Database Tuning: Principles, Experiments, and Troubleshooting Techniques

by Dennis Shasha and Philippe Bonnet ISBN: 1558607536

Morgan Kaufmann Publishers ?2002 (415 pages)

Learn to improve transferable skills that will facilitate in tuning many database systems on numerous hardware and operating systems.

Table of Contents

Database Tuning—Principles, Experiments, and Troubleshooting Techniques

Dennis Shasha
Courant Institute of Mathematical Sciences
New York University
Philippe Bonnet
University of Copenhagen
MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER SCIENCE

AMSTERDAM BOSTON LONDON NEW YORK OXFORD PARIS SAN DIEGO SAN FRANCISCO SINGAPORE SYDNEY TOKYO

PUBLISHING DIRECTOR: Diane D. Cerra
ASSISTANT PUBLISHING SERVICES MANAGER: Edward Wade
SENIOR PRODUCTION EDITOR: Cheri Palmer
SENIOR DEVELOPMENTAL EDITOR: Belinda Breyer
EDITORIAL ASSISTANT: Mona Buehler
COVER DESIGN: Yvo Riezebos
COVER IMAGSE: Dr. Mike Hill (Cheetah) and EyeWire Collection (Violin)/gettyimages
TEXT DESIGN: Frances Baca
TECHNICAL ILLUSTRATION: Technologies ‘n’ Typography
COMPOSITION: International Typesetting and Composition
COPYEDITOR: Robert Fiske
PROOFREADER: Jennifer McClain
INDEXER: Ty Koontz
PRINTER: Edwards Brothers Incorporated

Morgan Kaufmann Publishers
An Imprint of Elsevier Science
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205, USA
http://www.mkp.com

07 06 05 04 03 03 5 4 3 2 1

Library of Congress Control Number: 2001099791
1-55860-753-6

This book is printed on acid-free paper.

Dedication

To Karen, who always sings in tune

D.S.

To Annie, Rose, and to the memory of
Louise and Antoine Bonnet

P.B.

AUTHOR BIOGRAPHIES

Dennis Shasha is a professor at NYU's Courant Institute, where he does research on database tuning, biological pattern discovery, combinatorial pattern matching on trees and graphs, and database design for time series.

After graduating from Yale in 1977, he worked for IBM designing circuits and microcode for the 3090. He completed his Ph.D. at Harvard in 1984.
Dennis has previously written *Database Tuning: A Principled Approach* (1992, Prentice Hall), a professional reference book and the predecessor to this book. He has also written three books about a mathematical detective named Dr. Ecco: *The Puzzling Adventures of Dr. Ecco* (1988, Freeman, and republished in 1998 by Dover), *Codes, Puzzles, and Conspiracy* (1992, Freeman), *Dr. Ecco's Cyberpuzzle* (2002, Norton). In 1995, he wrote a book of biographies about great computer scientists called *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists* (Copernicus/Springer-Verlag). Finally, he is a coauthor of *Pattern Discovery in Biomolecular Data: Tools, Techniques, and Applications* (1999, Oxford University Press). In addition, he has coauthored 30 journal papers, 40 conference papers, and 4 patents. He spends most of his time these days building data mining and pattern recognition software. He writes a monthly puzzle column for *Scientific American* and *Dr. Dobb's Journal*.

Philippe Bonnet is an assistant professor in the computer science department of the University of Copenhagen (DIKU), where he does research on database tuning, query processing, and data management over sensor networks.

After graduating in 1993 from INSA Lyon, France, he worked as a research engineer at the European Computer-Industry Research Center in Munich, Germany. He then joined GIE Dyade—a joint venture between INRIA and Bull—in Grenoble, France, and he obtained a Ph.D. from Universite de Savoie in 1999. He then joined Cornell University and is presently at DIKU. He is responsible for the public domain object-relational system Predator.

Foreword

by Jim Gray, *Microsoft Research*
Series Editor, Morgan Kaufmann Series in Data Management Systems

Each database product comes with an extensive tuning and performance guide, and then there are the after-market books that go into even more detail. If you carefully read all those books, then did some experiments, and then thought for a long time, you might write this book. It abstracts the general performance principles of designing, implementing, managing, and use of database products. In many cases, it exemplifies the design trade-offs by crisp experiments using all three of the most popular products (IBM's DB2, Oracle's Oracle, and Microsoft's SQLServer). As such, it makes interesting reading for an established user as well as the novice. The book draws heavily from the author's experience working with Wall Street customers in transaction processing, data warehousing, and data analysis applications. These case studies make many of the examples tangible.

For the novice, the book gives sage advice on the performance issues of SQL-level logical database design that cuts across all systems. For me at least, the physical database design was particularly interesting, since the book presents the implications of design choices on the IBM, Oracle, and Microsoft systems. These systems are quite different internally, and the book's examples will surprise even the systems' implementers. It quantifies the relative performance of each design choice on each of the three systems. Not surprisingly, no system comes out "best" in all cases; each comes across as a strong contender with some blind spots. The book can be read as a tutorial (it has an associated Web site at http://www.mkp.com/dbtune/), or it can be used as a reference when specific issues arise. In either case, the writing is direct and very accessible. The chapters on transaction design and transaction chopping, the chapters on time series data, and the chapters on tuning will be of great interest to practitioners.

Preface

Goal of Book

Database tuning is the activity of making a database application run more quickly. "More quickly" usually means higher throughput though it may mean lower response time for some applications.

To make a system run more quickly, the database tuner may have to change the way applications are constructed, the data structures and parameters of a database system, the configuration of the operating system, or the hardware. The best database tuners, therefore, can solve problems requiring broad knowledge of an application and of computer systems.

This book aims to impart that knowledge to you. It has three operational goals.

1. To help you tune your application on your database management system, operating system, and hardware.
2. To give you performance criteria for choosing a database management system, including a set of experimental data and scripts that may help you test particular aspects of systems under consideration.
3. To teach you the principles underlying any tuning puzzle.

The best way to achieve the first goal is to read this book in concert with the tuning guide that comes with your particular system. These two will complement each other in several ways.

- This book will give you tuning ideas that will port from one system to another and from one release to another. The tuning guide will mix such general techniques with system- and release-specific ones.
- This book embodies the experience and wisdom of professional database tuners (including ourselves) and gives you ready-made experimental case study scripts. Thus, it will suggest more ideas and strategies than you'll find in your system's tuning guide.
- This book has a wider scope than most tuning guides since it addresses such issues as the allocation of work between the application and the database server, the design of transactions, and the purchase of hardware.

NOTE TO TEACHERS Although this book is aimed primarily at practicing professionals, you may find it useful in an advanced university database curriculum. Indeed, we and several of our colleagues have taught from this book's predecessor, *Database Tuning: A Principled Approach* (Prentice Hall, 1992).

Suppose your students have learned the basics of the external views of database systems, query languages, object-oriented concepts, and conceptual design. You then have the following choice:

- For those students who will design a database management system in the near future, the best approach is to teach them query processing, concurrency control, and recovery. That is the classical approach.
- For those students who will primarily use or administer database management systems, the best approach is to teach them some elements of tuning.

The two approaches complement each other well if taught together. In the classical approach, for example, you might teach the implementation of B-trees. In the tuning approach, you might teach the relevant benefits of B-trees and hash structures as a function of query type. To give a second example, in the classical approach, you might teach locking algorithms for concurrency control. In the tuning approach, you might teach techniques for chopping up transactions to achieve higher concurrency without sacrificing serializability.

We have tried to make the book self-contained inasmuch as we assume only a reading knowledge of the relational query language SQL, an advanced undergraduate-level course in data structures, and if possible, an undergraduate-level course in operating systems. The book discusses the principles of concurrency control, recovery, and query processing to the extent needed.

If you are using this book as a primary text for a portion of your classes, we can provide you with lecture notes if you e-mail us at shasha@cs.nyu.edu or bonnet@diku.dk.

What You Will Learn

Workers in most enterprises have discovered that buying a database management system is usually a better idea than developing one from scratch. The old excuse—"The performance of a commercial system will not be good enough for my application"—has given way to the new realization that the amenities offered by database management systems (especially, the standard interface, tools, transaction facilities, and data structure independence) are worth the price. That said, relational database systems often fail to meet expressability or performance requirements for some data-intensive applications (e.g., search engines, circuit design, financial time series analysis, and data mining). Applications that end up with a relational database system often have the following properties: large data, frequent updates by concurrent applications, and the need for fault tolerance.

The exceptional applications cited here escape because they are characterized by infrequent bulk updates and (often) loose fault tolerance concerns. Further, the SQL data model treats their primary concerns as afterthoughts.

But many notable applications fall within the relational purview: relational systems capture aggregates well, so they are taking over the data warehouse market from some of the specialized vendors. Further, they have extended their transactional models to support e-commerce.

Because relational systems are such an important part of the database world, this book concentrates on the major ones: DB2, SQL Server, Oracle, Sybase, and so on. On the other hand, the book also explores the virtues of specialized systems optimized for ordered queries (Fame, KDB) or main memory (TimesTen).

You will find that the same principles (combined with the experimental case study code) apply to many different products. For this reason, examples using one kind of system will apply to many.

After discussing principles common to all applications and to most data base systems, we proceed to discuss special tuning considerations having to do with Web support, data warehousing, heterogeneous systems, and financial time series. We don't discuss enterprise resource planning systems such as SAP explicitly, but since those are implemented on top of relational engines, the tuning principles we discuss apply to those engines.

How to Read This Book

The tone of the book is informal, and we try to give you the reasoning behind every suggestion. Occasionally, you will see quantitative rules of thumb (backed by experiments) to guide you to rational decisions after qualitative reasoning reaches an impasse. We encourage you to rerun experiments to reach your own quantitative conclusions.

The book's Web site at http://www.mkp.com/dbtune contains SQL scripts and programs to run the experiments on various database systems as well as scripts to generate data. The Web site also details our setup for each experiment and the results we have obtained.

You will note that some of our experimental results are done on versions of DBMSs that are not the latest. This is unfortunately inevitable in a book but serves our pedagogical goal. That goal is not to imply any definitive conclusions about a database management system in all its versions, but to suggest which parameters are important to performance. We encourage you to use our experimental scripts to do your own experiments on whichever versions you are interested in.

Acknowledgments

Many people have contributed to this book. First, we would like to thank our excellent chapter writers: Joe Celko (on data warehouses) and Alberto Lerner (on performance monitoring). Second, we would like to thank a few technical experts for point advice: Christophe Odin (Kelkoo), Hany Saleeb (data mining), and Ron Yorita (data joiner).

This book draws on our teaching at New York University and the University of Copenhagen and on our tuning experience in the industry: Bell Laboratories, NCR, Telcordia, Union Bank of Switzerland, Morgan Stanley Dean Witter, Millenium Pharmaceuticals, Interactive Imaginations, and others. We would like to express our thanks to Martin Appel, Steve Apter, Evan Bauer, Latha Colby, George Collier, Marc Donner, Erin Zoe Ferguson, Anders Fogtmann, Narain Gehani, Paulo Guerra, Christoffer Hall-Fredericksen (who helped implement the tool we used to conduct the experiments), Rachel Hauser, Henry Huang, Martin Jensen, Kenneth Keller, Michael Lee, Martin Leopold, Iona Lerner, Francois Llirbat, Vivian Lok, Rajesh Mani, Bill McKenna, Omar Mehmood, Wayne Miraglia, Ioana Manolescu, Linda Ness, Christophe Odin, Jesper Olsen, Fabio Porto, Tim Shetler, Eric Simon, Judy Smith, Gary Sorrentino, Patrick Valduriez, Eduardo Wagner, Arthur Whitney, and Yan Yuan.

We would like to thank our colleagues at Oracle—in particular Ken Jacobs, Vineet Buch, and Richard Sarwal—for their advice, both legal and technical.

The U.S. National Science Foundation, Advanced Research Projects Agency, NASA, and other funding agencies in the United States and in other countries deserve thanks—not for directly supporting the creation of this book, but for the support that they provide many researchers in this field. Many commercial technologies (hierarchical databases and bitmaps) and even more ideas have grown out of sponsored research prototypes.

The people at Morgan Kaufmann have been a pleasure to work with. The irrepressible senior editor Diane Cerra has successfully weathered a false start and has shown herself to be supportive in so many unpredictable ways. She has chosen a worthy successor in Lothlarien Homet. Our development and production editors Belinda Breyer and Cheri Palmer have been punctual, competent, and gently insistent. Other members of the team include Robert Fiske as copyeditor, Jennifer McClain as proofreader, Yvo Riezebos as lyrical cover designer, and Ty Koontz as indexer. Our thanks to them all. Part of their skill consists in finding excellent reviewers. We would very much like to thank ours: Krishnamurthy Vidyasankar, Gottfried

Vossen, and Karen Watterson. Their advice greatly improved the quality of our book. All remaining faults are our own doing.

Chapter 1: Basic Principles

1.1 The Power of Principles

Tuning rests on a foundation of informed common sense. This makes it both easy and hard.

Tuning is easy because the tuner need not struggle through complicated formulas or theorems. Many academic and industrial researchers have tried to put tuning and query processing generally on a mathematical basis. The more complicated of these efforts have generally foundered because they rest on unrealizable assumptions. The simpler of these efforts offer useful qualitative and simple quantitative insights that we will exploit in the coming chapters.

Tuning is difficult because the principles and knowledge underlying that common sense require a broad and deep understanding of the application, the database software, the operating system, and the physics of the hardware. Most tuning books offer practical rules of thumb but don't mention their limitations.

For example, a book may tell you *never* to use aggregate functions (such as AVG) when transaction response time is critical. The underlying reason is that such functions must scan substantial amounts of data and therefore may block other queries. So the rule is generally true, but it may not hold if the average applies to a few tuples that have been selected by an index. The point of the example is that the tuner must understand the reason for the rule, namely, long transactions that access large portions of shared data may delay concurrent online transactions. The well-informed tuner will then take this rule for what it is: an example of a principle (don't scan large amounts of data in a highly concurrent environment) rather than a principle itself.

This book expresses the philosophy that you can best apply a rule of thumb if you understand the principles. In fact, understanding the principles will lead you to invent new rules of thumb for the unique problems that your application presents. Let us start from the most basic principles— the ones from which all others derive.

1.2 Five Basic Principles

Five principles pervade performance considerations.

1. Think globally; fix locally.
2. Partitioning breaks bottlenecks.
3. Start-up costs are high; running costs are low.
4. Render unto server what is due unto server.
5. Be prepared for trade-offs.

We describe each principle and give examples of its application. Some of the examples will use terms that are defined in later chapters and in the glossary.

1.2.1 Think Globally; Fix Locally

Effective tuning requires a proper identification of the problem and a minimalist intervention. This entails measuring the right quantities and coming to the right conclusions. Doing this well

is challenging, as any medical professional can attest. We present here two examples that illustrate common pitfalls.

- A common approach to global tuning is to look first at hardware statistics to determine processor utilization, input-output (I/O) activity, paging, and so on. The naive tuner might react to a high value in one of these measurements (e.g., high disk activity) by buying hardware to lower it (e.g., buy more disks). There are many cases, however, in which that would be inappropriate. For example, there may be high disk activity because some frequent query scans a table instead of using an index or because the log shares a disk with some frequently accessed data. Creating an index or moving data files among the different disks may be cheaper and more effective than buying extra hardware.
- In one real case that we know of, there was high disk activity because the database administrator had failed to increase the size of the database buffer, thus forcing many unnecessary disk accesses.
- Tuners frequently measure the time taken by a particular query. If this time is high, many tuners will try to reduce it. Such effort, however, will not pay off if the query is executed only seldom. For example, speeding up a query that takes up 1% of the running time by a factor of two will speed up the system by at most 0.5%. That said, if the query is critical somehow, then it may still be worthwhile. Thus, localizing the problem to one query and fixing that one should be the first thing to try. But make sure it is the important query.

When fixing a problem, you must think globally as well. Developers may ask you to take their query and "find the indexes that will make it go fast." Often you will do far better by understanding the application goals because that may lead to a far simpler solution. In our experience, this means sitting with the designer and talking the whole thing through, a little like a therapist trying to solve the problem behind the problem.

1.2.2 Partitioning Breaks Bottlenecks

A slow system is rarely slow because all its components are saturated. Usually, one part of the system limits its overall performance. That part is called a *bottleneck*.

A good way to think about bottlenecks is to picture a highway traffic jam. The traffic jam usually results from the fact that a large portion of the cars on the road must pass through a narrow passageway. Another possible reason is that the stream from one highway merges with the stream from another. In either case, the bottleneck is that portion of the road network having the greatest proportion of cars per lane. Clearing the bottleneck entails locating it and then adopting one of two strategies:

1. Make the drivers drive faster through the section of the highway containing fewer lanes.
2. Create more lanes to reduce the load per lane or encourage drivers to avoid rush hour.

The first strategy corresponds to a local fix (e.g., the decision to add an index or to rewrite a query to make better use of existing indexes) and should be the first one you try. The second strategy corresponds to partitioning.

Partitioning in database systems is a technique for reducing the load on a certain component of the system either by dividing the load over more resources or by spreading the load over time. Partitioning can break bottlenecks in many situations. Here are a few examples. The technical details will become clear later.

- A bank has *N* branches. Most clients access their account data from their home branch. If a centralized system becomes overloaded, a natural solution is to put the account data of clients with home branch *i* into subsystem *i*. This is a form of partitioning in space (of physical resources).
- Lock contention problems usually involve very few resources. Often the free list (the list of unused database buffer pages) suffers contention before the data files. A solution is to divide such resources into pieces in order to reduce the concurrent contention on each lock. In the case of the free list, this would mean creating several free lists, each containing pointers to a portion of the free pages. A thread in need of a free page would lock and access a free list at random. This is a form of logical partitioning (of lockable resources).
- A system with a few long transactions that access the same data as many short ("online") transactions will perform badly because of lock contention and resource contention. Deadlock may force the long transactions to abort, and the long transactions may block the shorter ones. Further, the long transactions may use up large portions of the buffer pool, thereby slowing down the short transactions, even in the absence of lock contention. One possible solution is to perform the long transactions when there is little online transaction activity and to serialize those long transactions (if they are loads) so they don't interfere with one another (partitioning in time). A second is to allow the long transactions (if they are read-only) to apply to out-of-date data (partitioning in space) on separate hardware.

Mathematically, partitioning means dividing a set into mutually disjoint (nonintersecting) parts. These three examples (and the many others that will follow) illustrate partitioning either in space, in logical resource, or in time. Unfortunately, partitioning does not always improve performance. For example, partitioning the data by branches may entail additional communication expense for transactions that cross branches.
So, partitioning—like most of tuning—must be done with care. Still, the main lesson of this section is simple: *when you find a bottleneck, first try to speed up that component; if that doesn't work, then partition.*

1.2.3 Start-Up Costs Are High; Running Costs Are Low

Most man-made objects devote a substantial portion of their resources to starting up. This is true of cars (the ignition system, emergency brakes), of certain kinds of light bulbs (whose lifetimes depend principally on the number of times they are turned on), and of database systems.

- It is expensive to begin a read operation on a disk, but once the read begins, the disk can deliver data at high speed. Thus, reading a 64-kilobyte segment from a single disk track will probably be less than twice as expensive as reading 512 bytes from that track. This suggests that frequently scanned tables should be laid out consecutively on disk. This also suggests that vertical partitioning may be a good strategy when important queries select few columns from tables containing hundreds of columns.
- In a distributed system, the latency of sending a message across a network is very high compared with the incremental cost of sending more bytes in a single message. The net result is that sending a 1-kilobyte packet will be little more expensive than sending a 1-byte packet. This implies that it is good to send large chunks of data rather than small ones.
- The cost of parsing, performing semantic analysis, and selecting access paths for even simple queries is significant (more than 10,000 instructions on most systems). This suggests that often executed queries should be compiled.

- Suppose that a program in a standard programming language such as C++, Java, Perl, COBOL, or PL/1 makes calls to a database system. In some systems (e.g., most relational ones), opening a connection and making a call incurs a significant expense. So, it is much better for the program to execute a single SELECT call and then to loop on the result than to make many calls to the database (each with its own SELECT) within a loop of the standard programming language. Alternatively, it is helpful to cache connections.

These four examples illustrate different senses of start-up costs: obtaining the first byte of a read, sending the first byte of a message, preparing a query for execution, and economizing calls. Yet in every case, the lesson is the same: *obtain the effect you want with the fewest possible start-ups.*

1.2.4 Render unto Server What Is Due unto Server

Obtaining the best performance from a data-intensive system entails more than merely tuning the database management portion of that system. An important design question is the allocation of work between the database system (the server) and the application program (the client). Where a particular task should be allocated depends on three main factors.

1. *The relative computing resources of client and server*: If the server is overloaded, then all else being equal, tasks should be off-loaded to the clients. For example, some applications will off-load compute-intensive jobs to client sites.
2. *Where the relevant information is located*: Suppose some response should occur (e.g., writing to a screen) when some change to the database occurs (e.g., insertions to some database table). Then a well-designed system should use a trigger facility within the database system rather than poll from the application. A polling solution periodically queries the table to see whether it has changed. A trigger by contrast fires only when the change actually takes place, entailing much less overhead.
3. *Whether the database task interacts with the screen*: If it does, then the part that accesses the screen should be done outside a transaction. The reason is that the screen interaction may take a long time (several seconds at least). If a transaction *T* includes the interval, then *T* would prevent other transactions from accessing the data that *T* holds. So, the transaction should be split into three steps:
    a. A short transaction retrieves the data.
    b. An interactive session occurs at the client side outside a transactional context (no locks held).
    c. A second short transaction installs the changes achieved during the interactive session.

We will return to similar examples in the next chapter and in Appendix B when we discuss ways to chop up transactions without sacrificing isolation properties.

1.2.5 Be Prepared for Trade-Offs

Increasing the speed of an application often requires some combination of memory, disk, or computational resources. Such expenditures were implicit in our discussion so far. Now, let us make them explicit.

- Adding random access memory allows a system to increase its buffer size. This reduces the number of disk accesses and therefore increases the system's speed. Of course, random access memory is not (yet) free. (Nor is it random: accessing memory sequentially is faster than accessing widely dispersed chunks.)

- Adding an index often makes a critical query run faster, but entails more disk storage and more space in random access memory. It also requires more processor time and more disk accesses for insertions and updates that do not use the index.
- When you use temporal partitioning to separate long queries from online updates, you may discover that too little time is allocated for those long queries. In that case, you may decide to build a separate archival database to which you issue only long queries. This is a promising solution from the performance standpoint, but may entail the purchase and maintenance of a new computer system.

