the clustering index of table POLICY is index PNO as shown in Figure 15.3, there could be five million random touches to the table. Five million random touches take $5{,}000{,}000 \times 10$ ms $= 50{,}000$ s; 14 h, if every touch results in a random read from a disk drive. Let us assume that the size of the POLICY table is 1 GB. If the pool is significantly larger than 2 GB, each page in table POLICY is read from disk only once: $200{,}000 \times 10$ ms $= 2000$ s; 34 min. The remainder of the random touches, which now find the row in the pool, may each take 0.1 ms: $4{,}800{,}000 \times 0.1$ ms $= 480$ s; 8 min. Consequently, the time taken by the 5 million random touches is reduced from 14 to 42 minutes!

As the policy table has 250,000 pages, an average page is accessed 20 times. Thus, the average time between accesses to a page will be 2 min if the elapsed time of the program is about 40 min. The pool residency time must be at least 2×2 min $= 4$ min to ensure that it is unlikely that a page will be read more than once from disk.

Resident Tables and Indexes

Each page is read only once from disk after a system restart. The pages in the special pools are *never* overlaid if the pool is sized correctly; they remain in memory as long as the system stays up.

Identifying These Cheap Random Touches

When considering the first two categories (small tables and hot spots), it is important to have some idea of the pool residency time. This can be measured by a program that accesses a single page table every N seconds. The number of synchronous reads will reveal whether the page stayed in memory, and the average time per synchronous read will indicate whether the page stayed in read cache. The pool residency time varies considerably according to the workload. It is especially sensitive to massive batch jobs that perform a large number of random touches to a large table. Nevertheless, it should be possible to be able to find an order of magnitude for heavy loads. If the value is less than a minute, the pool is relatively small.

When considering the third category (comebacks), the size of the index or table should first be compared with the pool size. Is the difference large enough for the rest of the concurrent load? The second criteria is the average time between accesses to a page.

The fourth category (resident tables and indexes) is trivial, but the effect of restarting the system with an empty pool should not be forgotten. The optimizers, by the way, are not very good at *estimating pool hits*. They may not even see the fourth category.

Assisted Random Reads

We saw in Chapter 2 that there are several occasions when random reads will cost less than the figure assumed by the QUBE. We may now estimate how much faster the assisted random reads may be.

Skip-Sequential

Skip-sequential read means reading nonconsecutive rows in one direction, such as reading row 5, row 18, row 20. If the DBMS reads every other row, the time per row will be almost as short as with sequential read. On the other hand, if it reads three widely separated rows from a one-million-row table, the time per row will be almost as long as with random read.

There are two kinds of skip-sequential reads; we will call them *natural born* and *optimizer generated*. An example of the first kind is a singleton SELECT, which is repeatedly executed with host variable values that are in sequence but not consecutive. A second example, taken from Chapter 8, is discussed in more detail below. The second type occurs when the optimizer decides to collect a list of pointers from an index and sorts them before accessing the table rows. This can happen with single-table SELECTs using nonclustering indexes—list prefetch or during a join operation using a hybrid join, both features used by DB2 for z/OS.

The most important variable is the *average number of qualifying rows per page*. With skip-sequential, only the first row per page causes a disk read. The rest are equivalent to sequential touches.

Estimating Skip-Sequential I/O Time

When the number of skip-sequential touches (T) *exceeds* the number of leaf or table pages (P), the elapsed time for the scan can be estimated by:

```
P × 10 ms + (T - P) × 0.01 ms
```

P represents the number of pages within the index or table slice. The formula is based on the assumption that one page is read at a time—10 ms per page. This can be a very pessimistic assumption when few pages are skipped and the DBMS or disk system starts to read ahead; the I/O time per page can then be as low as 0.1 or 0.2 ms according to the page size (4 or 8K).

Let us apply this formula to the primary key index of table CUST shown in Figure 15.4. There are 20,000 skip-sequential touches when the rows of table INVOICE are in CNO sequence. Using the above formula, this will take

$$2500 \times 10 \text{ ms} + (20{,}000 - 2500) \times 0.01 \text{ ms} = 25 \text{ s}$$
$$(\text{QUBE} : 20{,}000 \times 10 \text{ ms} = 200 \text{ s})$$

With assisted random read, the actual time can be even shorter. This was observed when the join case study scenarios were measured on different platforms. DB2 for z/OS, for instance, started to read 32 pages ahead after a few single-page I/Os were performed (dynamic prefetch). This is why 100,000

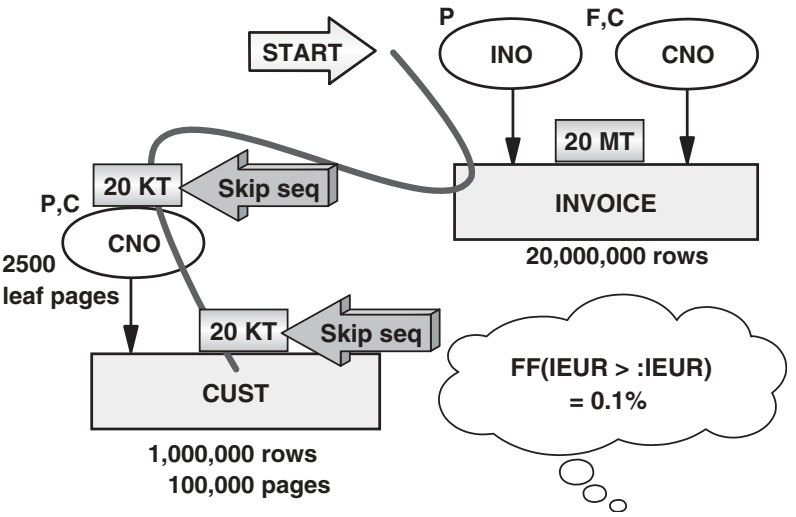


Figure 15.4 Natural born skip-sequential.

skip-sequential touches to an index with 20,000 leaf pages took only 13 s while the formula above predicted

$$20{,}000 \times 10 \text{ ms} + (100{,}000 - 20{,}000) \times 0.01 \text{ ms} = 201 \text{ s}$$

As the CUST table rows, as well as the index rows, are in CNO sequence, the table touches are skip-sequential; they access on average *every fifth page*; 20,000 touches from 100,000 pages, or 0.2 touches per page. Now the benefit is much smaller and becomes dependent on how consecutive pages are stored on disk, and how the disk server moves data from the disk drive into the read cache.

If the stripe size is 32K (eight 4K pages) and the disk server decides to read the whole stripe to the read cache (but not the next stripes), 1.6 (0.2×8) qualifying table rows are moved on average from drive to cache in one operation. The first touch per stripe takes about 10 ms, while the next touches per stripe take about 1 ms. The average time per touch is

$$\frac{1 \times 10 \text{ ms} + 0.6 \times 1 \text{ ms}}{1.6} = 7 \text{ ms}$$

The refined estimate for 20,000 skip-sequential touches is now

$$20{,}000 \times 7 \text{ ms} = 140 \text{ s}$$

compared to the QUBE estimate of $20{,}000 \times 10 \text{ ms} = 200$ s.

The benefit of skip-sequential would be greater in this case if the disk server decided to read a few stripes ahead (as with sequential processing).

As this example shows, skip-sequential read actually covers the whole spectrum between random read and sequential read. If random read is black and sequential read is white, skip-sequential is all shades of gray. This is particularly useful to remember when evaluating nested-loop joins, for instance. One thousand nonsequential touches, considered random in the QUBE, can be quite fast if the touches are skip-sequential.

