Relational Database Index Design and the Optimizers

DB2, Oracle, SQL Server, et al.

Tapio Lahdenmäki
Michael Leach
Relational Database
Index Design and the
Optimizers
# Contents

**Preface** xv

**1 Introduction** 1

Another Book About SQL Performance! 1
Inadequate Indexing 3
Myths and Misconceptions 4

- Myth 1: No More Than Five Index Levels 5
- Myth 2: No More Than Six Indexes per Table 6
- Myth 3: Volatile Columns Should Not Be Indexed 6

Example 7
Disk Drive Utilization 7

Systematic Index Design 8

**2 Table and Index Organization** 11

Introduction 11

- Index and Table Pages 12

Index Rows 12
Index Structure 13
Table Rows 13
Buffer Pools and Disk I/Os 13

- Reads from the DBMS Buffer Pool 14
- Random I/O from Disk Drives 14
- Reads from the Disk Server Cache 15
- Sequential Reads from Disk Drives 16
- Assisted Random Reads 16
- Assisted Sequential Reads 19
- Synchronous and Asynchronous I/Os 19

Hardware Specifics 20
DBMS Specifics 21

- Pages 21
- Table Clustering 22
- Index Rows 23
3 SQL Processing

Introduction 29
Predicates 30
Optimizers and Access Paths 30
  Index Slices and Matching Columns 31
  Index Screening and Screening Columns 32
  Access Path Terminology 33
  Monitoring the Optimizer 34
  Helping the Optimizer (Statistics) 34
  Helping the Optimizer (Number of FETCH Calls) 35
  When the Access Path Is Chosen 36
Filter Factors 37
  Filter Factors for Compound Predicates 37
  Impact of Filter Factors on Index Design 39
Materializing the Result Rows 42
  Cursor Review 42
    Alternative 1: FETCH Call Materializes One Result Row 43
    Alternative 2: Early Materialization 44
    What Every Database Designer Should Remember 44
Exercises 44

4 Deriving the Ideal Index for a SELECT

Introduction 47
Basic Assumptions for Disk and CPU Times 48
Inadequate Index 48
Three-Star Index—The Ideal Index for a SELECT 49
  How the Stars Are Assigned 50
  Range Predicates and a Three-Star Index 52
Algorithm to Derive the Best Index for a SELECT 54
  Candidate A 54
  Candidate B 55
  Sorting Is Fast Today—Why Do We Need Candidate B? 55
5 Proactive Index Design 63

Detection of Inadequate Indexing 63
Basic Question (BQ) 63
Warning 64

Quick Upper-Bound Estimate (QUBE) 65
Service Time 65
Queuing Time 66
Essential Concept: Touch 67
Counting Touches 69
FETCH Processing 70
QUBE Examples for the Main Access Types 71

Cheapest Adequate Index or Best Possible Index: Example 1 75

Basic Question for the Transaction 78
Quick Upper-Bound Estimate for the Transaction 78
Cheapest Adequate Index or Best Possible Index 79
Best Index for the Transaction 79
Semifat Index (Maximum Index Screening) 80
Fat Index (Index Only) 80

Cheapest Adequate Index or Best Possible Index: Example 2 82

Basic Question and QUBE for the Range Transaction 82
Best Index for the Transaction 83
Semifat Index (Maximum Index Screening) 84
Fat Index (Index Only) 85

When to Use the QUBE 86
6 Factors Affecting the Index Design Process

I/O Time Estimate Verification 87
Multiple Thin Index Slices 88
  Simple Is Beautiful (and Safe) 90
Difficult Predicates 91
  LIKE Predicate 91
  OR Operator and Boolean Predicates 92
  IN Predicate 93
Filter Factor Pitfall 94
Filter Factor Pitfall Example 96
  Best Index for the Transaction 99
  Semifat Index (Maximum Index Screening) 100
  Fat Index (Index Only) 101
  Summary 101
Exercises 102

7 Reactive Index Design

Introduction 105
EXPLAIN Describes the Selected Access Paths 106
  Full Table Scan or Full Index Scan 106
  Sorting Result Rows 106
  Cost Estimate 107
  DBMS-Specific EXPLAIN Options and Restrictions 108
Monitoring Reveals the Reality 108
  Evolution of Performance Monitors 109
LRT-Level Exception Monitoring 111
  Averages per Program Are Not Sufficient 111
  Exception Report Example: One Line per Spike 111
  Culprits and Victims 112
  Promising and Unpromising Culprits 114
  Promising Culprits 114
  Tuning Potential 116
  Unpromising Culprits 120
  Victims 121
  Finding the Slow SQL Calls 123
8 Indexing for Table Joins

Introduction 135
Two Simple Joins 136
  Example 8.1: Customer Outer Table 137
  Example 8.2: Invoice Outer Table 138
Impact of Table Access Order on Index Design 139
  Case Study 140
  Current Indexes 143
  Ideal Indexes 149
  Ideal Indexes with One Screen per Transaction Materialized 153
  Ideal Indexes with One Screen per Transaction Materialized and FF Pitfall 157
Basic Join Question (BJQ) 158
  Conclusion: Nested-Loop Join 160
Predicting the Table Access Order 161
Merge Scan Joins and Hash Joins 163
  Merge Scan Join 163
  Example 8.3: Merge Scan Join 163
  Hash Joins 165
  Program C: MS/HJ Considered by the Optimizer (Current Indexes) 166
  Ideal Indexes 167
Nested-Loop Joins Versus MS/HJ and Ideal Indexes 170
  Nested-Loop Joins Versus MS/HJ 170
  Ideal Indexes for Joins 171
Joining More Than Two Tables 171
Why Joins Often Perform Poorly 174
  Fuzzy Indexing 174
  Optimizer May Choose the Wrong Table Access Order 175
  Optimistic Table Design 175
## Contents

| Designing Indexes for Subqueries | 175 |
| Designing Indexes for Unions | 176 |
| Table Design Considerations | 176 |
| Redundant Data | 176 |
| Unconscious Table Design | 180 |
| Exercises | 183 |