A consultant for one relational vendor puts it crassly: "You want speed. How much are you willing to pay?"

1.3 Basic Principles and Knowledge

Database tuning is a knowledge-intensive discipline. The tuner must make judgments based on complex interactions among buffer pool sizes, data structures, lock contention, and application needs. That is why the coming chapters mix detailed discussions of single topics with scenarios that require you to consider many tuning factors as they apply to a given application environment. Here is a brief description of each chapter.

- Chapter 2 discusses lower-level system facilities that underlie all database systems.
    - Principles of concurrency control and recovery that are important to the performance of a database system and tuning guidelines for these subcomponents.
    - Aspects of operating system configuration that are important to tuning and monitoring.
    - Hardware modifications that are most likely to improve the performance of your system.
- Chapter 3 discusses the selection of indexes.
    - Query types that are most relevant to index selection.
    - The data structures that most database systems offer (B-trees, hash, and heap structures) and some useful data structures that applications may decide to implement on their own.
    - Sparse versus dense indexes.
    - Clustering (primary) versus nonclustering (secondary) indexes.
    - Multicolumn (composite) indexes.
    - Distribution considerations.
    - Maintenance tips.
- Chapter 4 studies those factors of importance to relational systems.
    - Design of table schema and the costs and benefits of normalization, vertical partitioning, and aggregate materialization.
    - Clustering tables and the relative benefits of clustering versus denormalization.
    - Record layout and the choice of data types.
    - Query reformulation, including methods for minimizing the use of DISTINCT and for eliminating expensive nested queries.
    - Stored procedures.
    - Triggers.
- Chapter 5 considers the tuning problem for the interface between the database server and the outside world.

    - Tuning the application interface.
    - Bulk loading data.
    - Tuning the access to multiple database systems.
- Chapter 6 presents case studies mainly drawn from Shasha's work on Wall Street.
- Chapter 7, written by veteran DBA Alberto Lerner, describes strategies and tools for performance monitoring.
- Chapter 8 presents tuning techniques and capacity planning for e-commerce applications.
- Chapter 9 discusses the uses and abuses of data warehouses from Joe Celko's experience.
- Chapter 10 presents the technology underlying tuning techniques for data warehouses.
- The appendices discuss specialized performance hints for real-time, financial, and online transaction processing systems.
    - Description of considerations concerning priority, buffer allocation, and related matters for real-time database applications.
    - Systematic technique to improve performance by chopping up transactions without sacrificing isolation properties.
    - Usage and support for time series, especially in finance.
    - Hints on reading query plans.
    - Examples of configuration parameters.
- A glossary and an index will attempt to guide you through the fog of tuning terminology.

Chapter 2: Tuning the Guts

2.1 Goal of Chapter

This chapter discusses tuning considerations having to do with the underlying components common to most database management systems. Each component carries its own tuning considerations.

- Concurrency control—how to minimize lock contention.
- Recovery and logging—how to minimize logging and dumping overhead.
- Operating system—how to optimize buffer size, process scheduling, and so on.
- Hardware—how to allocate disks, random access memory, and processors.

Figure 2.1 shows the common underlying components of all database systems.

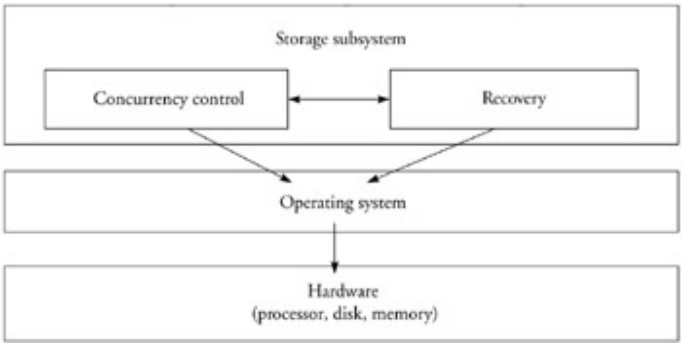


Figure 2.1: Underlying components of a database system.

WARNING ABOUT THIS CHAPTER This is the most difficult chapter of the book because we have written it under the assumption that you have only a passing understanding of concurrent systems. We want to lead you to a level where you can make subtle trade-offs between speed and concurrent correctness, between speed and fault tolerance, and between speed and hardware costs.

So, it may be tough going in spots. If you have trouble, take a break and read another chapter.

2.2 Locking and Concurrency Control

Database applications divide their work into *transactions*. When a transaction executes, it accesses the database and performs some local computation. The strongest assumption that an application programmer can make is that each transaction will appear to execute in isolation—without concurrent activity. Because this notion of isolation suggests indivisibility, transactional guarantees are sometimes called *atomicity guarantees*.[1] Database researchers, you see, didn't bother to let 20th-century physics interfere with their jargon.

The sequence of transactions within an application program, taken as a whole, enjoys no such guarantee, however. Between, say, the first two transactions executed by an application program, other application programs may have executed transactions that modified data items accessed by one or both of these first two transactions. For this reason, the length of a transaction can have important correctness implications.

EXAMPLE: THE LENGTH OF A TRANSACTION

Suppose that an application program processes a purchase by adding the value of the item to inventory and subtracting the money paid from cash. The application specification requires that cash never be made negative, so the transaction will roll back (undo its effects) if subtracting the money from cash will cause the cash balance to become negative.

To reduce the time locks are held, the application designers divide these two steps into two transactions.

1. The first transaction checks to see whether there is enough cash to pay for the item. If so, the first transaction adds the value of the item to inventory. Otherwise, abort the purchase application.
2. The second transaction subtracts the value of the item from cash.

They find that the cash field occasionally becomes negative. Can you see what might have happened?

Consider the following scenario. There is $100 in cash available when the first application program begins to execute. An item to be purchased costs $75. So, the first transaction commits. Then some other execution of this application program causes $50 to be removed from cash. When the first execution of the program commits its second transaction, cash will be in deficit by $25.

So, dividing the application into two transactions can result in an inconsistent database state. Once you see it, the problem is obvious though no amount of sequential testing would have revealed it. Most concurrent testing would not have revealed it either because the problem occurs rarely.

So, cutting up transactions may jeopardize correctness even while it improves performance. This tension between performance and correctness extends throughout the space of tuning options for concurrency control. You can make the following choices:

- The number of locks each transaction obtains (fewer tends to be better for performance, all else being equal).
- The kinds of locks those are (read locks are better for performance).
- The length of time that the transaction holds them (shorter is better for performance).

Because a rational setting of these tuning options depends on an understanding of correctness goals, the next section describes those goals and the general strategies employed by database systems to achieve them.

Having done that, we will be in a position to suggest ways of rationally trading performance for correctness.

2.2.1 Correctness Considerations

Two transactions are said to be *concurrent* if their executions overlap in time. That is, there is some point in time in which both transactions have begun, and neither has completed. Notice that two transactions can be concurrent even on a uniprocessor. For example, transaction *T* may begin, then may suspend after issuing its first disk access instruction, and then transaction

T' T' may begin. In such a case, T and T' would be concurrent. Figure 2.2 shows an example with four transactions.

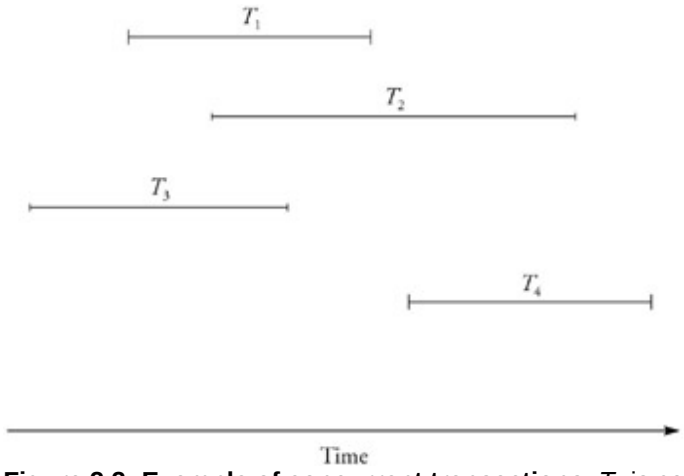


Figure 2.2: Example of concurrent transactions. T_1 is concurrent with T_2 and T_3. T_2 is concurrent with T_1, T_3, and T_4.

Concurrency control is, as the name suggests, the activity of controlling the interaction among concurrent transactions. Concurrency control has a simple correctness goal: make it *appear* as if each transaction executes in isolation from all others. That is, the execution of the transaction collection must be *equivalent* to one in which the transactions execute one at a time. Two executions are equivalent if one is a rearrangement of the other; and in the rearrangement, each read returns the same value, and each write stores the same value as in the original.

Notice that this correctness criterion says nothing about what transactions do. Concurrency control algorithms leave that up to the application programmer. That is, the application programmer must guarantee that the database will behave appropriately provided each transaction appears to execute in isolation. What is appropriate depends entirely on the application, so is outside the province of the concurrency control subsystem.

Concurrent correctness is usually achieved through mutual exclusion. Operating systems allow processes to use semaphores for this purpose. Using a semaphore S, a thread (or process) can access a resource R with the assurance that no other thread (or process) will access R concurrently.

A naive concurrency control mechanism for a database management system would be to have a single semaphore S. Every transaction would acquire S before accessing the database. Because only one transaction can hold S at any one time, only one transaction can access the database at that time. Such a mechanism would be correct but would perform badly at least for applications that access slower devices such as disks.

EXAMPLE: SEMAPHORE METHOD

Suppose that Bob and Alice stepped up to different automatic teller machines (ATMs) serving the same bank at about the same time. Suppose further that they both wanted to make deposits but into different accounts. Using the semaphore solution, Alice might hold the semaphore during her transaction during the several seconds that the database is updated and an envelope is fed into the ATM. This would prevent Bob (and any other person) from performing a bank transaction during those seconds. Modern banking would be completely infeasible.

Surprisingly, however, careful design could make the semaphore solution feasible. It all depends on when you consider the transaction to begin and end and on how big the database is relative to high-speed memory. Suppose the transaction begins after the user has made all decisions and ends once those decisions are recorded in the database, but before physical cash has moved. Suppose further that the database fits in memory so the database update takes well under a millisecond. In such a case, the semaphore solution works nicely and is used in some main memory databases.

An overreaction to the Bob and Alice example would be to do away with the semaphore and to declare that no concurrency control at all is necessary. That can produce serious problems, however.

EXAMPLE: NO CONCURRENCY CONTROL

Imagine that Bob and Alice have decided to share an account. Suppose that Bob goes to a branch on the east end of town to deposit $100 in cash and Alice goes to a branch on the west end of town to deposit $500 in cash. Once again, they reach the automatic teller machines at about the same time. Before they begin, their account has a balance of 0. Here is the progression of events in time.

1. Bob selects the deposit option.
2. Alice selects the deposit option.
3. Alice puts the envelope with her money into the machine at her branch.
4. Bob does the same at his branch.
5. Alice's transaction begins and reads the current balance of $0.
6. Bob's transaction begins and reads the current balance of $0.
7. Alice's transaction writes a new balance of $500, then ends.
8. Bob's transaction writes a new balance of $100, then ends.

Naturally, Bob and Alice would be dismayed at this result. They expected to have $600 in their bank balance but have only $100 because Bob's transaction read the same original balance as Alice's. The bank might find some excuse ("excess static electricity" is one favorite), but the problem would be due to a lack of concurrency control.

So, semaphores on the entire database can give ruinous performance, and a complete lack of control gives manifestly incorrect results. Locking is a good compromise. There are two kinds of locks: *write* (also known as exclusive) locks and *read* (also known as shared) locks. Write locks are like semaphores in that they give the right of exclusive access, except that they apply to only a portion of a database, for example, a page. Any such lockable portion is variously referred to as a *data item*, an *item*, or a *granule*.

Read locks allow shared access. Thus, many transactions may hold a read lock on a data item *x* at the same time, but only one transaction may hold a write lock on *x* at any given time. Usually, database systems acquire and release locks implicitly using an algorithm known as *Two-Phase Locking*, invented at IBM Almaden research by K. P. Eswaran, J. Gray, R. Lorie, and I. Traiger in 1976. That algorithm follows two rules.

1. A transaction must hold a lock on *x* before accessing *x*. (That lock should be a write lock if the transaction writes *x* and a read lock otherwise.)
2. A transaction must not acquire a lock on any item *y* after releasing a lock on any item *x*. (In practice, locks are released when a transaction ends.)

The second rule may seem strange. After all, why should releasing a lock on, say, Ted's account prevent a transaction from later obtaining a lock on Carol's account? Consider the following example.

EXAMPLE: THE SECOND RULE AND THE PERILS OF RELEASING SHARED LOCKS

Suppose Figure 2.3 shows the original database state. Suppose further that there are two transactions. One transfers $1000 from Ted to Carol, and the other computes the sum of all deposits in the bank. Here is what happens.

1. The sum-of-deposits transaction obtains a read lock on Ted's account, reads the balance of $4000, then releases the lock.
2. The transfer transaction obtains a lock on Ted's account, subtracts $1000, then obtains a lock on Carol's account and writes, establishing a balance of $1000 in her account.
3. The sum-of-deposits transaction obtains a read lock on Bob and Alice's account, reads the balance of $100, then releases the lock. Then that transaction obtains a read lock on Carol's account and reads the balance of $1000 (resulting from the transfer).

ACCOUNT OWNER	BALANCE
Ted	4000
Bob and Alice	100
Carol	0

Figure 2.3: Original database state.

So, the sum-of-deposits transaction overestimates the amount of money that is deposited.

The previous example teaches two lessons.

- The two-phase condition should apply to reads as well as writes.
- Many systems (e.g., SQL Server, Sybase, etc.) give a default behavior in which write locks are held until the transaction completes, but read locks are released as soon as the data item is read (degree 2 isolation). This example shows that this can lead to faulty behavior. *This means that if you want concurrent correctness, you must sometimes request nondefault behavior.*

2.2.2 Lock Tuning

Lock tuning should proceed along several fronts.

1. Use special system facilities for long reads.
2. Eliminate locking when it is unnecessary.
3. Take advantage of transactional context to chop transactions into small pieces.
4. Weaken isolation guarantees when the application allows it.
5. Select the appropriate granularity of locking.

6. Change your data description data during quiet periods only. (Data Definition Language statements are considered harmful.)
7. Think about partitioning.
8. Circumvent hot spots.
9. Tune the deadlock interval.

You can apply each tuning suggestion independently of the others, but you must check that the appropriate isolation guarantees hold when you are done. The first three suggestions require special care because they may jeopardize isolation guarantees if applied incorrectly.

For example, the first one offers full isolation (obviously) provided a transaction executes alone. It will give no isolation guarantee if there can be arbitrary concurrent transactions.

Use facilities for long reads

Some relational systems, such as Oracle, provide a facility whereby read-only queries hold no locks yet appear to execute serializably. The method they use is to re-create an old version of any data item that is changed after the read query begins. This gives the effect that a read-only transaction *R* reads the database *as the database appeared when R began* as illustrated in Figure 2.4.

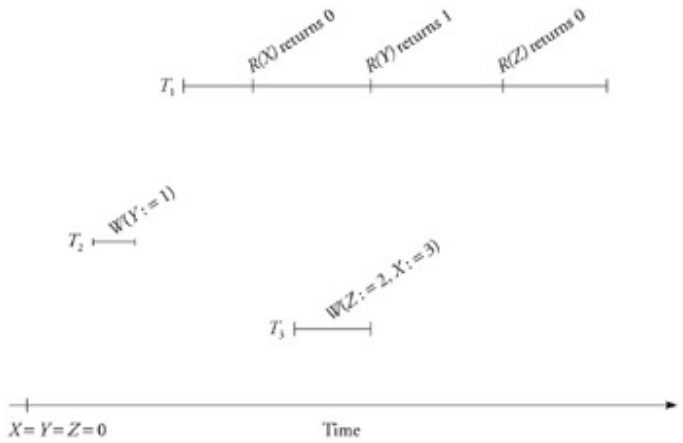


Figure 2.4: Multiversion read consistency. In this example, three transactions T_1, T_2, and T_3 access three data items *X*, *Y*, and *Z*. T_1 reads the values of *X*, *Y*, and *Z* (T_1:*R*(*X*), *R*(*Y*), *R*(*Z*)). T_2 sets the value of *Y* to 1 (T_2:*W*(*Y*: = 1)). T_3 sets the value of *Z* to 2 and the value of *X* to 3 (T_3:*W*(*Z* : = 2), *W*(*X*: = 3)). Initially *X*, *Y*, and *Z* are equal to 0. The diagram illustrates the fact that, using multiversion read consistency, T_1 returns the values that were set when it started.

Using this facility has the following implications:

- Read-only queries suffer no locking overhead.
- Read-only queries can execute in parallel and on the same data as short update transactions without causing blocking or deadlocks.
- There is some time and space overhead because the system must write and hold old versions of data that have been modified. The only data that will be saved, however, is that which is updated while the read-only query runs. Oracle snapshot isolation avoids this overhead by leveraging the before-image of modified data kept in the rollback segments.

Although this facility is very useful, please keep two caveats in mind.

1. When extended to read/write transactions, this method (then called snapshot isolation) does *not* guarantee correctness. To understand why, consider a pair of transactions

- $T_1 : x := y$
- $T_2 : y := x$

Suppose that x has the initial value 3 and y has the initial value 17. In any serial execution in which T_1 precedes T_2, both x and y have the value 17 at the end. In any serial execution in which T_2 precedes T_1, both x and y have the value 3 at the end. But if we use snapshot isolation and T_1 and T_2 begin at the same time, then T_1 will read the value 17 from y without obtaining a read lock on y. Similarly, T_2 will read the value 3 from x without obtaining a read lock on x. T_1 will then write 17 into x. T_2 will write 3 into y. No serial execution would do this.

What this example (and many similar examples) reveals is that database systems will, in the name of performance, allow bad things to happen. It is up to application programmers and database administrators to choose these performance optimizations with these risks in mind. A default recommendation is to use snapshot isolation for read-only transactions, but to ensure that read operations hold locks for transactions that perform updates.

2. In some cases, the space for the saved data may be too small. You may then face an unpleasant surprise in the form of a return code such as "snapshot too old." When automatic undo space management is activated, Oracle 9i allows you to configure how long before-images are kept for consistent read purpose. If this parameter is not set properly, there is a risk of "snapshot too old" failure.

Eliminate unnecessary locking

Locking is unnecessary in two cases.

1. When only one transaction runs at a time, for example, when loading the database
2. When all transactions are read-only, for example, when doing decision support queries on archival databases

In these cases, users should take advantage of options to reduce overhead by suppressing the acquisition of locks. (The overhead of lock acquisition consists of memory consumption for lock control blocks and processor time to process lock requests.) This may not provide an enormous performance gain, but the gain it does provide should be exploited.

Make your transaction short

The correctness guarantee that the concurrency control subsystem offers is given in units of transactions. At the highest degree of isolation, each transaction is guaranteed to appear as if it executed without being disturbed by concurrent transactions.

An important question is how long should a transaction be? This is important because transaction length has two effects on performance.

- The more locks a transaction requests, the more likely it is that it will have to wait for some other transaction to release a lock.

- The longer a transaction *T* executes, the more time another transaction will have to wait if it is blocked by *T*.

Thus, in situations in which blocking can occur (i.e., when there are concurrent transactions some of which update data), short transactions are better than long ones. Short transactions are generally better for logging reasons as well, as we will explain in the recovery section. Sometimes, when you can characterize all the transactions that will occur during some time interval, you can "chop" transactions into shorter ones without losing isolation guarantees. Appendix B presents a systematic approach to achieve this goal. Here are some motivating examples along with some intuitive conclusions.

EXAMPLE: UPDATE BLOB WITH CREDIT CHECKS

A certain bank allows depositors to take out up to $1000 a day in cash. At night, one transaction (the "update blob" transaction) updates the balance of every accessed account and then the appropriate branch balance. Further, throughout the night, there are occasional credit checks (read-only transactions) on individual depositors that touch only the depositor account (not the branch). The credit checks arrive at random times, so are not required to reflect the day's updates. The credit checks have extremely variable response times, depending on the progress of the updating transaction. What should the application designers do?

Solution 1: Divide the update blob transaction into many small update transactions, each of which updates one depositor's account and the appropriate branch balance. These can execute in parallel. The effect on the accounts will be the same because there will be no other updates at night. The credit check will still reflect the value of the account either before the day's update or after it.

Solution 2: You may have observed that the update transactions may now interfere with one another when they access the branch balances. If there are no other transactions, then those update transactions can be further subdivided into an update depositor account transaction and an update branch balance transaction.

Intuitively, each credit check acts independently in the previous example. Therefore, breaking up the update transaction causes no problem. Moreover, no transaction depends on the consistency between the account balance value and the branch balance, permitting the further subdivision cited in solution 2. Imagine now a variation of this example.

EXAMPLE: UPDATES AND BALANCES

Instead of including all updates to all accounts in one transaction, the application designers break them up into minitransactions, each of which updates a single depositor's account and the branch balance of the depositor's branch. The designers add an additional possible concurrent transaction that sums the account balances and the branch balances to see if they are equal. What should be done in this case?

Solution: The balance transaction can be broken up into several transactions. Each one would read the accounts in a single branch and the corresponding branch balance. The updates to the account and to the branch balance may no longer be subdivided into two transactions, however. The reason is that the balance transaction may see the result of an update to a depositor account but not see the compensating update to the branch balance.

These examples teach a simple lesson: *whether or not a transaction T may be broken up into smaller transactions depends on what is concurrent with T.*

Informally, there are two questions to ask.

1. Will the transactions that are concurrent with *T* cause *T* to produce an inconsistent state or to observe an inconsistent value if *T* is broken up?
2. That's what happened when the purchase transaction was broken up into two transactions in the example entitled "The Length of a Transaction" on page 11. Notice, however, that the purchase transaction in that example could have been broken up if it had been reorganized slightly. Suppose the purchase transaction first subtracted money from cash (rolling back if the subtraction made the balance negative) and then added the value of the item to inventory. Those two steps could become two transactions given the concurrent activity present in that example. The subtraction step can roll back the entire transaction before any other changes have been made to the database. Thus, rearranging a program to place the update that causes a rollback first may make a chopping possible.
3. Will the transactions that are concurrent with *T* be made inconsistent if *T* is broken up?

That's what would happen if the balance transaction ran concurrently with the finest granularity of update transactions, that is, where each depositor-branch update transaction was divided into two transactions, one for the depositor and one for the branch balance.