The principles discussed above would apply equally well to List Prefetch (and to hybrid joins in DB2 for z/OS); the table rows are accessed skip-sequentially.

Data Block Prefetching

As Oracle reads an index slice, it may initiate multiple synchronous reads, as we saw in Chapter 2. Assuming data striping is in effect (as Oracle recommends), some of these reads will take place in parallel. The actual elapsed time for n random touches will now be less than 10 ms. If we assume a table is striped over three drives, the most optimistic estimate we could make for n random touches would be ($n \times 10$ ms)/3. The actual value would be somewhere between this and the QUBE.

Comment

As we have seen, the contribution of n random touches to the response time can be reduced significantly by the use of assisted random reads. Despite this, we

would strongly suggest that a much better approach would be the elimination of the random reads, as we have constantly emphasized throughout this book, because otherwise the *drive load* caused by the touches remains high.

Assisted Sequential Reads

In addition to the parallelism provided by striping, sequential read can be made faster by utilizing more than one processor. The implementation is platform specific; shared memory and disk storage, as used by DB2 for z/OS, or dedicated memory and disk storage by processor, as used by Oracle clusters. With n parallel processes the elapsed time for a sequential scan may approach T/n, where T is the elapsed time with one processor.

ESTIMATING CPU TIME (CQUBE)

The QUBE can be easily adapted to provide quick SQL CPU time estimates (CQUBE); RS represents the number of rows sorted:

$$\text{CPU time} = 100\ \mu\text{s} \times \text{TR} + 5\ \mu\text{s} \times \text{TS} + 100\ \mu\text{s} \times \text{F} + 10\ \mu\text{s} \times \text{RS}$$

We will call these numbers *the default CPU coefficients*.

Each of these components will be discussed in turn to see how the coefficients in the above formula can be measured and, in particular, to show the wide variation that arises from the many factors involved; platform, processor speed and cache, DBMS, buffer pools, SQL characteristics, column specifications, page size, and locking environment, to name but a few!

CPU Time per Sequential Touch

A sequential touch, as defined in the QUBE process, refers to the retrieval of a row within a sequential scan, and applying any nonmatching predicates (matching predicates will have first been used in the process of defining the thickness of the scan). With a full table scan and a full index scan, all predicates are nonmatching, of course.

The CPU time for a sequential touch depends on the length of the row because processing each page incurs a relatively high CPU cost. The number and complexity of the nonmatching predicates is another major factor. Other factors include locking and compression.

It is relatively easy to find a platform-specific *base* estimate for the CPU time per sequential touch; we will consider the approach used to provide the following measurements, which readers may wish to replicate. Three examples will be shown.

In our *first example*, two single-table SELECTS were written with WHERE clauses that were never true. The only predicate in each WHERE clause referred to a nonindexed column. Therefore, the optimizer was forced to choose a full

table scan for both SELECTs. One of the tables, CUST, had 1,000,000 long rows (400 bytes) in 111,000 4K pages; the other, INVOICE, had 4,000,000 short rows (80 bytes) in 77,000 4K pages.

The tables were not compressed and all the columns were of fixed length. The two single WHERE clauses were extremely simple. Lock size was page, isolation level CS. Additional predicates, particularly complex ones, would increase the CPU times as would longer INVOICE rows, variable length rows, and compression.

The results, using an old and busy server (100 mips per processor) were:

SQL CPU time for a full table scan

CUST (1 million rows, 111,000 4K pages): 2.5 s, 2.5 μs per row

INVOICE (4 million rows, 77,000 4K pages): 5 s, 1.25 μs per row

Assuming that the number of rows and the number of pages are the only *significant* factors, we can find the coefficients for these two variables:

$$1{,}000{,}000\ X + 111{,}000\ Y = 2.5\ s$$
$$4{,}000{,}000\ X + 77{,}000\ Y = 5\ s$$

$$X = \text{CPU time per row} = 1\ \mu s$$
$$Y = \text{CPU time per page} = 13\ \mu s$$

The relative figures for X and Y clearly show the impact that the number and length of the columns have on the CPU time.

Our *second example* used a large, busy server (440 mips per processor); a table with 13,000,000 short rows in 222,000 4K pages was scanned, again all rows being rejected. The table was compressed, and it contained variable-length columns, but again the single predicate was extremely simple.

The reported CPU time was 9 s, 0.7 μs per row.

In a *third example* using a small server (750 MHz per processor), the CUST table with 1,000,000 long rows (400 bytes) was scanned and 1078 rows FETCHed and sorted. The WHERE clause had two simple predicates. The reported CPU time was 10.4 s.

The TS CPU time was 10.4 s − X, where X is the CPU time for the FETCHes and the sort (1078 rows for each). Using our CPU time coefficients,

$$X = 1078 \times 110\ \mu s = 0.1\ s$$

Thus, the CPU time per sequential touch for the long rows turns out to be 10 μs.

It would seem that 5 μs is a reasonable default value to use, but it would be wise to calibrate the coefficient for each platform by measuring the CPU with both long and short rows.

CPU Time per Random Touch

There are several reasons why a random touch requires much more CPU time than a sequential touch:

1. Almost every random touch causes a page request.
2. The I/O-related CPU time, when the page is not in the buffer pool, tends to be high. The I/O cost of a sequential read per page is lower because of multipage reads. In addition, the cost per page is shared by many sequential touches.
3. Significant *memory waits* may occur because the CPU cannot prefetch pages into the high-speed CPU caches; by definition, random touches are unpredictable.
4. A random touch to an index, as defined by the QUBE, ignores the nonleaf pages because they are assumed to be in the buffer pool. For CPU time, however, accessing an index row in a three-level index randomly normally causes three page requests. This is why random index touches consume more CPU time than random table touches.

One way to easily determine the CPU time per random touch is simply to compare the CPU time before and after unproductive random touches are eliminated with semifat or fat indexes.

A second way would be to eliminate the cheaper components (e.g., sequential touch and sort). To illustrate this approach, a nested loop join SELECT with 813 random touches to a three-level index, had the following profile in an installation with 100 mips processors:

$$\begin{aligned} \text{TR} &= 814 \text{ (medium index rows)} \\ \text{Disk Reads} &= 845 \text{ (random reads from disk)} \\ \text{TS} &= 8130 \text{ (short index rows)} \\ \text{F} &= 814 \\ \text{RS} &= 0 \\ \text{CPU time} &= 401 \text{ ms} \end{aligned}$$

The CPU time per random index touch can be deduced by using the *installation-calibrated* coefficients for TS and F:

$$\frac{401 \text{ ms} - [(8130 \times 1\ \mu\text{s}) + (814 \times 50\ \mu\text{s})]}{814} = 434\ \mu\text{s}$$

$$\text{Converted to 250 mips} \qquad \frac{430\ \mu\text{s}}{2.5} = 173\ \mu\text{s}$$

This CPU time was measured in a test environment with a cold pool (no hits in the pool or disk cache, 845 random reads from disk). When the same transaction was executed again quite soon after the first measurement, the CPU time was 165 ms. In this case, assuming the CPU cost of the other components remains at 48 ms, the CPU time per random touch becomes

$$\frac{165 \text{ ms} - 48 \text{ ms}}{814} = 144\ \mu\text{s}$$

$$\text{Converted to 250 MIPS} \qquad \frac{144\ \mu\text{s}}{2.5} = 58\ \mu\text{s}$$

In the first case the average number of disk reads per random touch was $845/814 = 1.04$; in the second case there were no disk I/Os at all. This result supports a common and important observation: Eliminating random reads from a disk cache or drive saves a significant amount of CPU time. It also illustrates one of the factors that cause a large variation in the CPU time per random touch.