### 9 Star Join Considerations

| Introduction | 185 |
| Indexes on Dimension Tables | 187 |
| Huge Impact of the Table Access Order | 188 |
| Indexes on Fact Tables | 190 |
| Summary Tables | 192 |

### 10 Multiple Index Access

| Introduction | 195 |
| Index ANDing | 195 |
| Index ANDing with Query Tables | 197 |
| Multiple Index Access and Fact Tables | 198 |
| Multiple Index Access with Bitmap Indexes | 198 |
| Index ORing | 199 |
| Index Join | 200 |
| Exercises | 201 |

### 11 Indexes and Reorganization

| Physical Structure of a B-Tree Index | 203 |
| How the DBMS Finds an Index Row | 204 |
| What Happens When a Row Is Inserted? | 205 |
| Are Leaf Page Splits Serious? | 206 |
| When Should an Index Be Reorganized? | 208 |
| Insert Patterns | 208 |
| Volatile Index Columns | 216 |
| Long Index Rows | 218 |
| Example: Order-Sensitive Batch Job | 219 |
| Table Disorganization (with a Clustering Index) | 222 |
| Table Disorganization (Without Clustering Index Starting with CNO) | 223 |
Table Rows Stored in Leaf Pages 223

| SQL Server | 223 |
| Oracle     | 224 |

Cost of Index Reorganization 225

| Split Monitoring | 226 |
| Summary         | 227 |

### 12 DBMS-Specific Indexing Restrictions 231

- Introduction 231
- Number of Index Columns 231
- Total Length of the Index Columns 232
- Variable-Length Columns 232
- Number of Indexes per Table 232
- Maximum Index Size 232
- Index Locking 232
- Index Row Suppression 233
- DBMS Index Creation Examples 234

### 13 DBMS-Specific Indexing Options 237

- Introduction 237
- Index Row Suppression 237
- Additional Index Columns After the Index Key 238
- Constraints to Enforce Uniqueness 240
- DBMS Able to Read an Index in Both Directions 240
- Index Key Truncation 241
- Function-Based Indexes 241
- Index Skip Scan 242
- Block Indexes 243
- Data-Partitioned Secondary Indexes 243
- Exercises 244

### 14 Optimizers Are Not Perfect 245

- Introduction 245
- Optimizers Do Not Always See the Best Alternative 246
  - Matching and Screening Problems 246
  - Non-BT 247
  - Unnecessary Sort 250
  - Unnecessary Table Touches 251
Optimizers’ Cost Estimates May Be Very Wrong 252
- Range Predicates with Host Variables 252
- Skewed Distribution 253
- Correlated Columns 255
- Cautionary Tale of Partial Index Keys 256

Cost Estimate Formulas 259
- Estimating I/O Time 259
- Estimating CPU Time 261
  
- Helping the Optimizer with Estimate-Related Problems 261

Do Optimizer Problems Affect Index Design? 265
Exercises 265

15 Additional Estimation Considerations 267

Assumptions Behind the QUBE Formula 267

Nonleaf Index Pages in Memory 268
- Example 268
  
- Impact of the Disk Server Read Cache 269
  
- Buffer Subpools 270
  
- Long Rows 272
  
- Slow Sequential Read 272

When the Actual Response Time Can Be Much Shorter Than the QUBE 272
- Leaf Pages and Table Pages Remain in the Buffer Pool 273
  
- Identifying These Cheap Random Touches 275
  
- Assisted Random Reads 275
  
- Assisted Sequential Reads 278

Estimating CPU Time (CQUBE) 278
- CPU Time per Sequential Touch 278
  
- CPU Time per Random Touch 279
  
- CPU Time per FETCH Call 281
  
- CPU Time per Sorted Row 282

CPU Estimation Examples 282
- Fat Index or Ideal Index 283
  
- Nested-Loop Join (and Denormalization) or MS/HJ 283
  
- Merge Scan and Hash Join Comparison 286
  
- Skip-Sequential 287
  
- CPU Time Still Matters 288
16 Organizing the Index Design Process  289

Introduction  289
Computer-Assisted Index Design  290
Nine Steps Toward Excellent Indexes  292

References  295

Glossary  297
   Index Design Approach  297
   General  299

Index  305
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
Copyright © 2005 John Wiley & Sons, Inc.
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.
Chapter 1 Introduction

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](image_url)  
**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 × 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 × 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 × 4 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 $6 \times 30 \text{ ms} = 180 \text{ ms}$ 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 $2 \times 10 \text{ ms} = 20 \text{ ms}$. 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.
1. Detect SELECT statements that are **too slow** due to inadequate indexing

   Worst input: Variable values leading to the longest elapsed time

2. Design indexes that make all SELECT statements **fast enough**

   Table maintenance (INSERT, UPDATE, DELETE) must be fast enough as well

---

**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.