Here is a rule of thumb that often helps when chopping transactions (it is a special case of the method to be presented in Appendix B):

Suppose transaction T accesses data X and Y, but any other transaction T´accesses at most one of X or Y and nothing else. Then T can be divided into two transactions, one of which accesses X and the other of which accesses Y.

This rule of thumb led us to break up the balance transaction into several transactions, each of which operates on the accounts of a single branch. This was possible because all the small update transactions worked on a depositor account and branch balance of the same branch. CAVEAT Transaction chopping as advocated here and in Appendix B works correctly if properly applied. The important caution to keep in mind is that adding a new transaction to a set of existing transactions may invalidate all previously established choppings.

Weaken isolation guarantees carefully

Quite often, weakened isolation guarantees are sufficient. SQL offers the following options:

1. *Degree 0:* Reads may access *dirty data*, that is, data written by uncommitted transactions. If an uncommitted transaction aborts, then the read may have returned a value that was never installed in the database. Further, different reads by a single transaction to the same data will not be *repeatable*, that is, they may return different values. Writes may overwrite the dirty data of other transactions. A transaction holds a write lock on *x* while writing *x*, then releases the lock immediately thereafter.
2. *Degree 1—read uncommitted:* Reads may read dirty data and will not be repeatable. Writes may not overwrite other transactions' dirty data.
3. *Degree 2—read committed:* Reads may access only committed data, but reads are still not repeatable because an access to a data item *x* at time T_2 may read from a different committed transaction than the earlier access to *x* at T_1. In a classical locking implementation of degree 2 isolation, a transaction acquires and releases write locks according to two-phase locking, but releases the read lock

immediately after reading it. Relational systems offer a slightly stronger guarantee known as *cursor stability* : during the time a transaction holds a cursor (a pointer into an array of rows returned by a query), it holds its read locks. This is normally the time that it takes a single SQL statement to execute.

4. Degree 3—serializable: Reads may access only committed data, and reads are repeatable. Writes may not overwrite other transactions' dirty data. The execution is equivalent to one in which each committed transaction executes in isolation, one at a time. Note that ANSI SQL makes a distinction between repeatable read and serializable isolation levels. Using repeatable read, a transaction T1 can insert or delete tuples in a relation that is scanned by transaction T2, but T2 may see only some of those changes, and as a result the execution is not serializable. The serializable level ensures that transactions appear to execute in isolation and is the only truly safe condition.

Figures 2.5, 2.6, and 2.7 illustrate the risks of a weak isolation level and the value of serializability. The experiment simply consists of a summation query repeatedly executed together with a fixed number of money transfer transactions. Each money transfer transaction subtracts (debits) a fixed amount from an account and adds (credits) the same amount to another account. This way, the sum of all account balances remains constant. The number of threads that execute these update transactions is our parameter. We use row locking throughout the experiment. The serializable isolation level guarantees that the sum of the account balances is computed in isolation from the update transactions. By contrast, the read committed isolation allows the sum of the account balances to be computed after a debit operation has taken place but before the corresponding credit operation is performed. The bottom line is that applications that use any isolation level less strong than serializability can return incorrect answers.

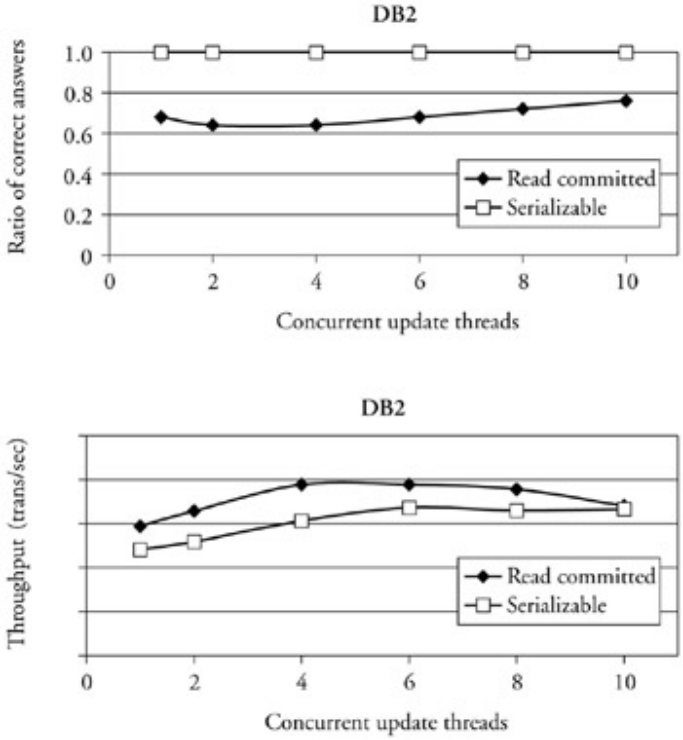


Figure 2.5: Value of serializability (DB2 UDB). A summation query is run concurrently with swapping transactions (a read followed by a write in each transaction). The read committed isolation level does not guarantee that the summation query returns correct answers. The serializable isolation level guarantees correct answers at the cost of decreased throughput. These graphs were obtained using DB2 UDB V7.1 on Windows 2000.

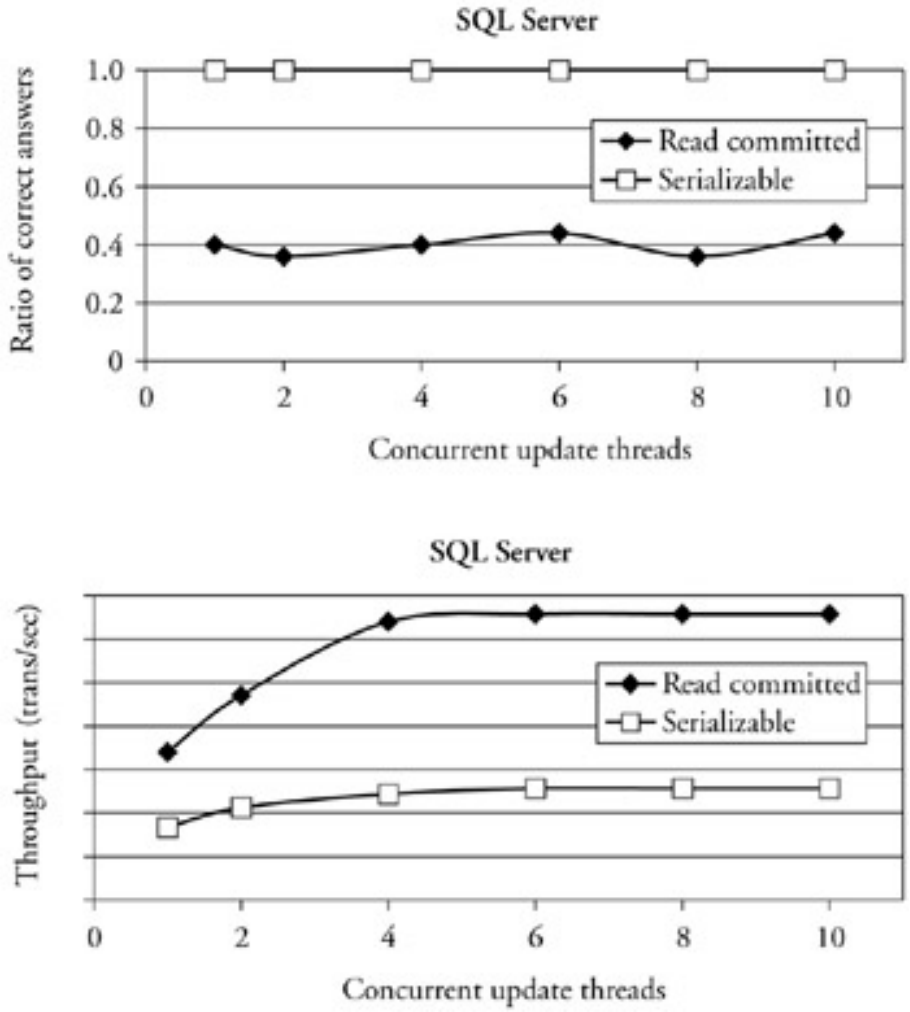


Figure 2.6: Value of serializability (SQL Server). A summation query is run concurrently with swapping transactions (a read followed by a write in each transaction). Using the read committed isolation level, the ratio of correct answers is low. In comparison, the serializable isolation level always returns a correct answer. The high throughput achieved with read committed thus comes at the cost of incorrect answers. These graphs were obtained using SQL Server 7 on Windows 2000.

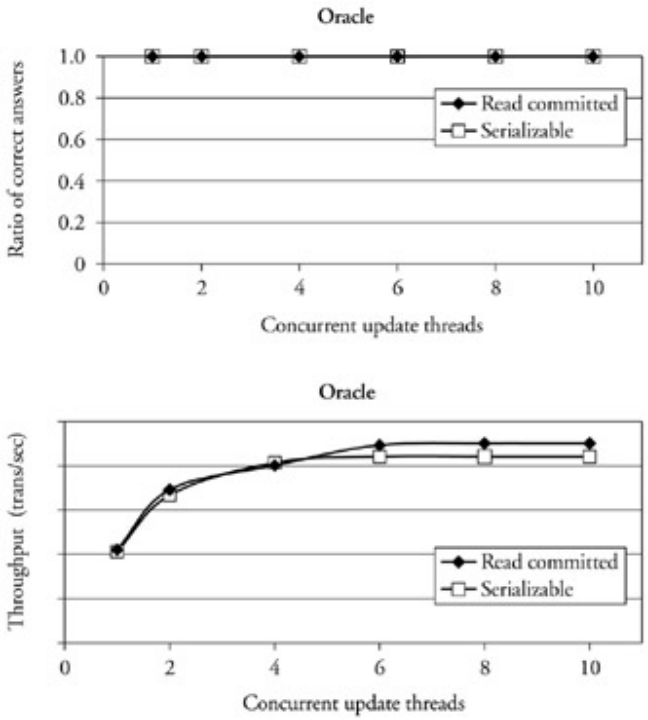


Figure 2.7: Value of serializability (Oracle). A summation query is run concurrently with swapping transactions (a read followed by a write in each transaction). In this case, Oracle's snapshot isolation protocol guarantees that the correct answer to the summation query is returned regardless of the isolation level because each update follows a read on the same data item. Snapshot isolation would have violated correctness had the writes been "blind." Snapshot isolation is further described in the next section on facilities for long reads. These graphs were obtained using Oracle 8i EE on Windows 2000.

Some transactions do not require exact answers and, hence, do not require degree 3 isolation. For example, consider a statistical study transaction that counts the number of depositor account balances that are over $1000. Because an exact answer is not required, such a transaction need not keep read locks. Degree 2 or even degree 1 isolation may be enough. The lesson is this. *Begin with the highest degree of isolation (serializable in relational systems). If a given transaction (usually a long one) either suffers extensive deadlocks or causes significant blocking, consider weakening the degree of isolation, but do so with the awareness that the answers may be off slightly.*

Another situation in which correctness guarantees may be sacrificed occurs when a transaction includes human interaction and the transaction holds hot data.

EXAMPLE: AIRLINE RESERVATIONS

A reservation involves three steps.

1. Retrieve the list of seats available.
2. Determine which seat the customer wants.
3. Secure that seat.

If all this were encapsulated in a single transaction, then that transaction would hold the lock on the list of seats in a given plane while a reservations agent talks to a customer. At busy times, many other customers and agents might be made to wait.

To avoid this intolerable situation, airline reservation systems break up the act of booking a seat into two transactions and a nontransactional interlude. The first transaction reads the list of seats available. Then there is human interaction between the reservation agent and the customer. The second transaction secures a seat that was chosen by a customer. Because of concurrent activity, the customer may be told during step 2 that seat S is available and then be told after step 3 that seat S could not be secured. This happens rarely enough to be considered acceptable. Concurrency control provides the lesser guarantee that no two passengers will secure the same seat.

It turns out that many airlines go even farther, allocating different seats to different cities in order to reduce contention. So, if you want a window seat in your trip from New York to San Francisco and the New York office has none, consider calling the San Francisco office.

Control the granularity of locking

Most modern database management systems offer different "granularities" of locks. The default is normally record-level locking, also called row-level locking. A *page-level lock* will prevent concurrent transactions from accessing (if the page-level lock is a write lock) or modifying (if the page-level lock is a read lock) all records on that page. A *table-level lock* will prevent concurrent transactions from accessing or modifying (depending on the kind of lock) all pages that are part of that table and, by extension, all the records on those pages. Record-level locking is said to be *finer grained* than page-level locking, which, in turn, is finer grained than table-level locking.

If you were to ask the average application programmer on the street whether, say, record-level locking was better than page-level locking, he or she would probably say, "Yes, of course. Record-level locking will permit two different transactions to access different records on the same page. It must be better."

By and large this response is correct for online transaction environments where each transaction accesses only a few records spread on different pages.
Surprisingly, for most modern systems, the locking overhead is low even if many records are locked as the insert transaction in Figure 2.8 shows.

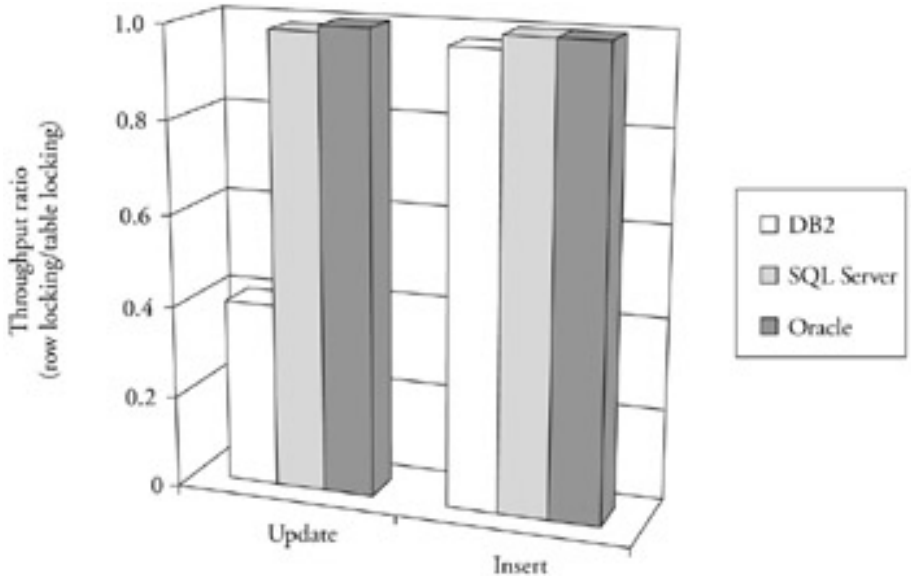


Figure 2.8: Locking overhead. We use two transactions to evaluate how locking overhead affects performance: an update transaction updates 100,000 rows in the accounts table while an insert transaction inserts 100,000 rows in this table. The transaction commits only after all updates or inserts have been performed. The intrinsic performance costs of row locking and table locking are negligible because recovery overhead (the logging of updates) is so much higher than locking overhead. The exception is DB2 on updates because that system does "logical logging" (instead of logging images of changed data, it logs the operation that caused the change). In that case, the recovery overhead is low and the locking overhead is perceptible. This graph was obtained using DB2 UDB V7.1, SQL Server 7, and Oracle 8i EE on Windows 2000.

There are three reasons to ask for table locks. First, table locks can be used to avoid blocking long transactions. Figures 2.9, 2.10, and 2.11 illustrate the interaction of a long transaction (a summation query) with multiple short transactions (debit/credit transfers). Second, they can be used to avoid deadlocks. Finally, they reduce locking overhead in the case that there is no concurrency.

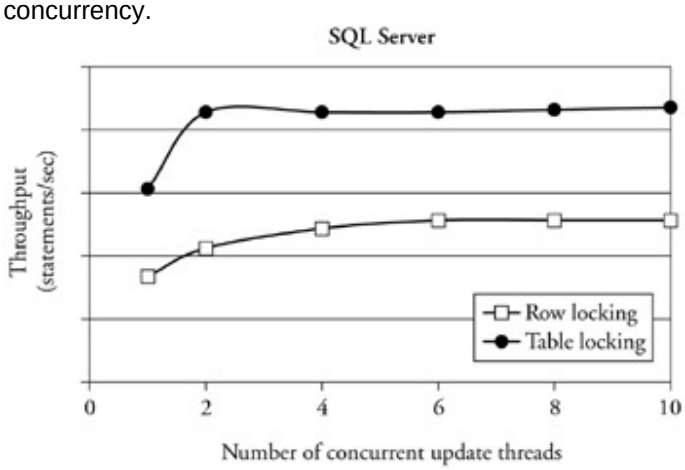


Figure 2.9: Fine-grained locking (SQL Server). A long transaction (a summation query) runs concurrently with multiple short transactions (debit/credit transfers). The serializable isolation level is used to guarantee that the summation query returns correct answers. In order to guarantee a serializable isolation level, row locking forces the use of key range locks (clustered indexes are sparse in SQL Server, thus key range locks involve multiple rows; see Chapter 3 for a description of sparse indexes). In this case, key range locks do not increase concurrency significantly compared to table locks while they force the execution of summation queries to be stopped and resumed. As a result, with this workload table locking performs better. Note that in

SQL Server the granularity of locking is defined by configuring the table; that is, all transactions accessing a table use the same lock granularity. This graph was obtained using SQL Server 7 on Windows 2000.

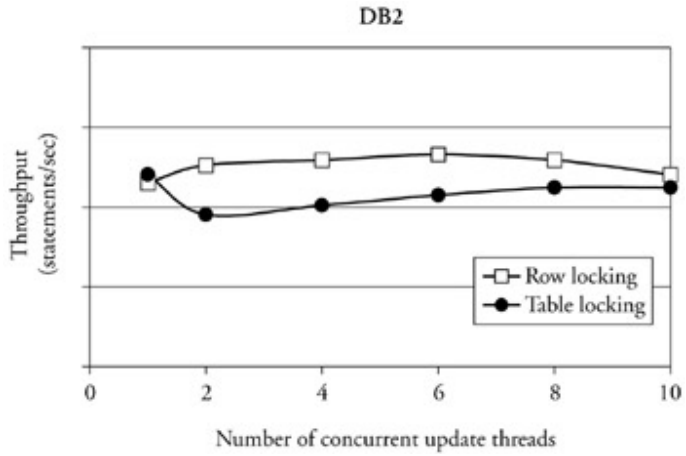


Figure 2.10: Fine-grained locking (DB2). A long transaction (a summation query) with multiple short transactions (debit/credit transfers). Row locking performs slightly better than table locking. Note that by default DB2 automatically selects the granularity of locking depending on the access method selected by the optimizer. For instance, when a table scan is performed (no index is used) in serializable mode, then a table lock is acquired. Here an index scan is performed and row locks are acquired unless table locking is forced using the LOCK TABLE command. This graph was obtained using DB2 UDB V7.1 on Windows 2000.

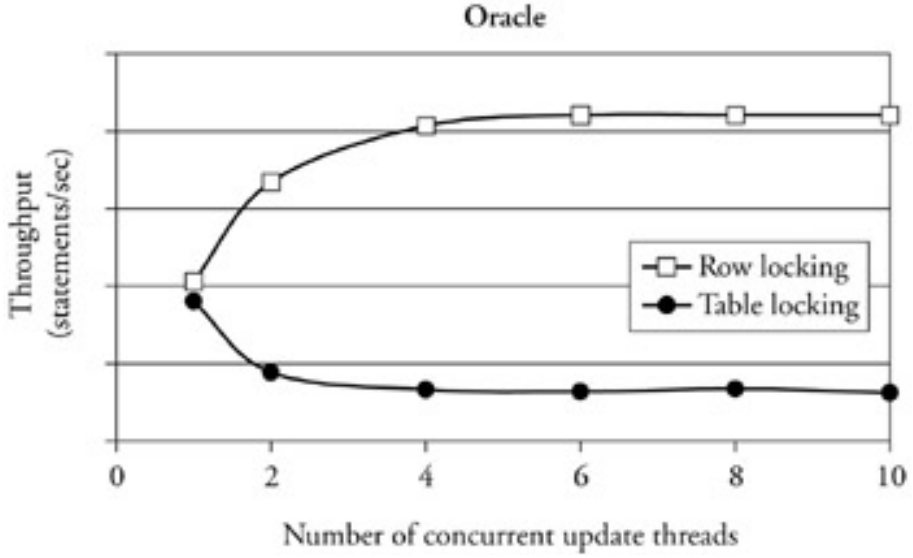


Figure 2.11: Fine-grained locking (Oracle). A long transaction (a summation query) with multiple short transactions (debit/credit transfers). Because snapshot isolation is used the summation query does not conflict with the debit/credit transfers. Table locking forces debit/credit transactions to wait, which is rare in the case of row locking. As a result, the throughput is significantly lower with table locking. This graph was obtained using Oracle 8i EE on Windows 2000.

SQL Server 7 and DB2 UDB V7.1 provide a lock escalation mechanism that automatically upgrades row-level locks into a single table-level lock when the number of row-level locks reaches a predefined threshold. This mechanism can create a deadlock if a long transaction tries to upgrade its row-level locks into a table-level lock while concurrent update or even read transactions are waiting for locks on rows of that table. This is why Oracle does not support

lock escalation. Explicit table-level locking by the user will prevent such deadlocks at the cost of blocking row locking transactions.
The conclusion is simple. *Long transactions should use table locks mostly to avoid deadlocks, and short transactions should use record locks to enhance concurrency.* Transaction length here is relative to the size of the table at hand: a long transaction is one that accesses nearly all the pages of the table.

There are basically three tuning knobs that the user can manipulate to control granule size.

1. *Explicit control of the granularity*

- *Within a transaction:* A statement within a transaction explicitly requests a table-level lock in shared or exclusive mode (Oracle, DB2).
- *Across transactions:* A command defines the lock granularity for a table or a group of tables (SQL Server). All transactions that access these tables use the same lock granularity.

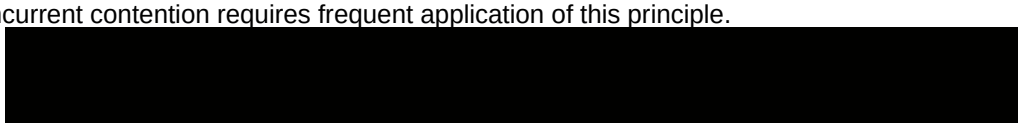
2. *Setting the escalation point:* Systems that support lock escalation acquire the default (finest) granularity lock until the number of acquired locks exceeds some threshold set by the database administrator. At that point, the next coarser granularity lock will be acquired. The general rule of thumb is to set the threshold high enough so that in an online environment of relatively short transactions, escalation will never take place.
3. *Size of the lock table:* If the administrator selects a small lock table size, the system will be forced to escalate the lock granularity even if all transactions are short.