According to our experience, the CPU time consumed by a random touch is often in the following range with current hardware:

Table touches: between 10 and 100 μs

Index touches: between 20 and 300 μs

The largest cause of this enormous variation with random touches appears to be the high-speed CPU cache. When a burst of random touches occurs to a small number of pages in memory, the CPU time per random touch may be even less than 10 μs; in fact, it can approach the level of sequential touches (a few microseconds).

Conclusion

The coefficient of 100 μs CPU per random touch still seems valid for quick estimates. However, because of the large variations, as discussed above, an appropriate range of values should be used whenever an important decision, such as table denormalization or adding a new index, is being made based on CPU time estimates. For instance, the coefficient could be assumed to be between 100 and 300 μs for truly random touches to a large index.

Unfortunately, this could well cause problems when estimating CPU potential savings; if an SQL statement with 10,000 random touches consumed 600 ms of CPU time, eliminating 9000 random touches couldn't possibly save 900 ms of CPU time. We would have to assess the proportion of the 600 ms that was due to TRs.

CPU Time per FETCH Call

The CPU time per FETCH call itself (excluding the touches) depends on the platform and the way in which the application program and the DBMS are connected. In multitier environments and with some transaction managers, the cost of shipping the SQL call to the DBMS (and shipping the result back) can consume from 50 to 100 μs of CPU time. In other environments, especially when the SQL calls are embedded in a batch program running in the database server, the CPU time per FETCH may be between 10 and 20 μs.

Figure 7.9 shows a transaction that issued approximately one million SQL calls with an efficient access path; most of the calls caused only one sequential touch. The total CPU time was about 100 s when the CPU time consumed by the transaction manager (CICS) is taken into account; about 100 μs per FETCH.

A FETCH call issued by a batch program may be much less expensive. The 100 μs coefficient is a default value that may be very pessimistic in some environments, as the following batch measurement shows.

In the example we used earlier, an installation with a large busy server (440 mips per processor) showed a measured TS CPU time of 0.7 μs by scanning a table and rejecting all the rows. The SELECT was repeated with every row being *accepted*; 13,000,000 rows in 222,000 4K pages, the SELECT list containing only one column.

The reported CPU time was 115 s, 8.8 μs per FETCH (115 s/13,000,000). The CPU time for each FETCH was therefore 8 μs (8.8 μs − 0.7 μs). This is the lower-bound figure; the CPU time becomes longer when many columns are selected. With a current medium server (250 mips per processor), the lower-bound figure would be (440 mips/250 mips) × 8 μs = 14 μs with only one column selected.

If readers wish to repeat this sort of measurement on their own platforms, they should be aware that a part of the FETCH-related CPU time may be reported on the transaction monitor reports and not on the DBMS monitor reports.

Some products now support a multirow FETCH:

```
FETCH NEXT ROWSET FROM cursor_x FOR 10 ROWS
   INTO :hv1, :hv2, ...:hv10
```

This may reduce the CPU time for the application program interface (API) by up to 50% if a small number of columns is selected. Thus, the API CPU time for the 10-row FETCH above could be estimated as 10 × 0.1 ms = 1 ms/2.

CPU Time per Sorted Row

The sort CPU time appears to be reasonably close to linear as long as memory pools are large enough to eliminate disk I/O. The platform-specific coefficient is therefore easy to determine by measuring a SELECT that causes a fairly large sort, at least 10,000 rows, and then some time later, measuring the same SELECT without ORDER BY. The rule of thumb is 10 μs per sorted row, so the expected difference in CPU time with 10,000 sorted rows would be 100 ms.

CPU ESTIMATION EXAMPLES

The ability to provide estimates of CPU requirements will be found to be useful in many decision-making processes. Throughout this book we have made comparisons of various sorts based purely on elapsed times. One such example was the point at which an optimizer may decide to use a table scan as opposed to an index scan. Based on the QUBE figures of 10 ms per TR and 0.01 ms per TS, the cut-off point would be a filter factor of 0.1%. The issues discussed at the beginning of this chapter, such as comebacks and skip-sequential, make this point much more difficult to ascertain. The CPU cost involved would also be

an important consideration. The following comparisons are some of the more important ones we have considered; we will now assess the CPU issues.

Fat Index or Ideal Index

This example from Chapter 6 showed that an ideal index could not really be justified in terms of the elapsed time; in the summary we stated that:

> *Although there would be a further small advantage to be gained in using the three-star index, particularly for the empty result table case, this would not be significant and it would incur the overheads associated with a new index.*

The QUBEs for the Fat and the Best indexes were:

```
Fat index: empty result set

Index LNAME, FNAME, CITY, CNO   TR = 1      TS = 10,000
Fetch 0 × 0.1 ms

LRT                             TR = 1      TS = 10,000
                                1 × 10 ms   10,000 × 0.01 ms
                                10 ms + 100 ms + 0 ms = 110 ms

Best index: 20 result rows (1 screen)

Index LNAME, CITY, FNAME, CNO   TR = 1      TS = 20
Fetch 20 × 0.1 ms

LRT                             TR = 1      TS = 20
                                1 × 10 ms   20 × 0.01 ms
                                10 ms + 0.2 ms + 2 ms = 12 ms
```

If we now determine the CQUBE for the fat index and the best index:

$$\text{Fat index} \quad 100\ \mu s \times 1 + 5\ \mu s \times 10{,}000 + 100\ \mu s \times 0 + 10\ \mu s \times 0 = 50\ \text{ms}$$

$$\text{Best index} \quad 100\ \mu s \times 1 + 5\ \mu s \times 20 + 100\ \mu s \times 20 + 10\ \mu s \times 0 = 2\ \text{ms}$$

we can see clearly that from the CPU point of view, according to our CPU coefficients, the ideal index would be 25 times better. Figure 15.5 shows a comparison, LRT and CPU, for all the indexes considered in that example.

A new index would increase the CPU time for INSERT CUST and DELETE CUST by only a fraction of a millisecond (one random touch and the write-related CPU time). Therefore, the ideal index, alternative 3 in Figure 15.5, may well be justified by the CPU time saving if the SELECT is executed frequently.

Nested-Loop Join (and Denormalization) or MS/HJ

The finalists in the join case study in Chapter 8 were

1. A BJQ join with nested loop (with table denormalization)
2. Ideal indexes for MS/HJ (no denormalization)

① Semifat ② Fat ③ Best ***

	Index	QUBE (Worst input)	
		LRT	CPU
	LNAME, FNAME	100 s	1s
①	LNAME, FNAME, CITY	0.2 s	54 ms
②	LNAME, FNAME, CITY, CNO	0.1 s	50 ms
③	LNAME, CITY, FNAME, CNO	0.01 s	2 ms

Figure 15.5 Save CPU time with a three-star index.