Data definition language (DDL) statements are considered harmful

Data definition data (also known as the system catalog or metadata) is information about table names, column widths, and so on. DDL is the language used to access and manipulate that table data. Catalog data must be accessed by every transaction that performs a compilation, adds or removes a table, adds or removes an index, or changes an attribute description. As a result, the catalog can easily become a hot spot and therefore a bottleneck. A general recommendation therefore is to avoid updates to the system catalog during heavy system activity, especially if you are using dynamic SQL (which must read the catalog when it is parsed).

Think about partitioning

One of the principles from Chapter 1 held that partitioning breaks bottlenecks. Overcoming concurrent contention requires frequent application of this principle.



EXAMPLE: INSERTION TO HISTORY

If all insertions to a data collection go to the last page of the file containing that collection, then the last page may, in some cases, be a concurrency control bottleneck. This is often the case for history files and security files. A good strategy is to partition insertions to the file across different pages and possibly different disks.

This strategy requires some criteria for distributing the inserts. Here are some possibilities.

1. Set up many insertion points and insert into them randomly. This will work provided the file is essentially write-only (like a history file) or whose only readers are scans.

2. Set up a clustering index based on some attribute that is not correlated with the time of insertion. (If the attribute's values are correlated with the time of insertion, then use a hash data structure as the clustering index. If you have only a B-tree available, then hash the time of insertion and use that as the clustering key.) In that way, different inserted records will likely be put into different pages.
You might wonder how much partitioning to specify. A good rule of thumb is to specify at least *n*/4 insertion points, where *n* is the maximum number of concurrent transactions writing to the potential bottleneck.
Figure 2.12 and Figure 2.13 illustrate the impact of the number of insertion points on performance. An insertion point is a position where a tuple may be inserted in a table. In the figures, "sequential" denotes the case where a clustered index is defined on an attribute whose value increases (or even decreases) with time. "Nonsequential" denotes the case where a clustered index is defined on an attribute whose values are independent of time. Hashing denotes the case where a composite clustered index is defined on a key composed of an integer generated in the range of 1 … *k* (*k* should be a prime number) and of the attribute on which the input data is sorted.

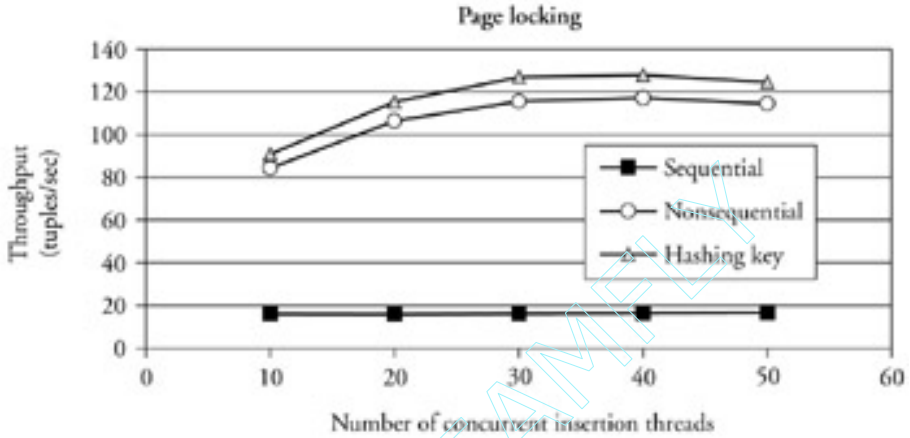


Figure 2.12: Multiple insertion points and page locking. There is contention when data is inserted in a heap or when there is a sequential key and the index is a B-tree: all insertions are performed on the same page. Use multiple insertion points to solve this problem. This graph was obtained using SQL Server 7 on Windows 2000.

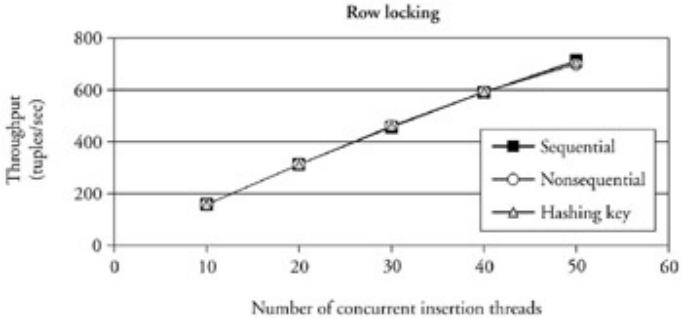


Figure 2.13: Multiple insertion points and row locking. Row locking avoids contention between successive insertions. The number of insertion points thus becomes irrelevant: it is equal to the number of inserted rows. This graph was obtained using SQL Server 7 on Windows 2000.

With page locking, the number of insertion points makes a difference: sequential keys cause all insertions to target the same page; as a result, contention is maximal. Row locking ensures that a new insertion point is assigned for each insert regardless of the method of insertion and hence eliminates the problem.

EXAMPLE: FREE LISTS

Free lists are data structures that govern the allocation and deallocation of real memory pages in buffers. Locks on free lists are held only as long as the allocation or deallocation takes place, but free lists can still become bottlenecks.

For example, in Oracle 8i, the number of free lists can be specified when creating a table (by default one free list is created). The rule of thumb is to specify a number of free lists equal to the maximum number of concurrent threads of control. Each thread running an insertion statement accesses one of these free lists. Oracle recommends to partition the master free list either when multiple user threads seek free blocks or when multiple instances run.[2]

Circumventing hot spots

A *hot spot* is a piece of data that is accessed by many transactions and is updated by some. Hot spots cause bottlenecks because each updating transaction must complete before any other transaction can obtain a lock on the hot data item. (You can usually identify hot spots easily for this reason: if transaction *T* takes far longer than normal at times when other transactions are executing at normal speeds, *T* is probably accessing a hot spot.) There are three techniques for circumventing hot spots.

1. Use partitioning to eliminate it, as discussed earlier.
2. Access the hot spot as late as possible in the transaction.

   Because transactions hold locks until they end, rewriting a transaction to obtain the lock on a hot data item as late as possible will minimize the time that the transaction holds the lock.
3. Use special database management facilities.

   Sometimes, the reason for a hot spot is surprising. For example, in some versions of Sybase Adaptive Server, performing a "select * into #temp…" locks the system catalog of the temporary database while the select takes place. This makes the system catalog a hot spot.

Here is another example. In many applications, transactions that insert data associate a unique identifier with each new data item. When different insert transactions execute concurrently, they must somehow coordinate among themselves to avoid giving the same identifier to different data items.

One way to do this is to associate a counter with the database. Each insert transaction increments the counter, performs its insert and whatever other processing it must perform, then commits. The problem is that the counter becomes a bottleneck because a transaction will (according to two-phase locking) release its lock on the counter only when the transaction commits.

Some systems offer a facility (sequences in Oracle and identity in SQL Server, DB2 UDB, or Sybase Adaptive Server) that enables transactions to hold a latch[3] on the counter. This eliminates the counter as a bottleneck but may introduce a small problem.
Consider an insert transaction *I* that increments the counter, then aborts. Before *I* aborts, a second transaction *I′* may increment the counter further. Thus, the counter value obtained by *I* will not be associated with any data item. That is, there may be gaps in the counter values. Most applications can tolerate such gaps but some cannot. For example, tax authorities do not like to see gaps in invoice numbers.
Figure 2.14 compares the performance of transactions that rely on a system counter facility with the performance of transactions that increment unique identifiers.

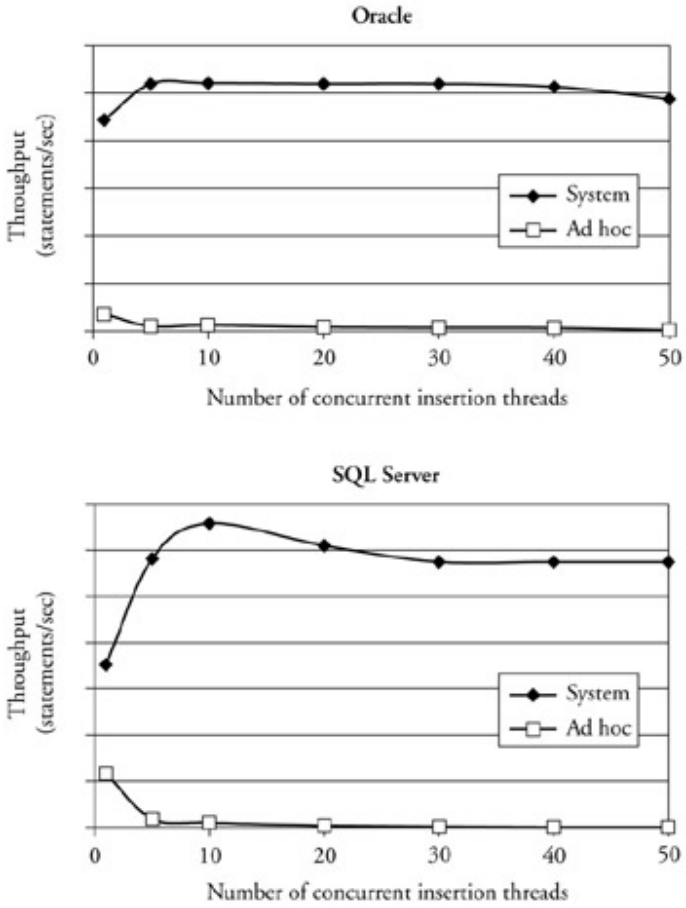


Figure 2.14: Counter facility. There is a significant difference in throughput between *system* insertions that rely on a counter facility for generating counter values (sequence for Oracle, identity data type for SQL Server) and *ad hoc* insertions that explicitly increment a counter attribute in an ancillary table for generating counter values. This difference is due to blocking when accessing the ancillary counter table. Note that in Oracle, we use the default sequence iteration mechanism that caches blocks of 20 sequence numbers. This graph was obtained using SQL Server 7 and Oracle 8i EE on Windows 2000.

[1]Three excellent references on the theory of this subject are the following:

Phil Bernstein, Vassos Hadzilacos, and Nat Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, Mass.: Addison-Wesley, 1987. Now in the public domain at: http://research.microsoft.com/pubs/ccontrol/.
Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann, 1993.

Gerhard Weikum and Gottfried Vossen, *Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery*. San Francisco: Morgan Kaufmann, 2001.

[2]Oracle 8i Parallel Server Concepts and Administration (8.1.5), Section 11.

[3]A latch is a simple and efficient mechanism used to implement exclusive access to internal data structures. A latch is released immediately after access rather than being held until the end of a transaction like a lock. Also unlike a lock, a latch does not allow shared access (in read mode) and does not provide any support for queuing waiting threads.

2.3 Logging and the Recovery Subsystem

Many database management systems make a claim like the following:
Our system has an integrated physical and logical recovery mechanism that protects database integrity in case of hardware and software failures.

The claim is exaggerated, to say the least. After all, an arbitrary software failure could transform a good database state to an arbitrarily erroneous one. (Even the failure to use the serializability isolation level could spoil the integrity of the data as we illustrated earlier.) Similarly, enough hardware failures could cause data to be lost or corrupted. For the best database management systems, the truth is much closer to the following:
Our system can protect database integrity against single hardware failures (of a processor, network, or any disk drive) and a few software failures (failures of the client, a few addressing errors within the server, and fail-stop crashes of the operating system).

Specifically, two kinds of hardware failures can be tolerated:

1. A *fail-stop* failure of a processor and erasure of its random access memory. (Fail-stop means that when the processor fails, it stops. On the hardware of many vendors such as IBM and HP, redundant checking circuitry stops the processor upon detection of failure.)
2. The fail-stop failure of a disk, provided enough redundant disks are available.

Systems that use mirrored disks, dual-powered controllers, dual-bus configurations, and backup processors can essentially eliminate the effect of such errors. In a still relevant paper, Jim Gray reported that Tandem Non-Stop systems have relegated hardware failures to under 10% of their system outages.[4]

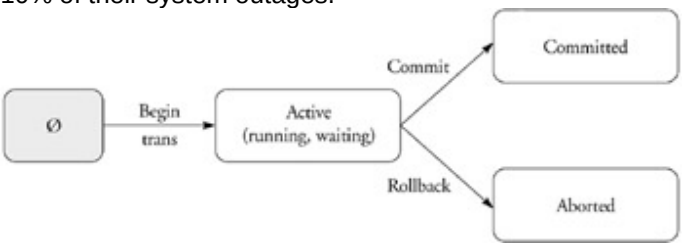


Figure 2.15: Transaction states. Once a transaction commits or aborts, it cannot change its mind. The goal of the recovery subsystem is to implement this finite state automaton.

As a side benefit, certain facilities (like database dumps to remote backups and process pairs) used to protect against hardware failures can be used to recover a database state, at least partially, even in the face of software failures. Many software failures (over 99%) occur only once and cause the system to stop. They are called "Heisenbugs."[5]

From the viewpoint of speed and hardware cost, however, the recovery subsystem is pure overhead and significant overhead at that. To understand how to tune to minimize this overhead, you should understand the basic algorithms for recovery.

2.3.1 Principles of Recovery

Theseus used a string to undo his entry into the labyrinth. Two thousand years later, the Grimm brothers report a similar technique used by Haensel and Gretel who discovered that the log should be built of something more durable than bread crumbs.[6]

Recall that transactions are the unit of isolation for the purposes of concurrency control. Transactions are also the unit of recovery in the following two senses:

1. The effects of *committed* transactions should be permanent. That is, changes should persist even after the completion of the transactions that make those changes.
2. Transactions should be *atomic*. That is, following the recovery from a hardware failure, it should be possible to reconstruct the database to reflect the updates of all committed (i.e., successfully completed) transactions. It should also be possible to eliminate the effects of any updates performed by aborted (also known as rolled back) or unfinished transactions.[7] Figure 2.15 shows the states of transactions.
Achieving the first objective requires putting the data of committed transactions on *stable storage*—storage that is immune to failures. Complete immunity is impossible to achieve, but a good approximation is possible. As a first step, stable storage must be built out of media (disks, tapes, or battery-backed random access memory) that survive power failures. Such media are called *durable*. As a second step, in order to survive failures of durable media, such as disk crashes, the data must be replicated on several units of durable media, such as redundant disks.

Achieving transaction atomicity

Algorithms to achieve transaction atomicity are based on two simple principles.

1. Before a given transaction commits, it must be possible to undo the effects of that transaction, even if random access memory fails. This implies that the *before images* of the updates of the transaction (i.e., the values of the data items before the transaction writes them) must remain on stable storage until commit time. On failure, they can be written to the database disks if they are not already there.
2. Once a transaction commits, it must be possible to install the updates that the transaction has made into the database, even if random access memory fails. Therefore, the *after images* of the updates of the transaction (i.e., the values of the data items that the transaction writes) must be written to stable storage some-time before the commit actually takes place. In that way, if there is a failure any time after the database commit takes place, it will be possible to "redo" the effects of those updates (install the updates into the database).

The sharp (but still uneducated) reader may wonder how it is possible to have both the before and after images on stable storage before the commit point. After all, there is room for only one of them on the database disks.[8]
The answer is simple: stable storage holds more than just the database. The other area on stable storage is called the *log* (Figure 2.16). It may contain after images, before images, or both.

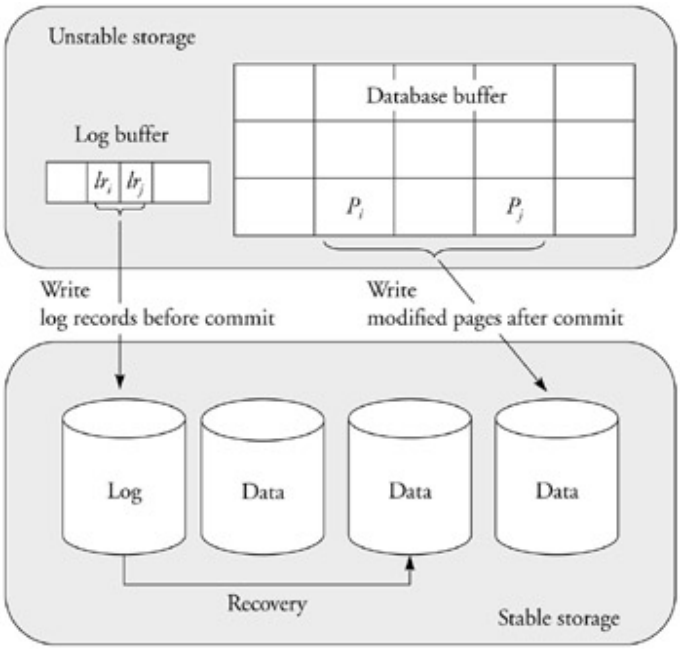


Figure 2.16: Stable storage holds the log as well as data. Consider a transaction *T* that writes values stored on two pages P_i and P_j. The database system generates log records for these two write operations: lr_i and lr_j (which contain the after images of P_i and P_j—or a logical representation of the write operation). The database system writes the log records to stable storage before it commits transaction *T*. The dirty pages P_i and P_j are written after the transaction is committed, unless the database buffer is full and these pages are chosen as victims by the page replacement algorithm. If the buffer fails, then data will be moved from the log to the database disks.

Commercial logging algorithms work as follows:

- The recovery subsystem writes the after images of each transaction's updates to the log before the transaction commits in order to satisfy principle 2. The reader may wonder whether the transaction's after images should be written to the database disks immediately after commit or whether those updates can be delayed. Delaying is better for failure-free performance. Here is why: forcing writes to disk immediately upon commit could require writes to random pages scattered over the disk. This in turn requires disk seeks, a very expensive operation. (Solid state memory caches reduce the seek delay, but they may be overwhelmed if update traffic is heavy.)
- Most systems also write uncommitted after images to the database disks when there is no room left in the database buffer. In order not to jeopardize principle 1, the recovery subsystem must ensure that the log contains the before image of each data item. It can do this by explicitly writing the before image or by using the value written by the last committed write on that data item. This is known as a *redo-undo* or *write-ahead* logging algorithm.

Every logging algorithm establishes the following guarantee:

i. *Current database state = current state of database disks + log*

The current database state is the state reflecting all committed transactions. By contrast, the database disks reflect only the committed transactions physically on the database disks. During normal operation, some transactions may have been committed on the log, but some of their updates may not yet have been written to the data-base disks.

Logging variants

Most commercial systems log an entire page whenever a portion of a page is modified. This is called *page-level logging*. It is also possible to log just a portion of a page, for example, divide the page into some number of portions and log all modified portions. This is called *byte-level logging*. This saves log disk space, especially if the page size is large. A third possibility is to log each changed record. This is called *record-level logging*.
By contrast, there is a technique known as *logical logging* in which the log information is the operation and argument that caused an update. For example, the logged operation might be "insert into employee," and the argument may be "(143-56-9087, Hackett,…)." This technique saves log space because it omits detailed information about index updates. As we mentioned, IBM DB2 uses logical logging and thereby reduces logging overhead.

Logical logging works without complication if the operations are guaranteed to be executed sequentially. This is the case in a main memory database (a database system that holds all the data of each database in memory) that executes transactions serially as well as in certain implementations of database replication.

Checkpoints

To prevent the log from growing too large, the recovery subsystem periodically copies the latest committed updates from the log to the database disks. This act is called a *checkpoint*. Setting the interval between checkpoints is a tuning parameter as we will discuss. Checkpoints cause overhead, but save space in some cases and reduce recovery time.

Database dump

A *database dump* is a transaction-consistent state of the entire database at a given time. A *transaction-consistent* state is one that reflects the updates of committed transactions only. Note that, in contrast to a checkpoint, the database dump consists of the entire database (rather than the latest updates only). Thus a dump is a form of transaction-consistent backup.

If a failure corrupts the database disks, then it is possible to reconstruct the correct state of the database from the previous database dump combined with the log. (In the absence of a dump, a failure of a database disk entails a loss of data, unless the data on that disk is duplicated on at least one other disk.) That is, the dump offers the following guarantee:

ii. *Current database state = log + database dump*

Because the log enters into both equations (i) and (ii), make sure it is reliable. It is usually a good idea to mirror the log, that is, replicate the log on two disks. Further, since the equations show that re-creating the current database state requires the presence of either the disk database or the dumped database, separate the dumped data from the disk database.

Database dumps offer an important side benefit that can be relevant to an application that can afford hardware but has trouble fitting all its long transactions into the batch window. The database dump can be used to populate an archival database (a data warehouse) against which queries can be posed that do not require extremely up-to-date information, for example, decision support queries or statistical queries.

Applications for which the data must be available virtually all the time may require that the entire system be mirrored. In that case, database dumps may not be necessary because one of the two mirrored systems is likely to survive any single failure.

Group commit

If every committing transaction causes a write to the log, then the log disk(s) may become a bottleneck. Therefore, many systems use a strategy known as *group commit*. According to this

strategy, the updates of many transactions are written together to a log from a space in memory sometimes called a log buffer. If your application must support many concurrent short update transactions, then ensure that your database management system offers group commit, that it is turned on, and that the log buffer is big enough.
Figure 2.17 shows the significant performance improvement provided by group commit.

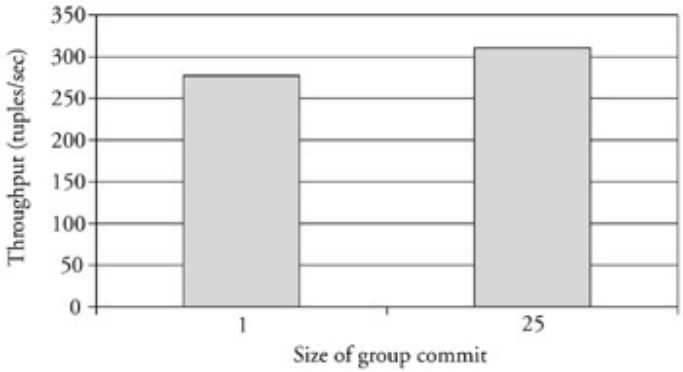


Figure 2.17: Group commit. Increasing the group commit size improves performance. This experiment was performed using DB2 UDB V7.1 on Windows 2000.

The only conceivable disadvantage of group commit is that a transaction's locks cannot be released until its updates have been written to the log. Because group commit delays those updates, group commits will cause locks to be held longer. This may cause a problem with very hot data items.

Some systems such as Informix, however, allow locks on data to be released even while the committed data is in the log buffer but not in the log. This allows the possibility that a transaction might read an updated item, report its result to a user, and then have the system fail along with the update of that item. Such a failure condition is a remote possibility, so many users consider the risk worthwhile.