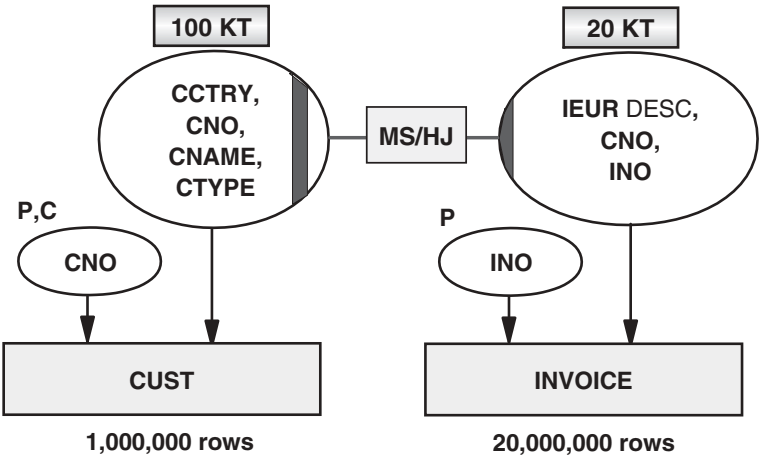


Figure 15.6 MS/HJ CPU time.

The response time with alternative 2 (the second finalist) was quite good: 1.8 s for the worst input that produced a large result, 1000 rows. The only issue with this alternative is the CPU time; after all, scanning the two index slices requires 120,000 touches and then $20{,}000 + 100{,}000$ rows had to be matched either by sort/merge or by hashing; refer to Figure 15.6. Before making a decision it would be sensible to make an estimate of the CPU times for alternative 2 to make quite sure.

As a comparison, the CPU time per screen for alternative 1 is very short:

$$\text{TR} = 1 + 20 = 21, \text{TS} = 20 + 0 = 20, \text{F} = 20, \text{no sort}$$

Thus the CPU time $= 21 \times 100\ \mu\text{s} + 20 \times 5\ \mu\text{s} + 20 \times 100\ \mu\text{s} = 4$ ms (per screen).

Alternative 2 with Merge Scan

Step 1: Access index (CCTRY, CNO, CNAME, CTYPE)

```
TR = 1   TS = 100,000
CPU time = 100 µs × 1 + 5 µs × 100,000 = 500 ms
```

Step 2: Access index (IEUR DESC, CNO, INO)

```
TR = 1   TS = 20,000
CPU time = 100 µs × 1 + 5 µs × 20,000 = 100 ms
```

Step 3: Merge and FETCH the result rows

The outer scan on CUST provides the rows in the required sequence. Therefore, the resulting CPU time is

CPU time for Sort and Merge: $2 \times 20{,}000 \times 0.01$ ms $= 0.4$ s

CPU time for Fetch: 2000×0.1 ms $= 0.2$ s

CPU Time With the worst input, the CPU time with this access path is the sum of the three components:

$$0.5 \text{ s} + 0.1 \text{ s} + 0.6 \text{ s} = 1.2 \text{ s}$$

Alternative 2 with Hash Join

Steps 1 and 2 are the same as with merge scan—CPU time $= 0.5$ s $+ 0.1$ s $=$ 0.6 s

Step 3: Match with hashing and FETCH the result rows

The optimizer hopefully chooses to start by building a hash table for the 20,000 short rows from the INVOICE table. The size of that table may be $20{,}000 \times 30$ bytes $= 0.6$ MB. This is small enough to load into memory in one go, but too large to stay in a 1-MB CPU cache in a production environment.

After the hash table has been built, the CUST index slice is scanned (estimated in step 1). The matching requires 100,000 random touches to the hash table; the average CPU time per touch depends on the CPU cache hit ratio. Given a 1-MB CPU cache, let us assume a range of 1 to 50 µs for the CPU time per hash table touch. Now the CPU time for 100,000 hash table touches is

$$100{,}000 \times (1 \ldots 50\ \mu\text{s}) = 0.1 \text{ s}, \ldots, 5 \text{ s}$$

Finally the result rows are obtained:

$$\text{CPU time for Fetch} \qquad 2{,}000 \times 0.1 \text{ ms} = 0.2 \text{ s}$$

If the hashing cost is high, the optimizer should use merge scan, as the CPU time estimate for the sort/merge phase was 0.4 s. An appropriate MERGE hint could be used in that case, if necessary.

CPU Time With the worst input, the CPU time with this access path is the sum of the three components:

$$\text{At least } 0.5 \text{ s} + 0.1 \text{ s} + 0.3 \text{ s} = 0.9 \text{ s}$$

Conclusion

Alternative 1 costs 4 ms of CPU time *per screen*. Alternative 2 costs about 1 s of CPU time for the maximum result (1000 rows, 50 screens). With small results (small countries), the CPU time with merge scan approaches 0.6 s (scanning, sorting, and merging 20,000 INVOICE index rows), while with hash join the CPU time approaches 0.2 s (scanning 20,000 INVOICE index rows).

The estimated CPU times for alternative 2 are probably acceptable if the SELECT is not executed very frequently. In extreme cases, two cursors could be written, one with a MERGE hint (for large countries) and one with a HASH hint (for small countries). *This is a small price to pay for avoiding table denormalization.*

Merge Scan and Hash Join Comparison

We considered this comparison in Chapter 8, but now that we have discussed the CPU issues, it would be appropriate to revisit this question.

In a university installation (933 MHz per processor, no concurrent load) a hash join was measured with the customer and invoice tables we used earlier: 5000 CUST rows were read with random touches via an index (FF CUST 0.5%) into the hash table in memory; 4,000,000 INVOICE rows were then read using a full table scan. The reported CPU time was 4.5 s; the CPU time per sequential touch was therefore less than 1.1 μs per row.

These are the main components of CPU time in this case:

4,000,000 sequential touches to table INVOICE

5000 random touches to table CUST

8000 random touches to the hash table (Invoice FF = 0.2%, 0.2% of 4 million rows = 8000 rows)

For each qualifying INVOICE row, the DBMS must access the hash table with the hashed CNO value (the join predicate CUST.CNO = INVOICE.CNO). The size of the hash table is about 5000 × 100 bytes = 0.5 MB. As the measurement was made in a stand-alone environment, that area could reside in the CPU cache for the duration of the INVOICE table scan. In this case, the CPU time per random touch to the hash table could be much less than 100 μs. If the random touches to table CUST take 100 μs and the sequential touches to INVOICE take 1 μs, the CPU time would be

4,000,000 sequential touches to table INVOICE	$4,000,000 \times 1\ \mu s = 4\ s$
5000 random touches to table CUST	$5000 \times 100\ \mu s = 0.5\ s$
8000 random touches to the hash table	$8000 \times X\ \mu s = 8X$ ms
40 FETCHes	$40 \times 100\ \mu s = 4$ ms

Because the hash table activities (building the table followed by 8000 random touches) do not contribute nearly as much to the CPU time as the 4 million sequential touches, we cannot estimate the hash table CPU time from this measurement. We can only speculate, based on the above discussion about the CPU cache, that it could be as low as a few microseconds per random touch.

With a merge scan, the 8000 random touches to the hash table would be replaced by a sort of 13,000 rows (CPU time estimate $13{,}000 \times 10\ \mu s = 0.1$ s) followed by a merge of 5000 rows and 8000 rows (CPU time estimate 0.1 s). Merge scan would use more CPU time than hash join in this case if the random touches to the hash table were to take less than 25 μs ($X < 25$).

The situation is different when the hash table is large and when many concurrent programs are sharing the high-speed CPU cache. Let us assume a 10% filter factor for both local predicates, CUST and INVOICE. Now the size of the hash table could be $100{,}000 \times 100$ bytes = 10 MB. With a 1-MB CPU cache shared by many concurrent users, cache hits would be rare. The CPU time per random touch could be as high as 50 μs (as a random table touch without I/O) at peak times. Then 400,000 random touches to the hash table could consume $400{,}000 \times 50\ \mu s = 20$ s. Now merge scan would be a better choice because the CPU time estimate for sorting and merging $100{,}000 + 400{,}000$ rows is $2 \times 500{,}000 \times 10\ \mu s = 10$ s. The difference increases if the hash table becomes so large that the optimizer decides not to place it all in memory at one time; it will then break it into partitions; with 10 partitions, the qualifying INVOICE rows would be scanned 10 times.