2.3.2 Tuning the Recovery Subsystem

There are four main ways to tune the recovery subsystem. They can be applied individually or in combination.

1. Put the log on a dedicated disk or disks to avoid seeks.
2. Delay writing updates to database disks as long as possible.
3. Trade desired recovery time against failure-free performance when setting checkpoint and database dump intervals.
4. Reduce the size of large update transactions.

Put log on separate disk

Because the log is on stable storage and because, for many systems, stable storage means disk, transactions that update must perform disk writes to the log. This is bad news. The corresponding good news is that, in the absence of a failure, the log can be written sequentially and in large chunks. Therefore, if a disk has nothing but the log on it, then the disk is written sequentially, rarely does seeks, and hence can maintain a very high I/O rate. So, the very first use of an extra disk should be to segregate log data onto a disk of its own (or several disks if you are mirroring). As mentioned earlier, reliability considerations suggest separating the database disks from the log in any case. Any additional logs (e.g., to support multiversion read consistency or for completely different databases) should again be separate. One of us once consulted at a company that said proudly, "This is our log disk. The logs from all of our

databases go here." It was easy to be a hero that day. The bigger problem was to be nice about it.

Figure 2.18 illustrates the advantage of placing the log on a separate disk.

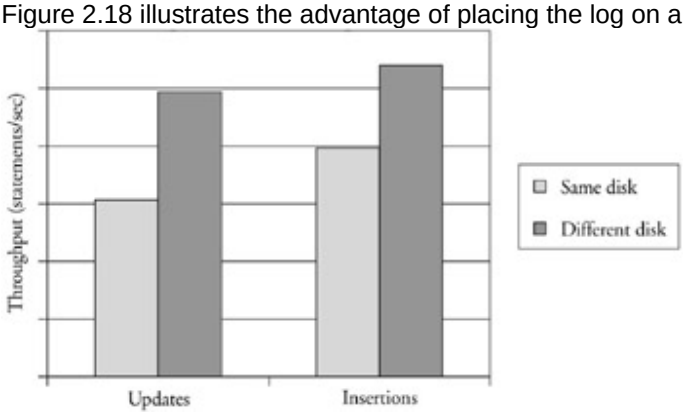


Figure 2.18: Log file location. For this experiment, we use the lineitem table from the TPC-H benchmark and we issue 300,000 insert or update statements. This experiment was performed with Oracle 9i on a Linux server with internal hard drives (no RAID controller). Each statement constitutes a separate transaction, and each transaction forces writes. This graph shows the throughput obtained with the log and the data on separate disks as opposed to the throughput obtained with the log and the data on the same disk. Locating the log file on a separate disk gives approximately a 30% performance improvement. Note that we also performed this experiment using separate disks on a RAID controller. In that case, the disk controller cache hides much of the negative impact of the seeks that are necessary to switch from the log to the data when they are both located on the same disk. The use of the disk controller cache is further discussed in the storage subsystem section.

TECHNICAL DIGRESSION: BACKGROUND COMMENT ON DISKS A disk is a collection of circular platters placed one on top of the other and rotating around a common axis (called a spindle). Each platter, except the top and bottom platters, has two read/write surfaces. (The top surface of the top platter and the bottom surface of the bottom platter are unused.) The data on each surface is held on *tracks*, each of which is a circle. Each platter is associated with a disk *head*. To access the data on track *i* of a given platter, the disk head must be over track *i*. At any time, all disk heads associated with a disk are over the same track on their respective platters. The set of track *i*'s is called the *i*th *cylinder*. Thus, the set of disk heads moves from cylinder to cylinder (Figure 2.19).

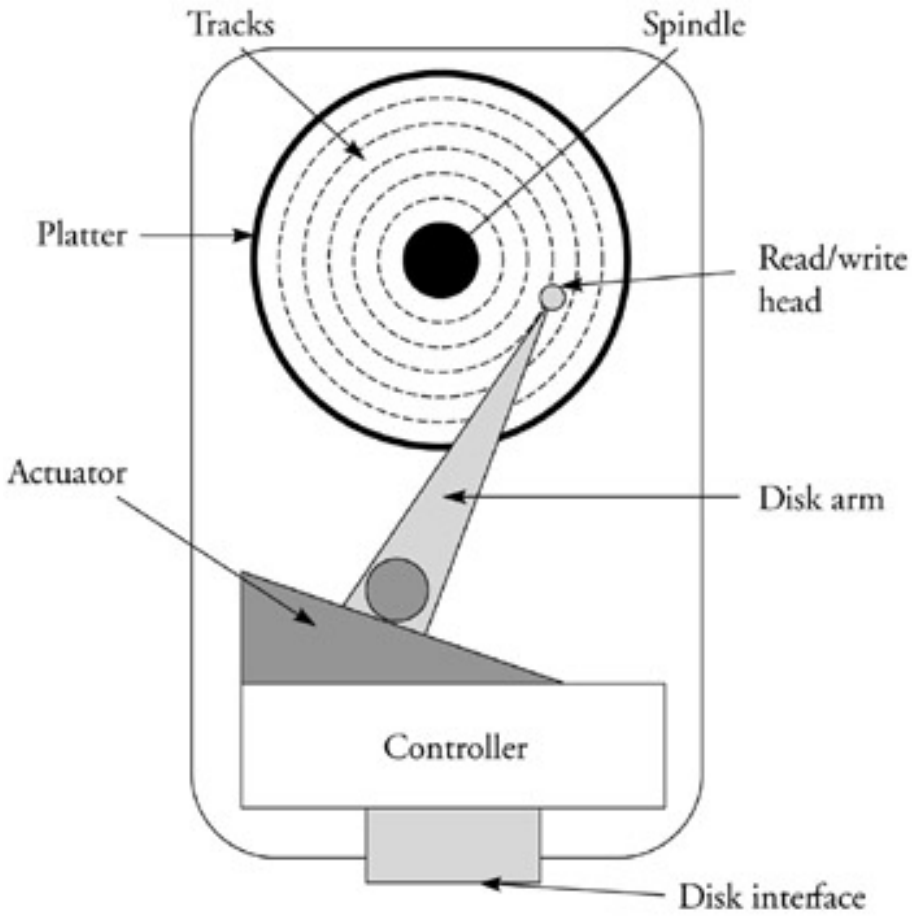


Figure 2.19: Disk organization.
On each disk, a controller is responsible for managing the positioning component, for transferring data between the disk and its clients, and for managing an embedded cache. As data is read off the platters more slowly than it is sent to clients (usually over a SCSI bus[9]), disk controllers partially fill their buffer before transferring data to their clients. Even if controller caches are small, they can be used for caching read (read-ahead) and write requests (immediate reporting). We discuss the use of controller caches in Section 2.5.1.

The time that it takes to access a disk is made up of four significant components.

1. *Controller overhead:* The time it takes to interpret the client commands (about 0.2 ms in 2001).
2. *Seek time:* The time it takes to move the disk head (the reading and writing device) to the proper track (between 4 and 9 milliseconds in 2001).
3. *Rotational delay:* The time to wait until the proper portion of the spinning track is underneath the disk head (between 2 and 6 milliseconds in 2001).
4. *Read/write time:* The time to read or write the data on the spinning track (between 10 and 500 kilobytes per millisecond in 2001).

Moore's law applies to disk storage capacity (disk capacity has increased 100-fold over the last decade), but it does not apply to access time. Gray and Shenoy[10] noted that the ratio between capacity and access time has increased tenfold over the last decade. This means that even if successive generations of disks provide faster access times, disk accesses become more expensive as time goes by.

Rotational delay can be minimized by reading or writing large chunks of data at a time. In fact the time to read or write a track is not much greater than the time to read or write a portion of the track. This explains why reading pages in advance of their need and group commits are important optimizations—an application of the principle that start-up costs are high, but running costs are low.

Seek time can be minimized in two ways.

1. The best way is to ensure that subsequent accesses continue from where previous ones left off. This explains why it is good to keep log data on a disk of its own (unless a memory cache can hide this latency). When writing such data, the operating system will ensure that after one track is filled, writing proceeds on a different track of the same cylinder until the cylinder is filled.
2. The next best way is to put frequently accessed data on the middle track of the disk if the disk is magnetic to reduce seek time.

The effects of seeks and rotational delay can be substantial. In 2001, a good-quality disk can perform 150 random disk accesses per second at best. (A random disk access is one that accesses a cylinder and track independently of previous accesses. So, most of these accesses will involve seek time and rotational delay.) If each access retrieves a 4-kilobyte page, then the total throughput is 600 kilobytes per second. If read (or written) sequentially, the same disk can offer a throughput between 10 and 500 megabytes per second. Thus, a well-organized disk can be a factor of ten or more faster than a poorly organized one for applications that do a lot of scanning.[11]

Tuning database writes

As you already know, before a database transaction commits, it writes the after images of the pages (or records, in some cases) it has updated onto the log. Thus, at the commit point the log (on disk) and the database buffer (in solid state memory) have the committed information, but the database disks may not. That is still sufficient for recovery since the log is a form of stable storage.
As we observed, it would be bad for performance to force the committed writes to the database disks because those writes would tend to be random writes and would require seeks. In fact, some data need never be written to the database disks. For example, suppose item *x* has an initial value of 3. Transaction *T* changes *x* to 5 and commits. At that point, the database buffer and log images of *x* are 5, but the database disks would still have the value 3. If a subsequent transaction *T´* changes *x* to 11 and commits, then the database buffer and log images of *x* would be 11. If *x* is then written to the database disks, the database disks would also have the value 11, but would never have had the value 5. This is a virtue, since we have avoided an entire write to the database disks without jeopardizing recoverability.

Since the buffer is of finite size and the database may be bigger, however, we eventually want to update the database disks for the sake of page replacement. (Even if the buffer could hold the entire database, we might want to update the database disks to decrease the time necessary to recover from a processor or solid state memory failure.)
Now to tuning considerations. Since writes to the database disks are potentially random writes, most database systems try to schedule writes when convenient, for example, write page *p* when the disk head is on a cylinder containing *p*. Such writes are much cheaper than totally random writes.

Some database systems give users certain relevant options having to do with writes to the database disks such as

- when to start convenient writing (how full should the database buffer be before you start these convenient writes). Different systems do this in different ways. For example, Oracle 8 uses an initialization parameter to define how many dirty pages the buffer can hold (DB_BLOCK_MAX_DIRTY_TARGET); once the number of dirty pages is above this threshold, pages are replaced. SQL Server starts replacing pages when the number of pages in the free list falls below a given threshold (3% of the buffer pool size in SQL Server 7).
- how often to checkpoint; a checkpoint is an operation that forces to disk all committed writes that are heretofore only in the buffer and the log.

Setting intervals for database dumps and checkpoints

Setting database dump intervals is a trade-off between the time to recover following a failure of one or more database disks and online performance. The more often the dump is performed, the less data will have to be read from the dumped database following a disk crash. Very few applications require dumps to occur more frequently than once or twice a day.

Some system administrators wonder whether it is worthwhile to perform dumps at all. Dumps offer the following benefits:

- If a database disk fails and there is no dump, then the system will lose data irretrievably. Thus, the dump is an insurance against database disk failure. Note, however, that a log failure will cause data to be lost whether or not there is a dump.
- A dump can be used for data mining queries that do not require completely up-to-date information.

A dump has two costs:

- It increases response time while it occurs.
- It requires space to store.

Recall that a checkpoint forces data that is only on the log and in the database buffer to the database disks and has the following properties:

1. A checkpoint reduces the time and log space needed to recover when there is a failure of random access memory because committed updates will already be on the database disks.
2. A checkpoint does not reduce the log space needed to recover from database disk failures. To recover from such failures, the log must hold all the updates since the last database dump.

The main cost is that a checkpoint degrades online performance, though much less than a dump. Applications that demand high availability should do checkpoints every 20 minutes or so. Less demanding applications should perform checkpoints less frequently.
In case the log file cannot accommodate all the log records that are generated during a transaction, Oracle forces a checkpoint (dirty pages are written to disk and log records are discarded). If the log is not properly sized, multiple checkpoints might thus occur while a transaction executes. Figure 2.20 illustrates the negatives impact of checkpoints on performance in this case.

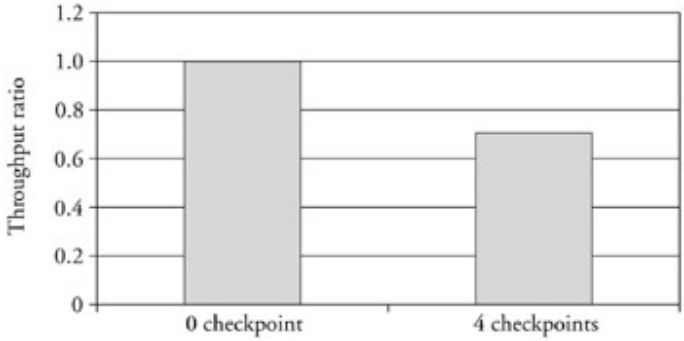


Figure 2.20: Checkpoints. A small log file forces checkpoints. If the log file cannot accommodate all the long entries generated during a long update transaction, it forces dirty data on disk. Here, four checkpoints are triggered: they have a negative impact on throughput. This experiment was performed on Oracle 8i EE on Windows 2000.

From batch to minibatch

A transaction that performs many updates without stringent response time constraints is called a *batch* transaction. If such a transaction is excessively long, the buffer may become full, and a rollback resulting from a failure may be very costly.

An approach around this problem is to break the transaction into small transactions (minibatching). Each minitransaction updates a persistent variable saying how much it has accomplished.

For example, suppose the task is to update a set of customer accounts in sorted order based on customer_ID. Each minibatch transaction can update a set of account records and then put the customer_ID of the last account modified into a special database variable called, say, lastcustomerupdated. These minibatch transactions execute serially. In case of failure, the program will know to continue modifying accounts from lastcustomerupdated onward.
CAVEAT Because this transformation is a form of transaction chopping, you must ensure that you maintain any important isolation guarantees, as discussed in Appendix B.

[4]J. Gray, "A Census of Tandem System Availability, 1985–1990," *IEEE Trans. on Reliability*, vol. 39, no. 4, 409–418, 1990.

[5]Most such "Heisenbugs" occur because of some unusual interaction between different components according to another classic article by E. Adams: "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Development*, vol. 28, no. 1, 1984.

[6]Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann, 1993.

[7]These two rules imply, in practice, that no committed transaction should depend on the updates of an unfinished or aborted transaction. That is ensured by the locking algorithms.

[8]We use "database disks" as shorthand for "the durable media on which the database is stored." Until we discuss disk failures, we will assume that the database disks are stable.

[9]The throughput of the SCSI bus is determined by its width (e.g., Narrow-SCSI at 8 bits and Wide-SCSI at 16 bits) and by its clock rate or frequency (e.g., Ultra-SCSI at 20 MHz, Ultra2-SCSI at 40 MHz, Ultra3-SCSI at 80 MHz).

[10]J. Gray, P. Shenoy, *Rules of Thumb in Data Engineering*, Microsoft Technical Report MS-TR-99-100, March 2000.

[11]We provide programs for evaluating the disk throughput in the book's Web site at http://www.mkp.com/dbtune/.

2.4 Operating System Considerations

The operating system performs several functions that can have a significant impact on database application performance.

- The operating system schedules *threads* of control (called processes in some operating systems), the fundamental units of execution in a computer system. Issues here include the amount of time needed to schedule a new thread (the less the better); the time slice of a database thread (should be long); and whether different threads should have different priorities (database threads should all run at the same priority for most applications). Some database systems such as Sybase implement threads within operating system processes so the overhead of thread switching is potentially lower.
- The operating system manages virtual and physical memory mappings. The issue here is how big to make the portion of virtual memory that is shared by all database threads, the *database buffer*, and how much random access memory to devote to that buffer.
- The operating system controls the number of user threads that can access the database concurrently. The goal is to have enough to accommodate the available users while avoiding thrashing.
- The operating system manages files. Issues here include whether the files can span devices (necessary for large databases); whether the files can be built from contiguous portions of the disk (helpful for scan performance); whether file reads can perform lookahead (also helpful for scan performance); whether accessing a page of a large file takes, on the average, more time than accessing a page of a small file (which obviously should be avoided); and whether a process can write pages asynchronously (useful for the buffered commit strategy discussed earlier).
- The operating system gives timing information. This can help determine whether an application is I/O-bound or processor-bound.
- The operating system controls communication between address spaces on the same site (the same processor or another processor within the same shared memory multiprocessor) and address spaces on different sites. The main issue here is the performance of messages. If they are fast, then database performance will be better. Otherwise, database performance will be worse. There is little you can do about this as a tuner, except take the speed into account when deciding on a distribution strategy.

2.4.1 Scheduling

Each time an operating system schedules a different thread of control, it goes through some computation that is useless to the database application. Therefore, the database tuner should try to minimize the amount of time spent switching contexts. There are two obvious ways to do this.

1. Choose an operating system that has a lightweight thread-switching facility. Many newer operating systems offer this facility. IBM's CICS, although not itself an operating system, introduced this notion in the late 1960s.
2. Minimize the number of such switches. A switch is inevitable when an application makes an I/O request, but optional reasons for switches should be avoided. This means that time slice-driven interrupts should be infrequent for the great majority of database applications that are concerned primarily with high throughput rather than low response

time. A good compromise on modern microprocessors is to give each thread a 1-second time slice. This is enough computation time for most applications and will prevent an infinite loop from hanging the system.

A second aspect of scheduling has to do with thread priorities. Two bad priority decisions can hurt database performance.

1. *Obviously bad decision:* The database system runs at a lower priority than other applications. When those applications consume a lot of resources, the database applications will perform badly.
2. *Subtly bad decision:* Transactions do not all run at the same priority. You may be tempted to give threads that execute transactions with greater importance higher priorities. Unfortunately, this strategy may backfire if transactions with different priorities access and may conflict on the same data item. Consider the following example.

[redacted]

EXAMPLE: PRIORITY INVERSION

Suppose that transaction T_1 has the highest priority, followed by T_2 (which is at the same priority as several other transactions), followed by T_3.

1. Transaction T_3 executes and obtains a lock on some data item X.
2. Transaction T_1 starts to execute and requests a lock on X but is blocked because T_3 has a lock on X.
3. Transaction T_2 now starts to execute (without accessing X). Other transactions of its same priority continue to execute for a long time, thereby precluding T_3 from executing to completion.

Indirectly, T_2 and other transactions of its same priority prevent T_1 from executing. This is called *priority inversion* (Figure 2.21).

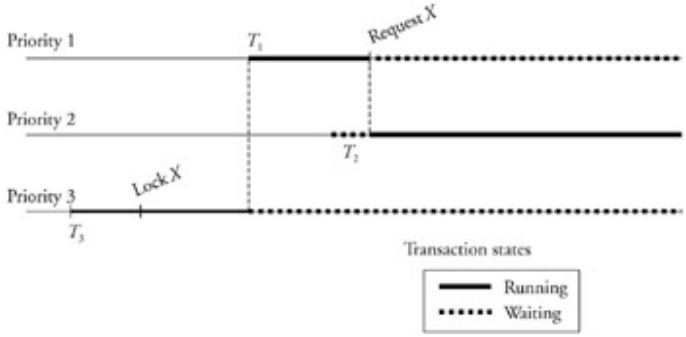


Figure 2.21: Priority inversion. T_1 waits for a lock that only T_3 can release. But the system runs T_2, which has a higher priority than T_3, thus blocking both T_1 and T_3.

[redacted]

Some database systems handle this problem by a method known as *priority inheritance*. The idea is that once a thread acquires a lock, its scheduling priority increases to the maximum level of any thread that is waiting for that lock. As you can see by reviewing the previous example, priority inheritance protects against priority inversion.

- If your system does not protect against priority inversion, then give the same priority to all pairs of transactions that may conflict.

- If your system does protect against priority inversion, then you may choose to give higher priority to online transactions than to batch ones. However, this may generate additional thread-switching overhead, thus hurting throughput.

2.4.2 Database Buffer

Because accesses to disk take much longer than accesses to random access memory, the database tuner must try to minimize the number of disk accesses. One way to do this is to store the entire database in solid state memory (RAM). This is feasible for an increasing number of applications. When it does hold, it is a good idea to bring the entire database into memory at initialization time (say, by reading every record) in order to get the best possible performance.

The goal of memory tuning for all other applications is to ensure that frequently read pages rarely require disk accesses.[12]
Recall that concurrent transactions in a database application share data in a certain portion of virtual memory known as the *buffer*, *database buffer*, or sometimes the *database cache*. The purpose of the buffer is to reduce the number of physical accesses to secondary storage (usually disks). Figure 2.22 shows the components of the buffer.

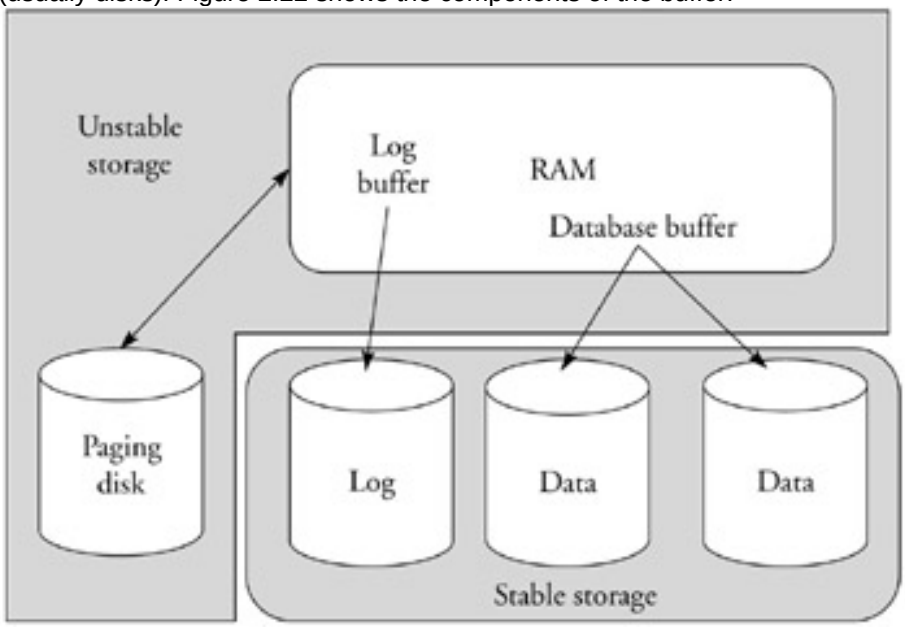


Figure 2.22: Buffer organization. The database buffer is located in virtual memory (RAM and paging file). Its greater part should be in RAM. It is recommended to have the paging file on a separate disk. If not possible, the paging file should be on a data disk rather than on the log disk so that paging does not disrupt sequential access to the log.

The impact of the buffer on the number of physical accesses depends on three different parameters.

1. *Logical reads and writes:* These are the pages that the database management system accesses via system read and write commands. Some of these pages will be found in the buffer. Others will translate to physical reads and writes.
2. *Database management system (DBMS) page replacements:* These are the physical writes to secondary storage that occur when a page must be brought into the buffer, there are no free pages, and the occupied pages have data values that are not present on the database disks. By keeping the database disks as up to date as possible through the use of convenient writes, the tuner can ensure that replacements occur rarely.

3. *Operating system paging:* These are physical accesses to secondary storage (in some systems, a swap disk) that occur when part of the buffer space lies outside random access memory. The tuner should ensure that such paging never happens.
Assuming that paging and page replacements occur rarely, the important question is how many logical accesses become physical accesses. The *hit ratio* is defined by the following equation:

$$\text{Hit ratio} = (\text{number of logical accesses} - \text{number of physical accesses})/\text{number of logical accesses}$$

That is, the hit ratio is the number of logically accessed pages found in the buffer divided by the total number of logically accessed pages.
The tuning parameter that determines the hit ratio is the size of buffer memory. To tune the buffer size, run a typical load for an hour. Check to see whether the hit ratio is too low.
Systems with ample random access memory should aim for a hit ratio of more than 90%. This may be impossible, however, for systems with extremely large databases and lots of random I/O. For example, if 30% of the accesses may touch any page in the database with equal likelihood and the database contains hundreds of billions of bytes, then even a system with a gigabyte-sized buffer will have a hit ratio of 70% or less. So, the best strategy is to *increase the size of the buffer until the hit ratio flattens out, while making sure that DBMS page replacements and operating system paging are low.* Some systems, such as Oracle, offer a utility that will tell the user what the hit ratio would be if the buffer were larger. Figure 2.23 illustrates the impact of buffer size on performance.

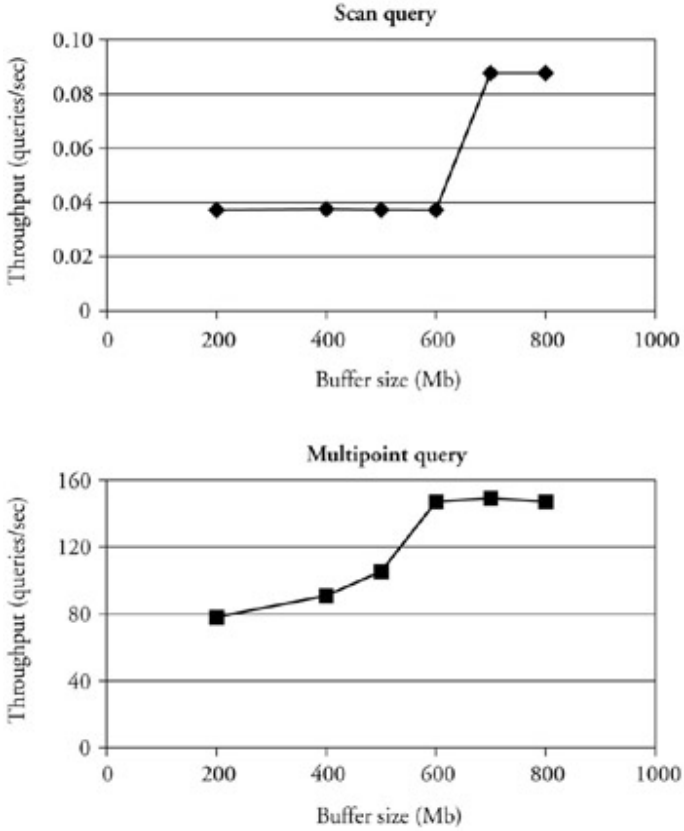


Figure 2.23: Buffer size. These experiments are performed with a warm buffer (the table is scanned before each run). The scan query is processed inside the RAM if the table fits in the buffer. Otherwise, the table is entirely read from disk because the LRU (least recently used) replacement policy will systematically evict pages before they are reread. The performance of the multipoint query increases linearly as the buffer size increases, up to the point where the table fits entirely in memory. This experiment was performed with SQL Server 7 on Windows 2000.

The second tuning parameter is to buy additional random access memory. This should be considered when two factors hold.

1. Increasing the size of the buffer within the currently available random access memory would cause significantly more paging.
2. Increasing the buffer size would increase the database access hit ratio significantly. That is, significantly more logical I/O's would access pages in the buffer.

If some set of applications *X* have much more demanding response time requirements than the rest of the applications *Y* and access different data, then consider dedicating a database buffer to *X* and a different one to *Y*. (This facility is available in DB2, for example.) However, if all applications have basically the same requirements, then use a single buffer.

2.4.3 How Much Memory Is Economical

Even ignoring performance considerations, you may want to move some data into main memory from disk if you access it often enough. Here is some quantitative guidance derived from a paper by Jim Gray and Frank Putzolu.[13]

The question they pose is when does it make sense to keep a particular page of data in random access memory as opposed to bringing it into memory periodically from disk?

Clearly, the more frequently the page is accessed, the more useful it is to put the page in memory. Similarly, the less expensive random access memory is compared with the cost of disk accesses, the more pages should be kept in random access memory. Now let us derive simple equations to guide this decision.

Quantitative discussion

Consider a selection of midrange technology with the following cost characteristics (the method is independent of the numbers you would obtain when performing this calculation):

- An 18-gigabyte disk offers 170 random page accesses per second and costs $300. That is access cost *A* of $1.8 per access per second.
- Random access memory (including support circuitry and packaging hardware) costs $0.5 per megabyte.
- The page size *B* is 8 kilobytes.

Suppose page *p* is accessed every *I* = 200 seconds. Should you keep it permanently in random access memory, considering cost factors alone?

- You save *A/I* dollars in accesses per second, that is, $1.8/200 = $0.009.
- You spend $0.002 for the 8 kilobytes of random access memory.

So, it would be worth it to keep *p* in random access memory. If the interaccess time is greater than 440 seconds (a bit more than five minutes), however, then keeping the page on disk is more economical.

In general, it is better to keep a page in memory if *A/I* is greater than the cost of the random access memory to store the page (the cost of storage on disk is usually negligible by comparison).

2.4.4 Multiprogramming Level

Many system administrators believe that the more concurrent users their system supports the better. It is true that increasing the number of threads of control (usually, in this case, within the database system's server) helps consume idle cycles of underutilized processor(s). However, high multiprogramming levels can actually hurt performance if either of the following limitations is reached:

1. The amount of random access memory the users occupy exceeds the real memory of the system, causing paging either in process space or in the buffer.
2. Lock conflicts arise from the large number of concurrently running transactions.

Various workers have proposed conflicting rules of thumb for setting the amount of concurrency. It turns out that no rule of thumb is valid in all cases, so a better approach if you have a stable application environment is to use the incremental steps method:[14]

- Start with a low bound on the maximum number of concurrent transactions allowed.
- Increase the bound by one and then measure the performance.
- If the performance improves, then increase the bound again. Otherwise, you have reached a local performance maximum. (Theoretical and practical studies indicate that the first local maximum is probably also an absolute maximum.)

If your application is stable but has different transaction profiles at different times during the day, then your bound should change with your application's profiles. Efforts to automate the selection of multiprogramming level have yielded some interesting results. Gerhard Weikum and his students have shown that a critical ratio is the number of locks held by blocked transactions over the total number of locks. (When that ratio is more than about 23%, then the system may thrash.) Unfortunately, such results require ongoing system monitoring and therefore should be performed by a load control facility within the database engine. Maybe some database vendor will give this to us one day.

2.4.5 Files: Disk Layout and Access

File systems allow users to create, delete, read, and write files (which are sequences of either bytes or records). The main tunable parameters for file systems you are likely to encounter follow:

- *The size of disk chunks that will be allocated at one time:* Some file systems call these *extents*. If many queries tend to scan portions of a file, then it is good to specify track-sized (or cylinder-sized) extents for the sake of performance. Write performance can also be improved by using extents. For example, logs and history files will benefit significantly from the use of large extents or other slicing techniques. If access to a file is completely random, then small extents are better because small extents give better space utilization.
- *The usage factor on disk pages:* Oracle 8i offers users the control of how full a page can be and still allow insertions. The higher the *usage factor*, the fuller the page can be when insertions occur. If there are many updates to rows that may make them larger (e.g., conversions of NULLS to non-NULLS) or insertions to a table having a clustering index, then it is good to make the usage factor low (70% or less). Otherwise, use a high usage factor (90% or higher) to improve the performance of table scans. Figure 2.24 illustrates how the disk usage factor affects scan queries.

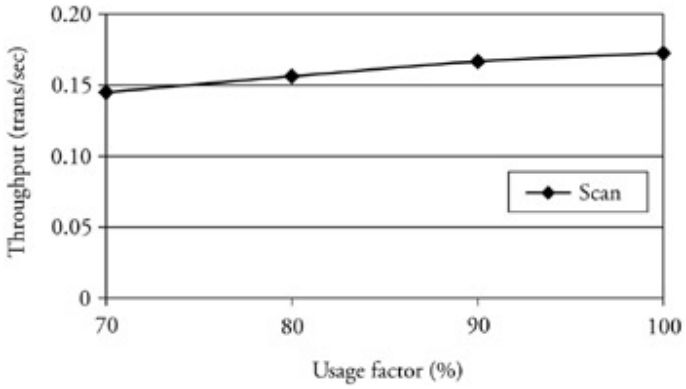


Figure 2.24: Disk usage factor. For this experiment, we use a simple query that scans the lineitem table of the TPC-H benchmark (http://www.tpc.org) when it is located on disk and the memory is cold. We use an aggregation query to reduce the effects of transmitting the lineitem tuples through the database client interface. Throughput increases from approximately 10% as the disk usage increases from 70% to 100%. This experiment was performed using DB2 UDB V7.1 on Windows 2000.

- *The number of pages that may be prefetched:* Again, prefetching is useful for queries that scan files. Unless random access memory is scarce, the number of pages to be prefetched should correspond to a large portion of a track. Oracle has a charming name for this parameter: DB_FILE_MULTIBLOCK_READ_COUNT. Try that as a pickup line at a bar. Figure 2.25 illustrates the effects of prefetching on the performance of scan queries.

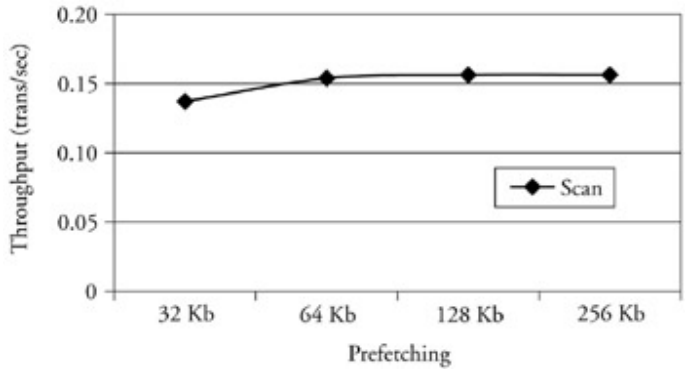


Figure 2.25: Prefetching. We use the lineitem table from TPC-H. The scan query is an aggregation query to reduce the effects of transmitting the lineitem tuples through the database client interface. Throughput increases by about 10% when the prefetching size increases from 32 Kb to 128 Kb and doesn't change thereafter. This experiment was performed using DB2 UDB V7.1 on Windows 2000.

- *The number of levels of indirection to access a particular page:* This is an important issue for Unix-based systems. Because the Unix file index structure interposes more levels of indirection for pages toward the end of a file than for pages near the beginning of the file, it may take much longer to access the former than the latter. To avoid this, many database management systems built on top of Unix use "raw slices."

[12]This discussion concerns the data page buffer, but similar considerations apply to the stored procedure buffer and the other buffers that many database systems offer.

[13]J. Gray and F. Putzolu, "The 5-Minute Rule for Trading Memory for Disc Accesses and the 5-Byte Rule for Trading Memory for CPU Time." *ACM SIGMOD Conference*, 1987.

[14]Hans-Ulrich Heiss and Roger Wagner, "Adaptive Load Control in Transaction Processing Systems." *Proceedings of the 17th International Conference on Very Large Data Bases.* Barcelona, September 1991, 47–54.

2.5 Hardware Tuning

The world would be much simpler if tuning the hardware could be done independently from tuning the software. Unfortunately, the two are intimately related. Although expert tuners often note the processor utilization, I/O activity, and paging, they rarely stop there.

The reason is simple. The fact that a resource is overloaded does not imply that you have to buy more of it. It may be better to rewrite the queries or to add indexes to support important queries, for example. That is, fixing the software may lessen the load on the hardware.

2.5.1 Tuning the Storage Subsystem

Large-scale storage subsystems comprise multiple storage devices, that is, disks or disk arrays, each connected to one or several processing units. The storage subsystem also includes software that manages and configures storage devices, logical volume managers that tie multiple storage devices together, and the file system that arranges data layout on logical volumes.

Tuning the storage subsystem involves

1. configuring the disk array (RAID level)
2. using the controller cache
3. performing capacity planning and sizing
4. configuring logical volumes

Configuring the disk array

A disk array comprises several disks managed by a controller. There are two potential benefits of disk arrays. First, they can provide fault tolerance by introducing redundancy across multiple disks. Second, they can provide increased throughput because the disk array controller supports parallel access to multiple disks.

The different types of disk arrays are known by their RAID (redundant arrays of inexpensive disks) levels. The currently most useful defined RAID levels are

- *RAID 0—striping:* RAID 0 places data evenly across all disks in the array. Consecutive *stripes* of data are placed on the disks of the array in a round-robin manner. For example, if there are eight disks in the array and the stripe size is 1 KB, then the first disk will get kilobyte 1 of the data, the next disk will get the next kilobyte 2, and so on until the eighth disk, which will receive kilobyte 8, and then the first disk will receive kilobyte 9. Thus the stripes on the first disk consist of kilobytes 1, 9, 17, and so on. The first *stripe unit* consists of kilobytes 1–8, the second consists of kilobytes 9–16, and so on.
  In RAID 0, there is no redundancy and thus no fault tolerance. A write operation (spanning multiple stripes) results in a write operation to a physical disk (for each stripe); a read operation (spanning multiple stripes) results in a read operation to a physical disk (or to multiple disks in parallel if several stripes are involved).
- *RAID 1—mirroring:* RAID 1 mirrors one disk to another disk, so there is no striping. There is no downtime (or lost data) if one of the disks fails. A write operation results in a write operation on both physical disks (the write operation terminates when the slowest disk is done writing though variations of this theme in which the operation terminates when the cache has been written are supported by storage devices such as in EMC

products); depending on the controller, a read operation results either in a read operation on both physical disks (the read operation terminates when the fastest disk is done reading) or in a read operation to the disk that is least busy. RAID 1 is a good choice when fault tolerance is needed on a limited volume of data (the limit is the capacity of one disk), such as for a log.

- *RAID 5—rotated parity striping:* RAID 5 relies on error correction, rather than full redundancy, to provide fault tolerance. As in RAID 0, data is striped. In RAID 5, however, each stripe unit contains an additional parity stripe. Parity is a checksum computed as an exclusive-or operation over all data stripes in the unit. Continuing the example used for RAID 0, there may be nine disks that support stripe units having eight data stripes each. Data and parity stripes are distributed evenly among all disks in the array. Initially, all disks are written with null stripes and checksums are computed. Each write to a stripe S requires a change to that stripe and to the parity stripe. The existing data stripe Sold and parity stripe Pold are read, the new parity is computed Pnew := (Sold xor Snew) xor Pold, and the new data and parity stripes are written. This is two disk reads and two disk writes for one data write. Algorithms to eliminate this overhead have been proposed by Savage and Wilkes and by Johnson and Shasha,[15] among others.
- *RAID 10—striped mirroring:* RAID 10 (1+0) stripes data across half of the disks in the array as in RAID 0, and each of these disks is mirrored. RAID 10 provides the read throughput benefits of disk striping (RAID 0) with the fault-tolerance benefits of mirroring (RAID 1). RAID 10 uses twice the disks of RAID 0, so is a worthwhile alternative only when disks are plentiful.

For the striping configurations (RAID 0, RAID 5, and RAID 10), you must decide on the stripe size. We recommend that the stripe size be the database page size since you want the minimum transfer unit to require access to only one disk.

Which RAID level should you use for what?[16]

- *Log file*. Use RAID 1 for log fifiles (and, in Oracle, for rollback segment tablespaces). Mirroring provides fault tolerance with high write throughput. Writes to the log are synchronous and sequential; consequently, they do not benefit from striping. If log writes are frequent and large, RAID 10 might be appropriate. RAID 5 is bad for log files (and for e-mail servers) because of the penalty it imposes on writes though caches can hide that penalty if the writes are bursty.
- *Temporary files*. RAID 0 is appropriate for temporary files or sorting buffers because the system can normally tolerate the data loss resulting from disk failure.
- *Data and index files*. RAID 5 is best suited for applications that require fault tolerance and in which read traffic vastly predominates over write traffic.

A RAID array can be defined at the level of the array controller (hardware RAID) or at the level of the volume manager (software RAID). Which solution is better depends on the hardware configuration and indirectly on price. An array controller with a large cache and a fast on-board processor will perform better than an overloaded server, whereas a multiprocessor with a huge cache will perform better than an array controller with limited computation and memory resources. Some DBAs recommend software RAID rather than hardware RAID to avoid having a controller error corrupt data on multiple disks; this is particularly true when the log is involved. Figures 2.26 and 2.27 illustrate the performance of write- and read-intensive workloads on different levels of software and hardware RAID. With our hardware configuration, RAID 5 works well even for a mixed workload because of the cache. This conclusion is supported by the experiment illustrated in Figure 2.28 that shows that the controller buffer is seldom overwhelmed in practice.

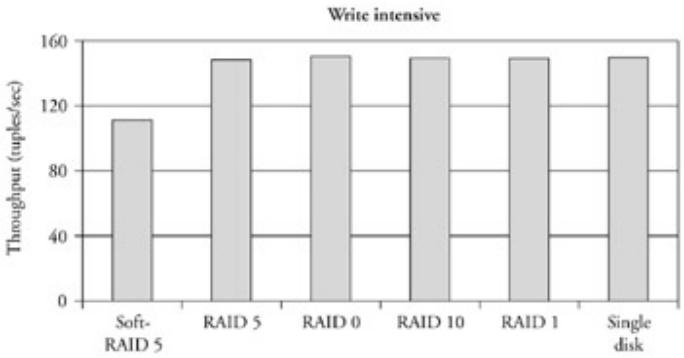


Figure 2.26: RAID and write-intensive applications. The negative impact of the additional read and write operations required by RAID 5 is obvious for software RAID 5. When the controller is responsible for these operations (hardware RAID 5), however, it manages to hide their impact on performance thanks to its cache. This experiment was performed using SQL Server 7 on Windows 2000.

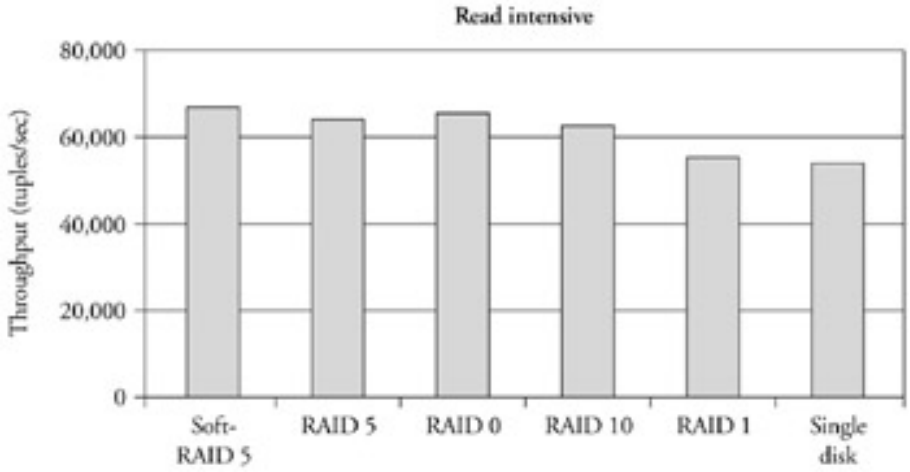


Figure 2.27: RAID and read-intensive applications. RAID 1 slightly improves on a single disk solution. The striped RAID levels (RAID 0, RAID 5, and RAID 10) significantly improve read performance by partitioning the reads across multiple disks. This experiment was performed using SQL Server 7 on Windows 2000.

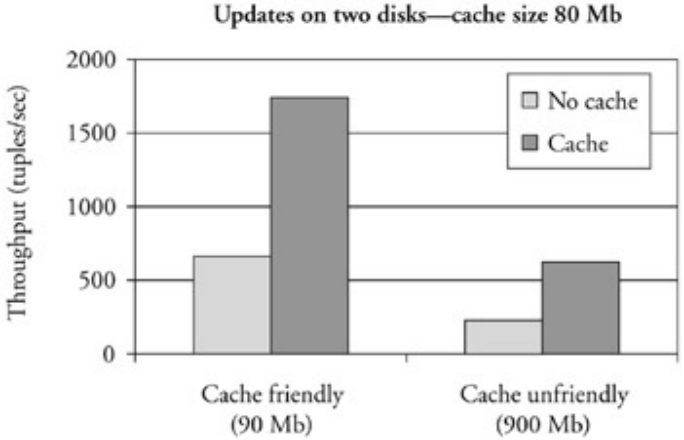


Figure 2.28: Controller cache. Using the cache controller (in write-back mode) provides similar benefits whether the write-intensive application is *cache friendly* (the volume of update is slightly larger than the controller cache) or *cache unfriendly* (the volume of update is ten times larger than the controller cache). This experiment was performed using SQL Server 7 on Windows 2000.

Using the controller cache

Disk and array controllers contain memory that can be used as a read cache or as a write cache. Most controller software allows selecting the read and write cache functions independently.
The read cache performs *read-ahead*: once the disk mechanism completes a read request, it continues to read data into the cache. As we have discussed, database systems already provide a prefetching mechanism that operates at a finer grain than the controller read-ahead mechanism. Indeed prefetching can be defined for files where sequential reads are most probable while it is not defined for files where random reads are most probable. None of the database system vendors recommend using the controller read-ahead capability.
The write cache performs *write-back*: the write requests terminate as soon as data is written to the cache. Contrast this with the default *write-through* mode where the write requests terminate as soon as data is written to disk. In write-back mode, the controller must guarantee that all the write operations that have terminated are actually performed on disk. The controller must preserve cached data in case of power or system failure and flush it to disk upon start-up.