The choice between hash join (if available) and merge scan is a problem that the optimizer must try to resolve; the high-speed CPU cache makes it difficult for the optimizer to predict the CPU time of a hash join.

This is why the products that provide both alternatives make hints available that enable us to make the decision. From the index design point of view, as we mentioned in Chapter 8, the only difference between HJ and MS is the sort avoidance consideration available with MS.

On the other hand, the choice between indexes designed for NLJ vs. MS/HJ is a different consideration. Especially with non-BJQ queries, MS/HJ often has a *lower elapsed time* but a *higher CPU time*.

Skip-Sequential

We suggested earlier that one way to easily determine the CPU time per random touch is simply to compare the CPU time before and after unproductive random touches are eliminated with semifat or fat indexes.

We also discussed the issue of skip-sequential reads earlier in this chapter, noting that:

> *If random read is black and sequential read is white, skip-sequential is all shades of gray.*

The following measurements were made for just one "gray" example. In an installation with 100 mips processors, a SELECT had this profile when the index used wasn't even semifat.

$$\begin{aligned} \text{TR} &= 10{,}000 \\ \text{TS} &= 10{,}000 \\ \text{F} &= 1000 \\ \text{RS} &= 1000 \\ \text{CPU time} &= 275 \text{ ms} \end{aligned}$$

A semifat index reduced TR to 1000, the eliminated random touches being skip-sequential, while TS, F, and RS remained the same. The CPU time was reduced by over half to 124 ms; the CPU time per skip-sequential read in this case would seem to be $(275 - 124 \text{ ms})/9000 = 17$ μs, illustrating the benefits that skip-sequential provides.

CPU Time Still Matters

Reducing the number of touches—all kinds of touches, not only the heavy random touches—can bring CPU time down significantly. While the random read time is normally the dominating component of the response time, saving CPU time is still important in many installations. Mainframe CPU time may cost U.S.$1000 per hour—largely because the monthly charge for many software licences depends on the installed mips (CPU power). With LINUX, UNIX, and Windows platforms, the cost of CPU time is much lower as we have seen, but saving CPU time may be important to avoid hitting a server's maximum processor capacity. Splitting the load across several servers always introduces CPU and storage overheads. It also increases the risk of lock waits.

Chapter 16

Organizing the Index Design Process

- How the index design process may be organized with regard to the responsibilities of the people involved
- Use of computer-assisted index design tools
- Their objectives and their strengths together with their shortcomings
- Summary of the basic estimation process
- Nine steps toward excellent indexes

INTRODUCTION

At the beginning of the database age, index design was treated as a centralized function within the specialist group. The applications used to be so simple that a single database specialist was able to become familiar with all the processing requirements. Today this is seldom possible.

These days, each application programmer would evaluate the current indexes when writing a new SQL statement, at least with the basic question; ideally they should also use the QUBE to check the performance with the worst input. If the performance appears to be unsatisfactory, they could consider any required amendments. If necessary, the SELECT statement could be referred to a database specialist, possibly with the proposed additional columns to an existing index, or perhaps even with a suggestion for a new index. The final decision could be centralized, at least with integrated applications at the corporate level; the specialists are in the best position to estimate and control the *overall* costs and side effects.

Delegating the responsibility for reacting to index alarms certainly leads to the best results. Many organizations feel, however, that the required education and mentoring (about 1 to 2 weeks) is too large an investment.

A reasonable compromise in large organizations would seem to be to nominate 50/50 specialists; they would spend 50% of their time in application development and 50% in assisting their colleagues in index evaluation and other

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

performance issues (such as reading the EXPLAIN output to solve any optimizer problems). Experience has shown that one 50/50 specialist per 5 to 10 application developers works quite well.

Index design requires technical skills and application knowledge. It is probably easier to teach indexing skills to application developers than to make database specialists familiar with the details of the application.

COMPUTER-ASSISTED INDEX DESIGN

It is surprising that index design tools did not appear on the market until the nineties. The first tools were rule-based, but many current tools use the optimizer to evaluate alternative indexes. The objective of these tools is to find a set of indexes that minimizes the average response time for a recorded workload (refer to Fig. 16.1).

It is generally quite fast to find the best index for a given SELECT, but this approach may lead to too many indexes. The tool must then find the least valuable indexes. This is obviously a very time-consuming task if there are many different SELECT statements. Think about how many cost estimates would be needed to evaluate the effect of dropping a single index. As the tool developers say, it is a *huge search space*. Heuristic shortcuts are necessary to keep the execution time within acceptable limits. These may indeed affect the quality of the proposal, as Figure 16.2 suggests. Furthermore, the optimal index set is not necessarily a subset of the ideal indexes.

Many current tools are like chess computers—it is possible to set limits for the duration of the think time. The result is then the best proposal that the tool is able to produce within the given time.

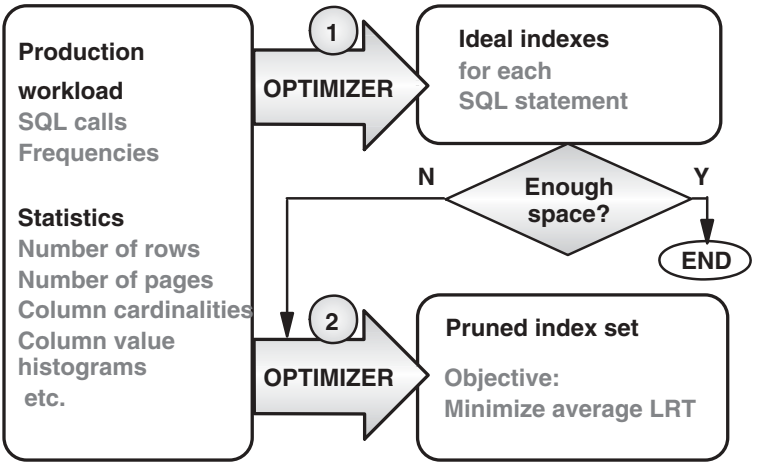


Figure 16.1 Computer-assisted index design.

Proposal quality	Probably good	Variable
Execution time	Probably acceptable	Very long

Figure 16.2 Pruning an index set may be extremely time consuming.

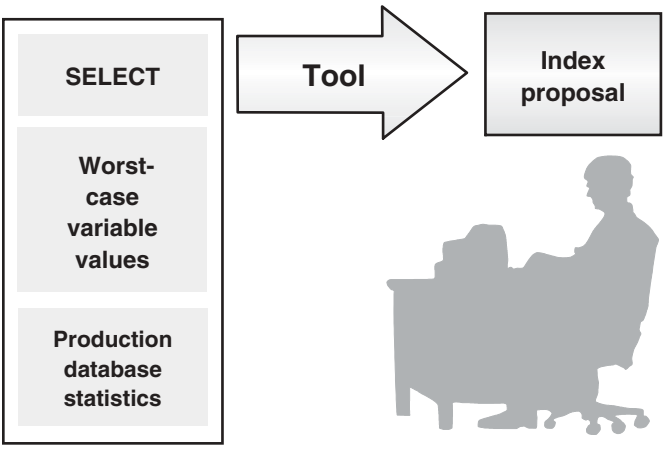


Figure 16.3 Realistic approach.

For the time being, a realistic approach (as shown in Fig. 16.3) appears to be to give the index design tool one SELECT statement at a time. The tool is likely to produce the best possible indexes even for a join accessing several tables—provided the worst input values are entered correctly instead of host variables; this, of course, is not a trivial task—remember the filter factor pitfall. What remains to be done then, is to make the compromises based on how many indexes each table tolerates and to choose the indexes for each table, possibly combining some of the proposed indexes.

A summary of the basic approach to index design is shown in Figure 16.4.

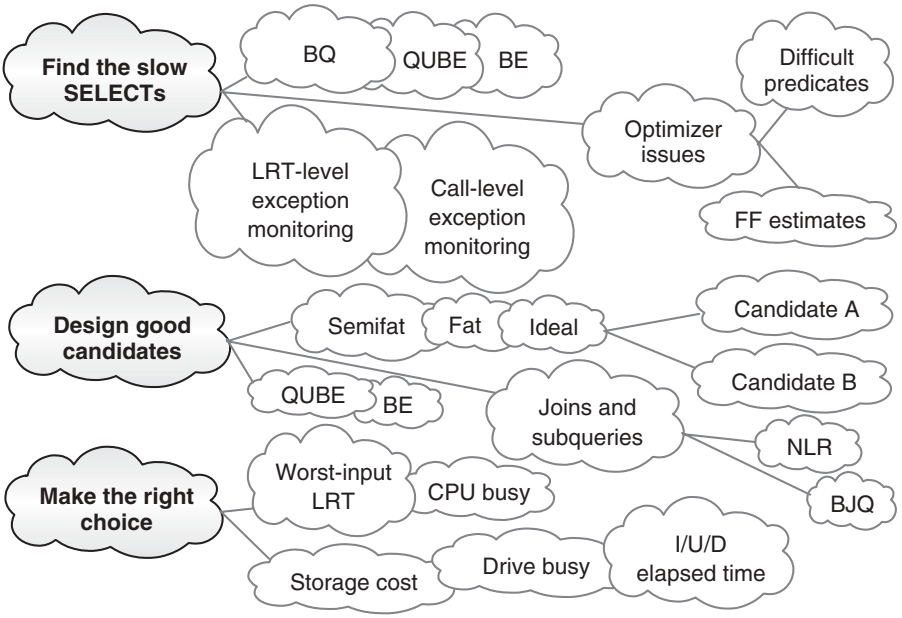


Figure 16.4 Summary of the basic approach to index design.

NINE STEPS TOWARD EXCELLENT INDEXES

1. When the first version of the table design is completed (primary keys, foreign keys, table row order), create the *version 0 indexes*: primary key indexes, foreign key indexes, and candidate key indexes, if any.
2. Check the performance of the first version of the table design: Using the QUBE, estimate the elapsed time of a few heavy transactions and batch programs with *ideal indexes*. If the estimates are not satisfactory, combine tables that have a 1 : 1 or 1 : C (1 to 0 or 1) relationship and add redundant data to the dependent tables in a 1 : M (one to many) relationship.
3. When the table structure appears stable, you may add any obvious indexes—based on knowledge of the application.
4. If a table is extremely volatile (say, more than 50 rows inserted, replaced, or deleted per second), you should estimate, using the QUBE, how many indexes the table tolerates.
5. When the database processing of a program (transaction or batch) is known, calculate the worst input QUBE using the latest database version. If the estimated local response time of a *transaction* exceeds the application-specific alarm limit (e.g., 2 s), the current database version is not satisfactory for that program. The acceptability of the estimated

elapsed time for a *batch* program must be evaluated case by case; but, to prevent long lock waits, the alarm limit for the elapsed time between two commit points should be the same as the alarm limit for local response time. When an alarm limit is exceeded, you should improve the indexing (semifat index, fat index, or ideal index). If the estimate (QUBE) is unsatisfactory even with ideal indexes (or if the number of required indexes exceeds the table-specific maximum number of indexes, set in step 4), you should make a more accurate estimate of the slow SELECTs, based on the issues discussed in Chapter 15. If the estimated elapsed time is then still too long you must resort to changing the table design as in step 2. In the worst case, you must negotiate the specifications with the users or the hardware configuration with the management.

6. When the SQL statements are written, the programmer should apply the basic question (BQ) or, if applicable, the basic join question (BJQ).
7. When the application is moved to the production environment, it is time to do a quick explain review: analyze all SQL calls that cause a full table scan or a full index scan. This review may reveal inadequate indexing or optimizer problems.
8. When production starts, create an LRT-level exception report (spike report or equivalent) for the first peak hour. When a long local response time is not due to queuing or an optimizer problem, you should proceed as in step 5.
9. Produce an LRT-level exception report at least once a week.

Variation 1

If there is no time to do a QUBE for all programs, the estimations may be limited to suspicious programs (such as browsing transactions and massive batch programs).

Variation 2

If LRT-level exception reporting is not possible, call-level exception reporting should be used with a low threshold (say, 100 ms elapsed time). In addition, visibly slow transactions should be analyzed with the QUBE if their SQL calls are not caught by the call-level exception report.

Note

A table diagram is sufficient in step 2; a complete CREATE TABLE statement is not needed. Therefore, index design can begin as soon as the first program has been specified. Subsequent programs may require additional table and index columns.

Note 2

In steps 2 to 5, the specification of the database processing must include the same information as the corresponding SQL statements, but the formulation of the SQL (e.g., join or subquery) is not required. At this point one should assume that the optimizer will choose (or can be made to choose) the assumed access path.

References

1. Peter Gulutzan and Trudy Pelzer, *SQL Performance Tuning*, Addison-Wesley, Reading, MA, 2002.
2. Mark Gurry, *Oracle SQL Tuning Pocket Reference*, O'Reilly, Sebastopol, CA, 2002.
3. Sharon Bjeletch, Greg Mable, Vipul Minocha, and Kevin Viers, *Microsoft SQL Server* 7.0 *Unleashed*, SAMS Publishing, Indianapolis, IN, 1999.
4. C. J. Date with Colin J. White, *A Guide to DB2*, Addison-Wesley, Reading, MA, 1990.
5. Marianne Winslett, Jim Gray Speaks Out, *www.acm.org/sigmod/record/issues/0303/Gray_SIGMOD_Interview_Final*.
6. Amy Anderson and Michael Cain, *www-1.ibm.com/servers/enable/site/bi/strategy/index.html*.
7. Dennis Sasha and Philippe Bonnet, *Database Tuning—Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann Publishers, San Francisco, CA, 2003.
8. Ari Hovi, *Data Warehousing*, Tietovarastotekniikka, Talentum, Helsinki, 1997.
9. Donald K. Burleson, *www.dba-oracle.com/art_dbazine_idx_rebuild.htm*.

Glossary

INDEX DESIGN APPROACH

The following terms and abbreviations are those we have used for the index design approach described throughout this book. They are not the official terms used by any of the database management systems in use today.

Algorithm for deriving the best index A fat index (third star) is designed, where the scanned index slice is as thin as possible (first star). If this index does not imply a sort (second star), it is a three-star index. Otherwise, it will only be a two-star index, having sacrificed the second star. Another candidate should then be derived that eliminates the sort, thereby having the second star but having sacrificed the first star. One of the resulting two star indexes will then be the best possible index for the given SELECT statement.