Intuitively, write-back seems always better than write-through. This is indeed the case when the controller is lightly loaded. However, write-through might perform better than write-back when a controller cache connected to several disks fills completely. Consider a controller connected to six disks. When its cache is full, the controller cache replacement policy kicks in to free space for the incoming requests. Usually the cache replacement policy writes data blocks to one disk at a time. As a result, when the cache is full, requests are serialized and the waiting time for each request depends on disk access time and on the length of the queue of waiting requests: the controller cache becomes a bottleneck. With write-through, each request accesses a disk directly and the parallelism across disks can be exploited. It is thus generally not recommended to use the write-back mode for controllers attached to heavily loaded database servers.
Figure 2.28 illustrates the performance of two write-intensive applications using the controller cache. This experiment shows that controller caches handle large though perhaps not enormous volumes of write requests well, again even with a RAID 5 configuration.

Disk capacity planning and sizing

Capacity planning includes the problems of deciding on the number and types of disks and the interconnection between the disks and the processors.

1. *The number of disks*. Because striping makes several disks act like a (faster) single logical disk, the question here is how many logical disks should there be. The lower bound in a high-performance system is dictated by considerations such as that each log file should be on its own logical disk (probably with redundancy), and similarly for each rollback segment. Secondary (nonclustering) index files should be located on different disks than data, as we will discuss in the next chapter. The catalog should be separated from user data, as we discussed earlier. Once you have this minimum number of disks, you can compute the aggregate expected bandwidth on each disk to see if still more are needed.
2. *The characteristics of each disk*. We have described disk characteristics in Section 2.3.2. Seek time, rotational latency, cache size, and data transfer rate influence the performance of the storage subsystem. For SCSI drives, the parameters of the SCSI interface (its clock rate and data path width) are also important.
3. *The characteristics of the interconnect*. SCSI is, as of this writing, the interconnection method of choice for database systems because it is mature and cost-effective. In 2001, the top-performing systems on the TPC-C benchmark used SCSI interconnects. However, SCSI buses are limited with respect to fault tolerance, scalability (16 devices per bus is the upper limit), and to a lesser degree speed (40 Mb/sec for Ultra-

2 Wide SCSI and 160 Mb/sec for Ultra-3 Wide SCSI). Storage-area networks based on fiber channels have started to emerge to overcome SCSI limitations. Each fiber channel loop can support up to 126 devices. Up to 200 loops can be connected to multiple processing units through hub and switches. To support high availability, disks can be added and removed while the interconnect remains in operation. Fiber channel is also faster than mid-2001 SCSI buses with a 106 Mb/sec throughput. Storage-area networks are best suited for shared disk architectures (see Section 2.5.2).

Configuring logical volumes

Logical volumes are a software layer that abstracts several disks into a single logical disk for higher layers. Logical volumes can be used to create a RAID device in software that appears as a single logical disk or can combine several physical storage devices into a single large disk abstraction. Multiple volumes can even be defined on a single physical storage device (a disk or a disk array) though we do not recommend this for frequently accessed volumes because this configuration would then entail significant head movement when accessing data from different volumes. As discussed, RAIDs constructed as logical volumes can give high performance provided the disk caches are large.

2.5.2 Enhancing the Hardware Configuration

There are three main ways to add hardware to your system: add memory, add disks, or add processors. This section discusses each and the general considerations guiding each purchase.

Add memory

In most systems, the cheapest hardware improvement is to add memory and then to increase the size of the buffer pool. This will reduce the load to the disks because it increases the hit ratio without increasing paging. An alternative is to buy memory for the disk cache, but processor memory is more versatile—it can be used for data from any disk and can be used for other purposes such as large sort operations.

Add disks

Obviously, you have to add disks if the amount of data you want to store exceeds the capacity of the disks, but this is unusual in most applications. In fact, most database applications underutilize their disks because disk bandwidth is the critical resource rather than disk space. Improvements to disk bandwidth can be achieved by buying more memory for disk caches or by buying more disks. Buying more disks is a good idea to achieve the following goals:

1. Put the log on a separate disk to ensure that writes to the log are sequential, as we discussed in the recovery section.
2. Use a different RAID level (e.g., switch from RAID 5 to RAID 10) to achieve better performance in update-intensive applications.
3. Partition large tables across several logical disks. Underutilize the space on your disks to ensure that they can handle the access load. Depending on your application, you may choose two different partitioning strategies.

- Write-intensive applications should move nonclustering (secondary) indexes to a disk other than the one on which the data resides. The reason is that each modification will have to update most of the indexes as well as the table, thus balancing the load.
- Read-intensive applications should partition frequently accessed tables across many logical disks. This balances the read load across the disks.

An ascendant disk technology provided by EMC and other vendors is to bundle disks with intelligent controllers and lots of solid state memory and have these powerful storage devices communicate with one another through *storage-area networks*. These configurations offer caching of disk requests and distribution from one storage device to another. The distribution is quite sophisticated. For example, EMC offers storage-area networks that will do replication for all or part of a storage device. The replication may go from device A to B for some disk blocks and from B to A for other disk blocks. The replication can be synchronous or asynchronous and can be turned on and off. When replication is turned off and then turned on, resynchronization will bring the backup up to date with respect to the primary. This variety of replication options allows, for example, an online transaction storage device to be replicated to a data warehouse storage device every night and then, during the day, to turn off the replication to avoid overhead to the primary and to ensure a consistent image of the database to warehouse users. The great advantage of storage-area networks is that the database administration required for replication is greatly reduced since the database management system need not even be aware of the replication. This gain may overshadow the hardware costs. The following rule of thumb may be helpful: if the items to be replicated are small and widely dispersed, then the storage-area network approach is less attractive because it replicates entire blocks. On the other hand, databases that are rewritten sequentially and in bulk are good targets for storage-area networks.

Add processors

Software is a gas; it expands to fit the container it's in.

—Nathan Myhrvold, Group VP of Applications and Content for Microsoft 1997.

It always seems possible to overload a processor. This is particularly true of application programs whose pretty pictures burn cycles with high efficiency. But it is also true of database systems for the simple reason that database scans and sorts are at least linearly proportional to size, and database sizes increase in proportion to our desires for information.

For example, at the ACM Sigmod Conference in 1999, Dirk Duelmann gave a talk about a 25-year-long physics experiment called the Hadron Collider project at CERN. (Hadrons are heavy particles.) Experiments are done in a vast evacuated ring that is 27 kilometers in diameter. Particles are accelerated to 0.035 km/h below the speed of light. Detectors get information about particles and their trajectories. Particles are bent by a magnetic field, so trajectories tell you the speed and mass. Each particle gives data so a larger number of particles gives much more data. Given raw data, the system produces tracks using a low-level filter. The data input is something like a terabyte per day and increasing.

Few of us have to deal with petabyte databases, but a terabyte comes up a lot. For example, all stock market trades and quotations for the last ten years (roughly, four billion ticks) approaches a terabyte. Queries that attempt to find correlations among groups of stocks or patterns among up and down ticks require scans.

The main decision you face when confronted with a large database is how to design your multiprocessor system. The cheapest way is to connect many independent computing systems together by a high-speed network in a so-called shared nothing environment. This can work well in the following situations:

1. Part of your application (e.g., the user interface) can be off-loaded to a separate processing system.
2. Your application can be divided into transaction processing and decision support, and the decision support application can access slightly old data without penalty. In that case, you can dump the online database to the backend every so often (e.g., every night) and allow read-only queries to go against the backend database. You might even use a

different database management system for the backend as we discuss in Chapters 9 and 10.

3. Allow read-only transactions to span several sites before allowing update transactions to span several sites.
Reasons. First, some systems will not guarantee transaction atomicity if updates occur at several sites. Second, read-only transactions incur less overhead then update transactions when they span sites. (The overhead of the "two-phase commit" protocol may be as much as four messages per site for updates, whereas the protocol is optimized away for read-only transactions.)
4. Partition the data (this applies to read-only and update applications). Partitioning is an art, but the basic goal is simple: every long-running query should have its work evenly distributed over as many processing systems as possible.

   In the case of stock quotes, for example, we might partition stocks among the processing systems, so a given stock has its entire history on one processing system. This works well for queries on the time series of individual stocks and on groups of stocks in the same partition. Alternatively, we might partition the data by time (round-robin by day), where each processing system contains information about all stocks for a given time interval. Correlations between stocks at time intervals would then be easy. Partial correlations are performed at each site and then roll-ups are performed at a master site. This is how KDB from Kx Systems performs time series queries, for example.
   But partitioning generalizes far beyond this example and is the basis for Teradata's data layout. Data in each of many tables is hash- or range-partitioned. Hash partitioning a table based on an attribute *A* and hash function *h* will place each row *r* on site *h*(*r.A*). (For example, if the hash function is "mod 17," then rows with *A* values 20, 27, and 37 will be placed on sites 3, 10, and 3, respectively.) Range partitioning a table based on an attribute *A* will place all rows having a range of *A* values on a certain site. Hash partitioning supports key-based queries very well and also supports parallel scans because the rows will be quite evenly distributed (if the partitioning attribute is a key). Range partitioning may fail to partition rows evenly if the ranges aren't equally populated. In these situations, the choice of partition attribute (or attributes) can also affect join behavior. For example, if *R* is partitioned based on attribute *A* and hash function *h* and *S* is partitioned based on attribute *B* and hash function *h*, then the join clause *R.A = S.B* can be computed locally on each site. These observations apply to transactional databases, enterprise resource planning systems, and data warehouses.
5. Certain applications require "tight coupling"—there is no partition of the data that satisfies all important application requirements (or the application is too ill-specified to be sure). In that case, a shared memory multiprocessor works better than a loosely coupled approach. It is also easier to expand such an application by buying a bigger multiprocessor. The main disadvantage is that such architectures are fundamentally not as scalable as the partitioned applications. Figure 2.29 illustrates how the degree of parallelism in a multi-processor system influences the performance of sequential and random accesses.

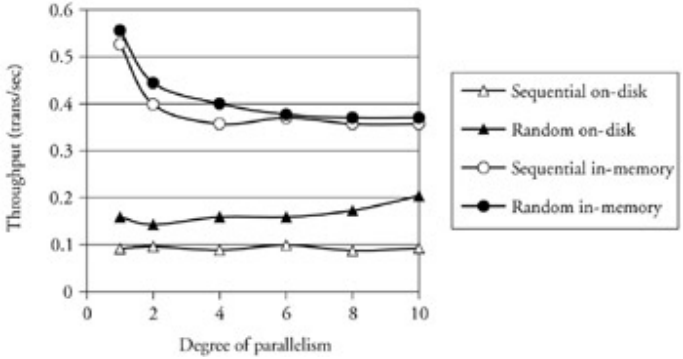


Figure 2.29: Degree of parallelism. This experiment is performed with DB2 UDB V7.1 on

a dual-processor shared memory multiprocessor running Windows 2000. When data is located in memory, the CPU is the critical resource and the cost of synchronization (between the threads that produce the data and the threads that consume them in order to transmit them to the client) and of context switches is higher than the benefit of increased data throughput that would result from increased multithreading. By contrast, when data is located on disk, a higher degree of parallelism significantly improves the performance of random access transactions as are typically found in online transaction processing. (The improvement for random access is further aided by the list prefetching mechanism implemented in DB2 that collects random accesses to a table and sorts them in order to minimize disk seek time.) For sequential access transactions such as are found in data warehouse applications, the disk is the bottleneck, and as a result, the throughput remains constant with increasing levels of multithreading.
Corresponding to the type of parallelism you want, there are three hardware architectures to choose from (Figure 2.30).

- Have several processors, but a single logical main memory and set of disks. Such an architecture is called *tightly coupled* and is typical of bus-based multiprocessors.

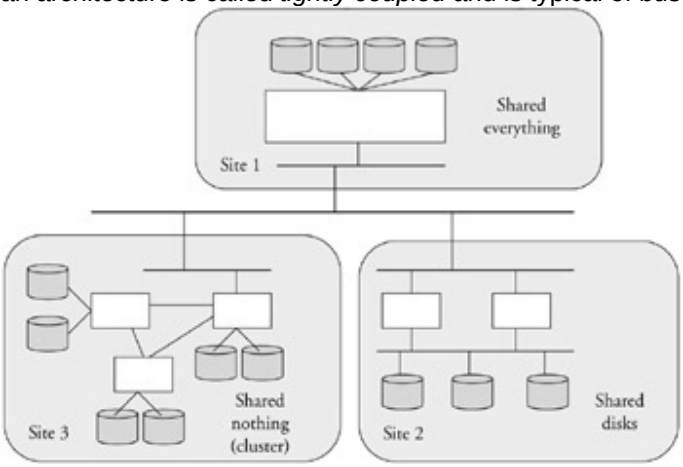


Figure 2.30: Hardware architectures. This diagram illustrates the various hardware architectures described in the text. Site 1 implements a shared everything architecture: a mainframe provides tight coupling. Site 2 implements a shared disk architecture. This is an increasingly popular alternative to tight coupling—disks are either tethered to servers or directly attached to the network (NAS). Site 3 implements a shared nothing architecture with a cluster (where each node usually implements tight coupling). A shared nothing architecture is actually implemented between these different sites.

- Have several processors, each with its own main memory and disks. Such an architecture is called *shared nothing* and is the one to use in an application that distributes work evenly (e.g., the stock tick application) or an application that can be decoupled into independent subapplications.
- Have several processors, where each has its own main memory but the processors share disks. This architecture is called *shared disk* and should be used when a tightly coupled architecture would be ideal, but the application overpowers a single tightly coupled site.

Bibliography

E. Adams. *Optimizing preventive service of software products. IBM Journal of Research and Development*, 28(1), 1984. This article analyzes the causes of Heisenbugs.

Steve Adams. *Oracle 8i Internal Services for Waits, Latches, Locks and Memory*. O'Reilly, 1999. This small book details the internals of Oracle's concurrency control mechanism. The author manages a very detailed FAQ about the internals of Oracle for Unix at http://www.ixora.com.au/q+a.

Phil Bernstein, Vassos Hadzilacos, and Nat Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987. The classic practical theory book. It's outdated but treats the basic concepts very well. It is available online at http://www.research.microsoft.com/pubs/ccontrol/.

Don Chamberlain. *A Complete Guide to DB2 Universal Database.* Morgan Kaufmann, 1998. This book describes DB2 functionalities. It is mainly aimed at application developers.

Kalen Delaney. *Inside MS SQL Server 2000*. Microsoft Press, 2000. This book is a revision of the SQL Server 7 edition.

Jim Gray. *A census of tandem system availability. IEEE Trans. on Reliability*, 39(4):409–418, 1990. Analysis of failure causes on Tandem Non-Stop SQL.

Jim Gray and Goetz Graefe. *The five-minute rule ten years later, and other computer storage rules of thumb. SIGMOD Record*, 26(4):63–68, 1997. Review of the five-minute rules and other rules of thumb for storage management.

Jim Gray and Gianfranco R. Putzolu. *The 5-minute rule for trading memory for disk accesses and the 10-byte rule for trading memory for CPU time.* In *Proceedings of the ACM SIGMOD 1987 Conference,* San Francisco, May 27–29, 1987, 395–398. ACM Press, 1987. This paper introduces the five-minute rule as an economic threshold for moving data from disk into memory.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993. The authors have a lot of experience building concurrency control and recovery systems, so if you are building a transactional system, this book's code and its perspectives provide an excellent start.

Jim Gray and P. Shenoy. *Rules of thumb in data engineering. Technical Report MS-TR-99-100, Microsoft Research*, 2001. Latest in the series of J. Gray's rules of thumb papers.

Hans-Ulrich Heiss and Roger Wagner. *Adaptive load control in transaction processing systems.* In *17th International Conference on Very Large Data Bases*, September 3–6, 1991, Barcelona, Catalonia, Spain, Proceedings, 47–54. Morgan Kaufmann, 1991. Incremental method for setting the amount of concurrency in a transaction processing system.

Ken Jacobs. *Concurrency control, transaction isolation and serializability in SQL92 and Oracle 7. Oracle white paper*. July 1995. This paper describes Oracle snapshot isolation.

Kevin Loney and Marlene Thierault. *Oracle9i DBA Handbook.* Oracle Press, 2001.

Richard Niemiec. *Oracle8i Performance Tuning*. Oracle Press, 1999. This book contains numerous tuning tips for Oracle as well as queries for monitoring performances.

Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1999. This book treats all aspects of distributed database systems, including query processing. Essential reading if you are building one.

Stefan Savage and John Wilkes. *AFRAID: A frequently redundant array of independant disks.* In *1996 Usenix Technical Conference*, 1996. Presents an algorithm to ameliorate the performance of parity checking in a disk array.

Ron Soukup and Kalen Delaney. *Inside MS SQL Server 7.0.* Microsoft Press, 1999. This book contains very complete sections on capacity planning and sizing as well as on the storage subsytem. It also introduces rules of thumb for tuning SQL Server 7.

John Toigo. *The Holy Grail of Data Storage Management.* Prentice Hall, 2000. This book reviews current storage management techniques and discusses strategies for administering large storage systems.

David Watts, Greg McKnight, Peter Mitura, Chris Neophytou, and Murat Gulver. *Tuning Netfinity Servers for Performance.* Prentice Hall, 2000. Operating system tuning for storage management and database performances.

Gerhard Weikum and Gottfried Vossen. *Transactional Information systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2001. An encyclopedic treatment of theory and some practice of all research in transaction processing. From workflow to data structure concurrency control, this book has both breadth and depth.

Exercises

As a kind of self-test, see if you can figure out what to do in the following situations. All situations begin when someone tells you that his or her application runs too slowly.

EXERCISE 1

The Employee table is on disk 1, the Student table is on disk 2, and the log is on disk 2. The Student table is smaller than the Employee table but is accessed more often. Disk 2 supports more than twice the I/O rate of disk 1. The customer is willing to buy a new disk. What should you do with it?
Action. Probably the best thing to do is to put the log on the third disk. The log works much better in that case (i.e., as a sequential storage medium), and the system can tolerate the failure of a database disk if there is also a database dump on tape.

EXERCISE 2

Response time is quite variable. You learn that new tables are added to the database concurrently with online transactions. Those new tables are irrelevant to the online transactions.
Action. Suggest that those new tables be added outside the online window. The reason is that any DDL statement will tend to interfere with online transactions that must access table description (catalog) information.

EXERCISE 3

A new credit card offers large lines of credit at low interest rates. Setting up a new card customer involves the following three-step transaction:

1. Obtain a new customer number from a global counter.
2. Ask the customer for certain information, for example, income, mailing address.
3. Install the customer into the customer table.

The transaction rate cannot support the large insert traffic.

Action. The interview with the customer should not take place while holding the lock on the customer number counter. Instead, step 2 should occur first outside a transactional context. Then steps 1 and 3 should be combined into a single transaction. If obtaining the customer number causes lock contention, then obtain it as late as possible, or use a special counter facility such as sequences in Oracle or identity fields in SQL Server.

EXERCISE 4

The business intelligence department would like to run data mining queries on the sales data. While they run their queries, arbitrary updates may occur. This slows down both the updates and the data mining queries.
Action. Partition in time or space. Either run the data mining queries at night or run them on a separate database system. As Celko writes later, "Want to tell people that the payroll will be a day late because someone in marketing decided to see if there is a relationship between 25 independent variables?"

EXERCISE 5

An application supports thousands of inserts per hour and hundreds of thousands of short update-only transactions per hour. The inserts come packaged as large transactions every 20 minutes or so and last for 5 minutes. When the inserts enter, update response time goes up to 15 to 30 seconds, deadlocks occur, and one of the disks shows exceedingly high utilization. In between the insert bursts, response time is subsecond.
Action. The inserts appear to monopolize the system, and the data appears to be poorly partitioned. Two changes in conjunction or in isolation are likely to help.

1. Smooth out the insert traffic by chopping the large insert transactions into small ones (if this is possible as far as concurrent correctness is concerned) and issuing them one at a time.
2. Repartition the data so that the insert traffic is spread to different disks but the updates still enjoy fast access. A clustering index on a nonsequential key (see Chapter 3) will work well.

EXERCISE 6

The system is slow due to excessive processor utilization. An important (relational) transaction executes an SQL query that accesses a single record from within a loop in the programming language.
Action. Replace the loop by a single query that accesses the records and then allows the programming language code to iterate over an array. This will save time because programming language to database interactions are expensive, whereas array-based access is less so.

EXERCISE 7

The disks show high access utilization but low space utilization. The log is on a disk by itself. The application is essentially read-only and involves many scans. Each scan requires many seeks. Management refuses to buy more disks.
Action. Consider reorganizing the files on disk to occupy large sequential portions of disk. Raise the prefetching level and increase the page utilization. This will reduce the seek time and the number of disk accesses.

EXERCISE 8 The new credit card company is now established. It bills its customers on the last Thursday of each month. The billing transaction takes all night, so other necessary batch jobs cannot be accomplished during this time. What should you do?
Action. The first question to ask is whether the application has to work this way. If 1/20 of the bills could be sent out every working day, then the billing application would create fewer

demands on the system each day (partitioning in time). Another approach is to run the billing job as a big batch job but only on the weekend.

[15]Stefan Savage and John Wilkes, *AFRAID—A Frequently Redundant Array of Independent Disks.* USENIX Technical Conference, January 22–26, 1996.

Theodore Johnson and Dennis Shasha, *Fault Tolerant Storage System*. April 2001, U.S. Patent 6,219,800.

[16]Oracle recommends to stripe and mirror everything. Juan Loaiza, Optimal storage configuration made easy. Oracle white paper.

Chapter 3: Index Tuning

3.1 Goal of Chapter

An *index* for a table is a data organization that enables certain queries to access one or more records of that table fast. Proper tuning of indexes is therefore essential to high performance. Improper selection of indexes can lead to the following mishaps:

- Indexes that are maintained but never used
- Files that are scanned in order to return a single record
- Multitable joins that run on for hours because the wrong indexes are present (we have seen this)

This chapter gives you guidance for choosing, maintaining, and using indexes. (Figure 3.1 shows the place of indexes in the architecture of a typical database system.) The hints apply directly to relational systems. If you use a nonrelational system, a VSAM-based, hierarchical, network-based, or object-oriented database system, then you will have to translate a few of the examples in your head, but you will find that the principles remain relevant.