Assisted random reads A single term used *within this book* to represent automatic skip-sequential, list prefetch, and data block prefetching.

Assisted sequential reads A term used *within this book* to represent splitting a cursor into several range-predicate cursors, each of which will scan one slice; when several processors and disk drives are available, the elapsed time will be reduced accordingly.

BJQ, Basic Join Question Is there an existing or planned index that contains all the local predicate columns (this includes the local predicates for *all* the tables involved)?

BQ, Basic Question Is there an existing or planned index that contains all the columns referenced by the WHERE clause (a semifat index)?

Best index The best index that can be designed for a SELECT statement. It may have all three stars, in which case it is also an ideal index; it may, however, have only two stars, having sacrificed the first or the second star because of the presence of an ORDER BY together with range predicates.

Call-level exception monitoring Producing exception reports that show the slowest SQL calls in a monitoring period; they can be of great assistance in finding the slow SQL calls in a slow program that has a large number of different SQL statements.

Candidate A and B The two possible indexes used in the algorithm for deriving the best index.

CPU coefficients The values used to calculate the CPU time for random and sequential touches, FETCH, and sort.

CQUBE, CPU Quick Upper-Bound Estimate A quick, rough estimate of the SQL CPU time using four variables: TR, TS, F, and RS.

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach

Culprit A transaction that monopolizes resources, perhaps because of inadequate indexing.

DB2 for LUW A DBMS that runs under LINUX, UNIX, and Windows.

Difficult predicate A predicate that cannot participate in defining the index slice—that is, it cannot be a matching predicate; sometimes such predicates are called *nonindexable*.

Fat index An index that contains all the columns referred to in a SELECT statement; there is no table access necessary.

Filter factor pitfall A situation where the worst case may not be the input with the highest filter factor; it may arise when there is *no sort* in the access path, the transaction responds as soon as the first screen is built, and *all* the predicate columns do not participate in defining the index slice to be scanned.

First star The index rows required by the SELECT statement are next to each other, or at least as close to each other as possible. This star minimizes the thickness of the index slice that must be scanned.

Ideal index An index having all three stars.

Index slice The portion of an index that is scanned; the thickness of the slice influences the number of synchronous reads that will be required to the table.

LRT-level exception monitoring Producing exception reports that show operational transactions whose local response time or SQL elapsed time is exceptionally long.

NLR, number of local rows The number of rows remaining after the local predicates have been applied using the maximum filter factors. Used to predict the best table access order for a join—the outermost table is the one with the lowest NLR.

QUBE, Quick Upper-Bound Estimate A quick and easy estimate of the local response time using only two variables, TR and TS; it is used to reveal potentially slow access paths at a very early stage. It can reveal performance problems relating to index or table design, assuming that the worst-case filter factors used are reasonably close to reality. By definition the QUBE is pessimistic; it sometimes makes false alarms.

Really difficult predicate A predicate that cannot even participate in index screening; therefore, each table row in a slice has to be accessed to evaluate the predicate, even though the predicate column or columns have been copied to the index.

Second star The index rows are in the right sequence to satisfy the SELECT statement ORDER BY requirement. This star avoids the need for a sort.

Semifat index An index that contains all the columns referenced by the WHERE clause, thereby accessing the table only when necessary.

Spike A single operational transaction whose local response time or SQL elapsed time is exceptionally long.

Third star The index rows contain all the columns referred to by the SELECT statement. This star eliminates table access—the access path is index only; the third star is often the most important one.

Three-star index The ideal index for a given SELECT statement.

Touch The cost of the DBMS reading one index or table row. If the DBMS scans a slice of an index or table (the rows being read are physically next to each other), reading the first row infers a random touch. Reading the next rows takes one sequential touch per row. The cost of an index touch is essentially the same as the cost of a table touch.

Tuning potential The upper limit for the achievable reduction in the local response time as a result of planned index improvements.

Victim A transaction that is affected by a culprit because it has to wait for a resource.

GENERAL

Access path The method chosen by the optimizer to build a result table for an SQL statement; for a single-table SELECT, using a chosen index in a certain way or a full table scan; for a join, in addition to this, a table access order and a join method.

Asynchronous read Performed in advance while a previous set of pages is being processed; there may be a considerable overlap between the processing and the I/O time; ideally the asynchronous I/O will complete before the pages are actually required for processing. This activity is called prefetch.

Bitmap index Used instead of a B-tree index; each value of an index column has a bit vector. Appropriate for columns with a low cardinality provided that inserts, updates, and deletes are rare. Fast with certain kinds of queries with complex WHERE clauses, typically in data warehouse. Not supported by all products.

Block See **page**.

Boolean term predicate A row can be rejected when a predicate is evaluated false, otherwise it is non-Boolean. Non-BT predicates may make a WHERE clause too difficult for the optimizer—the access path is not optimal. If a WHERE clause contains no OR operators, all predicates are BT.

B-tree The most common type of index; columns are copied from (normally) a single table. The lowest level (leaf pages) contains a pointer to each table row. The leaf page level has its own index tree whose top level is called the root page.

Buffer pool An area of computer memory into which index and table pages are read from disk; the pool may be subdivided into subpools, which are allocated to individual indexes and tables. The buffer pool managers attempt to ensure that frequently used data remains in the pool to avoid the necessity of additional reads from disk.

Clustered index In SQL server, an index that contains the table rows; in DB2, any index whose index rows are stored in the same, or almost the same, order as the table rows.

Clustering index Causes the DBMS to insert a new table row into (or close to) a home page defined by the clustering index key. A table may have only one clustering index. Not all products support a clustering index, but a reorganization or reload can be used to place the table rows in the required sequence.

Covering index In SQL Server, an index that contains all the columns referred to in a SELECT statement so that table access can be avoided; this is the opposite of the

SQL Server term *table lookup* for an access path that uses an index, but also reads table rows.

CPU cache The CPU chip's high-speed memory used to store the most frequently used program instructions and data.

Cursor A construction in embedded SQL for moving the result table into the application program one row at a time with FETCH calls.

Database Logically related data; a collection of tables—DBMS dependent.

Data-partitioned secondary index Partitioning very large indexes, one index partition per table partition, in order to reduce unavailability.

Data block prefetching An Oracle term used to represent the process whereby the pointers are collected from an index slice so that multiple random I/Os can be started to read the table rows in parallel.

DBMS, database management system The software used to provide, manage, and control all aspects of the use of databases. Network and hierarchical systems have now almost entirely been superseded by relational systems.

Data warehouse A business environment that provides consistent and time-dependent data to support decision-making processes. Designed to support a large variety of unpredictable queries and reports. It is often loaded from multiple operational systems converted into a consistent format. It enables trend analysis, for example, comparisons of sales by month.

Default An assumed value used by the DBMS unless specifically overridden.

Denormalization Adding redundant data to a table; fairly common in data warehouse environments; also sometimes necessary in operational databases to improve the performance of SELECT statements.

Disk drive A rotating storage device that is able to carry out one read or write operation at a time.

Execution plan The output provided by the DBMS to describe an access path being used.

Fact table Contains detailed transaction data, for example, sales or payment entries such as sales figures, prices, or balances. This data is summarized or grouped with the help of dimension data.

Fat table A table into which columns have been added from another table or which is built by combining several tables together.

FETCH An SQL call used in cursor operations to request one row at a time.

Filter factor Specifies the selectivity of a predicate—what proportion of the table rows satisfy the condition expressed by the predicate. It is dependent on the distribution of the column values. When evaluating the adequacy of an index, *worst-case filter factors* are more important than average filter factors.

Foreign key A column or column combination that points to the primary key in another or same table. Foreign keys may have referential integrity constraints to ensure data integrity.

Free space To cater for new rows being added to tables and indexes, a certain proportion of each page may be left free when they are loaded or reorganized.

Hash join A join method where one table is first stored in a temporary table (local predicates having been applied), hashed by the join column(s); for each row of the other table that satisfies its local predicates, the temporary table is checked for matching rows, using the hash value.

Hint A specification given in an SQL call or a bind option to influence the optimizer; the syntax is product specific. It should only be used when the optimizer is not able to choose the best access path because of inappropriate cost estimates.

Host variable A program variable that is used in a WHERE clause, such as :SALARY in WHERE SALARY > :SALARY.

Index matching The use of predicates to restrict the size of an index slice to be scanned. One or more columns (sometimes called matching columns) identify the beginning of the slice and use the B-tree index structure to find the first index entry required. The columns will also determine the end of the scan. Matching predicates are sometimes called *range-delimiting* predicates.

Index only An access path that is able to provide all the data requested without requiring access to the table.

Index read-ahead An SQL Server term used to represent reading-ahead the next leaf pages following leaf page splits.

Index screening Index columns that cannot be used in the matching process may still be compared to the values in the predicates; table access then need only take place when it is necessary to do so.

Index skip scan An Oracle term used to represent reading several index slices instead of doing a full index scan.

Integrity A state of a database in which the defined constraints and rules for the data are valid.

I/O A request from the processor to read a page from disk or to write a page to disk following an update.

Join method The optimizer's decision about how to join tables together; the normal choice is the nested-loop join, although others are available.

Leaf page The lowest level of an index; the pages contain the key and pointer combinations arranged in key sequence.

Least recently used The algorithm normally used by buffer and disk server cache managers to identify which pages should be overwritten to satisfy new requests.

List prefetch A facility used by DB2 for z/OS to sort the index pointers of the required rows into page number sequence, so that the table rows may be accessed using skip-sequential.

Local response time Of a transaction excluding transfer and wait times between the work station and the server; the server response time.

Lock A construction for serialization processing to ensure logical integrity; normally relates to a table, page, or row.

Materializing result rows Performing the database accesses required to build the result set. In the best case, this simply requires a row to be moved from the database buffer pool to the program. In the worst case, the DBMS will request a large number of disk reads.

Merge scan A join method whereby one or more tables are sorted into a consistent order if necessary (after local predicates have been applied), and the qualifying rows in the tables or work files are merged (Oracle: *sort-merge join*).

Mirroring Writing all pages on two drives.

Multiblock I/O Oracle sequential read.

Multiple index access The collection and comparison of pointers from several indexes or indeed from several slices of a single index, followed by the access of the required table rows. Also called *index ANDing* (index *intersection*) and *index ORing* (index *union*).

Multiple serial read-ahead SQL server sequential read.

Multirow FETCH An SQL call used in cursor operations to request multiple rows at a time.

Nested loop A join method whereby the DBMS first finds a row in the outer table that satisfies the local predicates referring to that table. Then it looks for the related rows in the next table, the inner table, and checks which of these satisfy their local predicates and so on.

Nonleaf page Index pages other than leaf pages that contain a (possibly truncated) key value, the highest key together with a pointer, to a page at the next lower level.

Null An empty or unknown value; when storing the table row for which no value has been provided for a column, the DBMS stores a special indicator for null.

Optimizer A component of a relational database management system, which chooses the access path for each SQL statement. It estimates the cost of feasible access paths, usually based on a weighted sum of I/O time and CPU time.

Page Index and table rows are grouped together in pages (Oracle uses the term *block*); these are often 4 K in size, but other page sizes may be used. The page size will determine the number of index and table rows in each page. An entire page will be read from disk into a buffer pool and so several rows are read with a single I/O.

Predicate A search argument in the WHERE clause of an SQL statement.

Primary key A column or columns that uniquely identify a table row.

Query A request expressed in SQL that provides the rows satisfying the search arguments.

RAID 5 A redundant array of inexpensive disks level 5—a commonly used way to store data—logical volumes are striped over several disk drives that form a RAID array; the first 32 K stripe, for instance, is written to disk drive 1, the second to drive 2, and so on.

RAID 10 Appropriate for databases with frequent random inserts, updates, or deletes; actually RAID 0 mirroring + RAID 1 striping. Instead of redundant parity data, an updated page is written to two disk drives; a page can be read from either drive. Disk load (drive busy) caused by random writes is lower than with RAID 5 but more disk drives are needed.

Read cache An area in the semiconductor memory (RAM) of the disk server used to store the most recently read pages from the disk drive. The objective is to reduce the number of reads from the disk drives. Often much larger than the database buffer

pool in memory; could contain pages that have been read randomly during the last 20 min or so.

Redundancy Storing additional copies of data on a disk drive or in a table, for safety or performance reasons. RAID 5 redundancy means storing a parity bit for each bit on a stripe set (e.g., seven stripes, 32 K each). With these parity bits, any of the stripes can be reconstructed if a drive fails.

Relation The term used in the relational model for a table.

Relational database A database built with a relational DBMS according to a relational model.

Reorganization Indexes are reorganized to restore their correct physical order, important for the performance of index slice and full index scans. Tables are reorganized to restore free space and table row order.

Root page The top page in the B-tree index structure.

Row A table row must fit in one table page; an index row must fit in one leaf page.

Sequential prefetch DB2 sequential read.

Sequential read Multiple index or table pages read from disk into the buffer pool. Because the DBMS knows in advance which pages will be required, the reads can be performed before the pages are actually requested; this includes s*equential prefetch*, *multiblock I/Os*, and *multiple serial read-ahead reads.*

Service time This is normally the sum of the CPU time and the synchronous disk I/O time.

Skip-sequential A set of noncontiguous rows are scanned in one direction.

Striping RAID *striping* means storing the first stripe of a table or index (e.g., 32 K) on drive 1, the second stripe on drive 2, and so on, to balance the load on a set of drives. The disk server may read ahead from drives in parallel, so when the next set of pages are required, they are likely to be in the read cache of the disk server.

Summary tables Denormalized fact tables. Because of the denormalization, no joins are needed; all the required columns are in one table.

Synchronous I/O While the I/O is taking place, the DBMS is not able to continue any further; it is forced to wait until the I/O has completed.

Table The implementation of a relation, consisting of rows containing column data values.

Transaction A single interaction between user and program, consisting of one or more SQL calls that may be read only or updates—relates to the local response time.

Trigger A program module, stored in the database, which is started automatically by an SQL call accessing the table. Different SQL calls have their own triggers for inserts, deletes, and updates. They are normally written with SQL and a procedural extension, such as Transact-SQL (SQL server) or PL/SQL (Oracle).

View A virtual table that, although containing no data of its own, provides a subset of the table columns; defined by a create view statement containing a SELECT statement.

Write cache An area in the semiconductor memory (RAM) of the disk server where the data is stored, backed up with battery power (nonvolatile storage, NVS). The DBMS writes modified pages to a disk server perhaps a few times per second; they

are first stored in the write cache. The LRU pages are written to the disk drives. The write cache may contain the pages that have been modified during the last minute or so, and if a page is frequently modified it could stay in the write cache all day long. The bigger the cache, the lower the drive load (drive busy) caused by updates.

Index

Relational Database Index Design and the Optimizers, by Tapio Lahdenmäki and Michael Leach