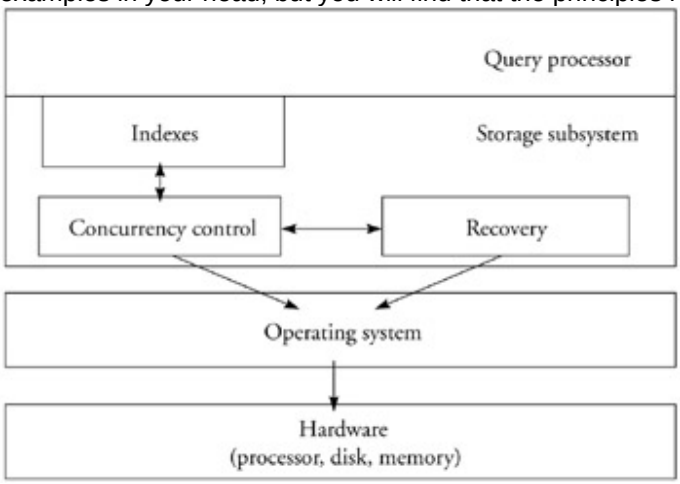


Figure 3.1: Place of indexes in the architecture of a typical database system. Indexes are provided by the storage manager. They organize the access to data in memory and, for clustering indexes, also organize the layout of data on disks. Indexes are tightly integrated with the concurrency control mechanisms. They are heavily used by the query processor during query optimization.

3.2 Types of Queries

The usefulness of an index depends on how queries use the index. For example, if there is an index on attribute *A*, but no query ever mentions *A*, then the index entails overhead (for maintenance on inserts, deletes, and some updates) without yielding any benefit. This is obviously folly. Less obvious sources of folly can result from placing the wrong kind of index on an attribute with respect to the queries performed on that attribute. Since an infinite number of queries are possible, we will abstract the queries into a few "types." Later, we'll characterize the strengths of each kind of index with respect to these types.

1. A *point query* returns at most one record (or part of a record) based on an equality selection. For example, the following returns the name field value of the single employee with ID number 8478:

```
SELECT name
FROM Employee
WHERE ID = 8478
```

5. A *multipoint query* is one that may return several records based on an equality selection. For example, if many employees may be in the same department, then the following is a multipoint query:

```
SELECT name
FROM Employee
WHERE department = "human resources"
```

A *range query* on attribute *X* is one that returns a set of records whose values lie in an interval or half-interval for *X*. Here are two examples.

```
SELECT name
FROM Employee
WHERE salary >= 120000
AND salary < 160000

SELECT name
FROM Employee
WHERE salary >= 155000
```

A *prefix match query* on an attribute or sequence of attributes *X* is one that specifies only a prefix of *X*. For example, consider the attributes last name, first name, city (in that order). The following would be prefix match queries on those attributes:

- last name = ‘Gates’
- last name = ‘Gates’ AND first name = ‘George’
- last name = ‘Gates’ AND first name LIKE ‘Ge%’
- last name = ‘Gates’ AND first name = ‘George’ AND city = ‘San Diego’

The following would not be prefix match queries because they fail to contain a prefix of last name, first name, and city:

- first name = ‘George’
- last name LIKE ‘%ates’

An *extremal query* is one that obtains a set of records (or parts of records) whose value on some attribute (or set of attributes) is a minimum or maximum. For example, the following query finds the names of employees who earn the highest salary:

```
SELECT name
FROM Employee
WHERE salary = MAX(SELECT salary FROM Employee)
```

An *ordering query* is one that displays a set of records in the order of the value of some attribute (or attributes). For example, the following query displays the Employee records in the order of their salaries. The first record is an Employee having the lowest salary, and the last record is an employee having the highest salary.

```
SELECT *
FROM Employee
ORDER BY salary
```

A *grouping query* is one that partitions the result of a query into groups.Usually, some function is applied to each partition. For example, the following query finds the average salary in each department:

```
SELECT dept, AVG(salary) as avgsalary
FROM Employee
GROUP BY dept
```

31. A *join query* is one that links two or more tables. If the predicates linking the tables are based on equality, the join query is called an *equality join query*. For example, the following query finds the Social Security number of employees who are also students:

32. SELECT Employee.ssnum

33. FROM Employee, Student

34. WHERE Employee.ssnum = Student.ssnum

Join queries that are not equality join queries require an examination of nearly every pair of records in the two tables even when indexes are present. Whenever possible, systems process such queries as selections following an equality join query. For example, the following query finds all employees who earn more than their manager.

```
SELECT e1.ssnum
FROM Employee e1, Employee e2
WHERE e1.manager = e2.ssnum
AND e1.salary > e2.salary
```

The system will first perform the join based on

```
e1.manager = e2.ssnum
```

and then filter the results with a selection based on the salary predicate. This is a good strategy even when an index on salary is present. To see why, notice that the number of records in the join based on

```
e1.salary > e2.salary
```

is approximately $(n^2)/2$, where n is the number of Employee records. If there are 100,000 Employee records, this is 5,000,000,000 records. The time to assemble such a quantity of records is likely to be far longer (by a factor of 50 or more) than the time a reasonable database management system would take to perform the equality join, even without indexes. (The reasonable system would likely either create a temporary index or sort both relations and then perform a merge join of the results.)

3.3 Key Types

A *key* of an index is a set or sequence of attributes. Index searches use values on those attributes to access records of an accompanying table. Many records in the accompanying table may have the same key value.[1]

From the viewpoint of the index designer, there are two kinds of keys with respect to a table *T*.

1. A *sequential key* is one whose value is monotonic with insertion order. That is, the last record inserted into *T*, has the highest value of the key. For example, if a timestamp marking the time of insertion of a record were a key, then it would be a sequential key. Another example would be a counter that is incremented with every record insert.
2. A *nonsequential key* is a key whose value is unrelated to the insertion order to the table. For example, if a Social Security number were a key of an Employee table, then it would be a nonsequential key because the last record inserted will only rarely have the highest Social Security number.

In certain cases, sequential keys will cause concurrency control problems as we will see.

[1] In normalization theory, as explained in the next chapter, the notion of key is different. In that theory, the key of a relation is a set of attributes with the property that no two distinct records have the same values on those attributes.

3.4 Data Structures

Most database systems support B-trees, and some support hash tables. (We'll review the definitions of these structures a bit later.) Some provide multidimensional data structures such as variants of quadtrees and R-trees. A few also support bit vectors and multitable join indexes. Main memory systems will support data structures having lower fanout such as T-trees, 2-3 trees, and binary search trees. We will start our discussion with the first two. We defer our discussion regarding bit vectors and multitable join indexes to our discussion of data warehouses, where they find their greatest use.

3.4.1 Structures Provided by Database Systems

The access pattern on an attribute of a table should guide the choice of data structure to access it. (Other factors such as whether the attribute is clustering or not also play an important role as we will see later in this chapter.) To understand what each data structure can do, you must understand a few details about its organization.

Levels, depth, and disk accesses

B-trees, hash structures, and multidimensional indexes can be viewed as trees. Some of the nodes in those trees are in random access memory and some are not. As a general rule, the root will often be in random access memory, and an arbitrary leaf will rarely be. In fact, the likelihood that a node will be in random access memory decreases the farther the node is from the root.

Because an access to disk secondary memory costs at least several milliseconds if it requires a seek (as these index accesses will), the performance of a data structure depends critically on the number of nodes in the average path from root to leaf. That number is known as the *number of levels* in a B-tree. (Note that if your disk device has a large cache, then the number of levels that are in random access memory increases accordingly.)

One technique that database management systems use to minimize the number of levels in a tree is to make each interior node have as many children as possible (several thousand for many B-tree implementations). The maximum number of children a node can have is called its *fanout* (Figure 3.2).

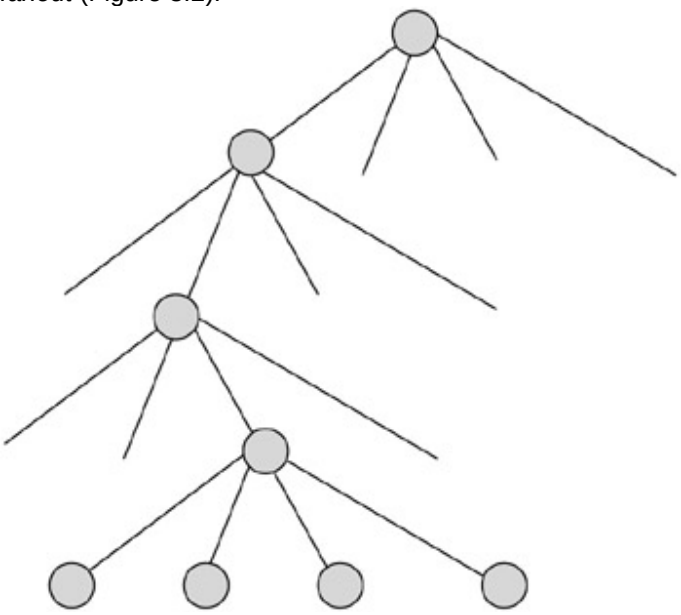


Figure 3.2: Levels and branching factor. This tree has five levels and a fanout of four.

When there is no room for an insert to fit in the node of a data structure, the node is said to *overflow*. In a B-tree, an overflowing node *n* splits into two nodes *n* and *n′* such that the distance between *n′* and the root is the same as the distance between *n* and the root. In some hash structures, the new page *n′* is pointed to from *n*, a technique known as *overflow chaining*. The chaining technique will cause the distance from the root to n′ to be one greater than the distance from the root to *n*.
Oracle uses *chained bucket hashing*: the number of hash keys (mathematically, the size of the range of the hash function) is fixed, there is one node for each hash key, and an overflow chain is defined for each node.

Thus, to compute the average number of nodes that must be traversed to reach a data node

- B-trees, count the number of levels
- for hash structures, compute the average number of overflows

Some systems extend the tree topology of their data structures by providing right links among the leaf nodes (and possibly the interior nodes as well). That is, each leaf node has a pointer to its right neighbor. Such a *link implementation* helps speed range queries in B-trees.

Specialized concurrency control methods can also use the links to remove concurrency control bottlenecks from data structures, as shown originally by Peter Lehman and S. B. Yao of the University of Maryland, and later elaborated by Yehoshua Sagiv of Jerusalem University, Betty Salzberg of Northeastern, and Vladimir Lanin and Shasha of New York University. C. Mohan of IBM Almaden research proposed practical algorithms to unify this style of concurrency control and recovery considerations.[2]

B-trees

A B-tree is a balanced tree whose leaves contain a sequence of key-pointer pairs. The keys are sorted by value[3](Figure 3.3).

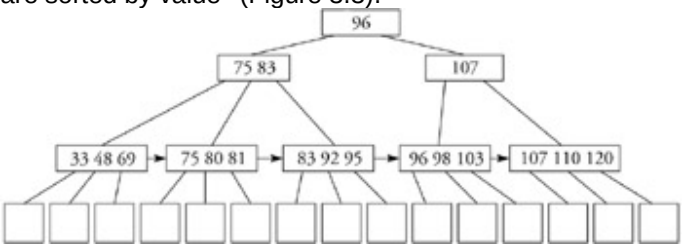


Figure 3.3: Example of B+ tree. Leaf nodes contain data entries (in this diagram, the data is represented by the box next to each key). All data entries are at the same distance from the root; that is the meaning of balance. Nonleaf nodes contain key-pointer pairs. There are actually *m* keys and m+1 pointers on each nonleaf node. Each pointer P_i, associated to a key K_i, points to a subtree in which all key values *k* lie between K_i and K_i+1 (P_0 points to a subtree in which all key values are less than K_0 and P_m points to a subtree in which all key values are greater than K_m). In most implementations, leaf nodes (and nonleaf nodes at the same level) are linked in a linked list to facilitate range queries and improve concurrency.

Evaluation B-trees are the best general data structure for database systems in that they seldom require reorganization, and they support many different query types well. If range queries (e.g., find all salaries between $70,000 and $80,000) or extremal queries (e.g., find the minimum salary) occur frequently, then B-trees are useful, whereas hash structures are not. If inserts are frequent and the B-tree key is nonsequential, then B-trees will give more consistent response times than hash structures because those structures may have overflow chains of unpredictable length. (Please note that we are discussing the data structure alone. Later, we will see that the data pages indexed by a clustering index, B-tree or not, may have overflow pages.)[4]

Because the fanout of a B-tree is the maximum number of key-pointer pairs that can fit on a node of the B-tree, the bigger the key is, the smaller the fanout. A smaller fanout may lead to more levels in the B-tree.

EXAMPLE: INFLUENCE OF KEY LENGTH ON FANOUT

Let us compute the number of levels deep a B-tree can be whose leaf pages contain 40 million key-pointer pairs.

If pointers are 6 bytes long and the keys are 4 bytes long, then approximately 400 key-pointer pairs can fit on a 4-kilobyte page. In this case, the number of levels (including the leaf pages) of a B-tree with 64 million key-pointer pairs would be 3. The root level would consist of one node (the root). The next level down would have 400 nodes. The leaf level would have 160,000 nodes, each with 400 key-pointer pairs giving a total of 64 million.

This assumes 100% utilization. If nodes are merged when they fall to half full, then their utilization holds at around 69%. In order to save restructuring time and increase the amount of concurrency, some systems never merge nodes. Studies by Ted Johnson and Shasha have shown that for B-trees with more inserts than deletes (both random), the utilization will be in the range from 65% to 69% even if nodes are never merged.[5] That would give more than 41 million key-pointer pairs in the leaves for a three-level B-tree.

By contrast, if keys are 94 bytes long, then only 40 can fit per page. In that case, even assuming 100% utilization,

- the root level would have one node.
- level 2 would have 40 nodes.
- level 3 would have 1600 nodes.
- level 4 would have 64,000 nodes.
- level 5 would have 2,560,000 nodes, each with at most 40 key-pointer pairs.

Thus, a five-level B-tree is needed to store all these key-pointer pairs.

In this example, a large key may cause the B-tree to be two levels deeper. If the data is static, then use your system's key compression option if it has one. This will reduce the space needed and save disk accesses at a relatively small processor cost (roughly 30% per record access). A typical compression technique is called *prefix compression*. Prefix compression will store in nonleaf nodes only that part of a key that is needed to distinguish the key from its neighbors in the node. For example, if there are three consecutive keys Smith, Smoot, and Smythe, then only Smi, Smo, and Smy need be held. Many systems use prefix compression.
Other forms of compression are more controversial. For example, one version of Oracle had a form of compression called *front compression* in which adjacent keys had their common leading portion factored out. So, the preceding three keys would be stored as Smith, (2)o, (2)y, where the 2 indicates that the first two characters are the same as the previous key. This saved space but caused two problems.

- It entailed substantial processor overhead for maintenance.
- It made item locking within B-tree nodes more subtle—locking (2)y would have implicitly required locking Smith as well.

Oracle continues to offer some form of compression (the COMPRESS clause in CREATE INDEX), but you should use this only in low update environments.

Hash structures

Hash structures are a method of storing key-value pairs based on a pseudo-randomizing function called a *hash function*. (Pseudo-randomizing here means that for most sets of keys, the hash function, when applied to these keys, will yield a distribution of values that is similar to what would be produced by randomly, with equal probability, assigning each key to some value in the range of the hash function.) The hash function can be thought of as the root of the structure. Given a key, the hash function returns a location that contains either a page address or a directory location that holds a page address. That page either contains the key and associated record or is the first page of an overflow chain leading to the record(s) containing the key. The hash function will return arbitrarily different locations on key values that are close but unequal (e.g., Smith and Smythe). So the records containing such close keys will likely be on different pages.

Evaluation Hash structures can answer point queries (e.g., find the employee with Social Security number 247–38–0294) in one disk access provided there are no overflow chains, making them the best data structures for that purpose (Figure 3.4). They are also good for multipoint queries especially if they are part of a clustering index (see Section 3.6).However, they are useless for range, prefix, or extremal queries.

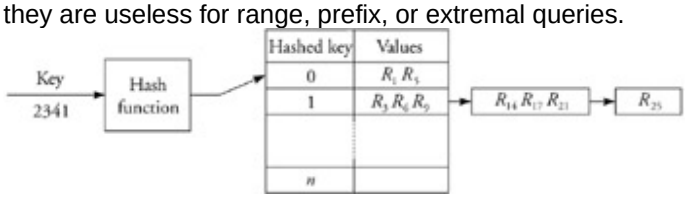


Figure 3.4: Hash structure with chain overflow.

If you find that your hash structure has a significant amount of overflow chaining, you should reorganize (drop and add or use a special reorganize function on) the hash structure following a large number of inserts. Avoiding overflow chaining may require you to underutilize the hash space. Some performance consultants suggest that a hash table should be no more than 50% full to ensure that a single disk access will be enough on the average.

So, hash structures have relatively poor space utilization, but since disk space is nearly free, this does not matter for most applications. Hash structures can be very sensitive to the choice of hash function, but the systems these days choose a hash function for you. If you choose your own, please make sure that you use prime numbers for modulus operations.

Because hash functions convert keys to locations or page identifiers, the size of a hash structure is not related to the size of a key. The only effect of key size on a hash structure is that the hash function takes slightly longer to execute on a long key. Given today's processor speeds, this is not a major factor.

Whereas hash structures with low space utilization perform well for point queries, they perform badly for queries that must scan all the data.

3.4.2 Data Structures for In-Memory Data

A *lookup table* is a table that is often read but never updated (or at least never updated during online transaction processing). Typical examples of lookup tables are conversion tables, for example, from two-letter codes to full spellings of states or countries.

Putting such small tables in the database system introduces unnecessary overhead, so many application designers store these tables as unordered arrays, linked lists, or sorted structures that are sequentially searched. (Basically, they use whatever their application libraries give them.)
You might, however, consider different data structures. We have already discussed hash structures and B-trees. A special case of a B-tree, known as a *2-3* tree, is a good choice to maintain sorted data within random access memory. Like a B-tree, all the leaves in a 2-3 tree are the same distance from the root. The difference is that the maximum fanout is 3 and the minimum fanout is 2. This makes for a deeper, less bushy tree than a standard B-tree.

A narrow deep tree reduces intranode search and increases the number of internode pointer traversals. In many situations, this reduces the number of instructions required for each search. In 1986, Mike Carey and his colleagues at the University of Wisconsin at Madison defined the T-tree to exploit this trade-off.[6] The TimesTen main memory database system uses the T-tree.

Because CPU speed has increased much faster than memory access speed over the last 20 years, memory accesses (especially nonconsecutive memory accesses) can be hundreds of times slower than an instruction cycle. Recent work has shown that in-memory data structures should not be optimized for reduced CPU usage but for reduced processor cache misses. Indeed, the gap between cache accesses and main memory accesses is continuously increasing. Recently, Kenneth Ross and Jun Rao have proposed a cache-sensitive B+ tree where each node has the size of a cache line and where pointers are eliminated when possible.[7] By removing pointers, more keys can be placed in each node. As a result, the number of key comparisons per cache line is maximized. Cache-sensitive structures may be included in future releases of database systems.
A structure that is great for looking up strings (or their prefixes) in memory is the *trie*. If you want to look up the infixes of strings, then use the related structures if available: the *suffix tree* and the *string B-tree.*
Another structure for designers who like the simplicity of linked lists is the *frequency-ordered linked list*. The idea of this structure is that the first entries of the linked list are the most frequently accessed and the last entries are the least frequently accessed. A background algorithm keeps track of accesses and rearranges the list appropriately during quiet periods. This works well for lists of frequently asked questions, where the frequency determines the position on the list.
If your application concerns geography or geometry, then you should know about *multidimensional data structures*. Typical examples that can be found in the literature are quadtrees, R-trees, R+ trees, grid structures, and hB-trees.[8] Like composite indexes (discussed later in this chapter), multidimensional data structures apply to multiple attributes. Unlike composite indexes, multidimensional data structures apply symmetrically to their attributes, which often makes the processing of geographical queries more efficient.

A typical such query finds all cities having at least 10,000 inhabitants within a certain latitude and longitude range—that is, symmetrically along both dimensions.

```
SELECT name
FROM City
WHERE population >= 10000
AND latitude >= 22
AND latitude < 25
AND longitude >= 5
AND longitude >= 15
```

So far, commercial geographical information systems have mainly used R-trees and quadtrees to answer such queries. If you have to answer such queries, make sure your database system has such structures built in or as an extension.

[2]C. Mohan, *ARIES—KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions on B-Tree Indexes*. 16th Very Large Data Bases Conference, San Francisco: Morgan Kaufmann, 1990.

Gerhard Weikum, Gottfried Vossen, *Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery*. Morgan Kaufmann, May 2001.

[3]What we call B-trees here are technically called B+ trees. Strictly speaking, a B-tree has pointers to data pages (or records) in its internal nodes as well as in its leaf nodes; this turns out to be a bad strategy since it yields low fanou t. Therefore, all major database vendors use B+ trees (which have pointers to data only at the leaves).

[4]In systems that use page-level locking, B-trees work poorly if they are based on sequential keys and there are many inserts because all inserts will access the last page, forming a concurrency control bottleneck.

[5]T. Johnson and D. Shasha, "Utilization of B-trees with Inserts, Deletes, and Modifies." *8th ACM SIGACT-SIGMOD Conference on Principles of Database Systems*, 235–246, March 189. If inserts enter in sorted order (i.e., not at all random), then utilization can fall to 50% in some systems. Check your system's tuning guide.

[6]Tobin J. Lehman and Michael J. Carey, *A Study of Index Structures for Main Memory Database Management Systems*. VLDB 1986.

[7]Jun Rao and Kenneth Ross, *Making B+-Trees Cache Conscious in Main Memory.* ACM SIGMOD 2000.

[8]Volker Gaede and Oliver Günther, "Multidimensional Access Methods," *ACM Computing Surveys*, 30(2), 1998.

3.5 Sparse Versus Dense Indexes

The data structure portion of an index has pointers at its leaves to either data pages or data records.

- If there is at most one pointer from the data structure to each data page, then the index is said to be *sparse*.
- if there is one pointer to each record in the table, then the index is said to be *dense*.

Assuming that records are smaller than pages (the normal case), a sparse index will hold fewer keys than a dense one. In fact, the

Number of pointers in dense index = number of pointers in sparse index × number of records per page

If records are small compared to pages, then there will be many records per data page, and the data structure supporting a sparse index will usually have one level less than the data structure supporting a dense index.[9] This means one less disk access if the table is large. By contrast, if records are almost as large as pages, then a sparse index will rarely have better disk access properties than dense indexes.

A significant advantage of dense indexes is that they can support ("cover" is the word used sometimes) certain read queries within the data structure itself. For example, if there is a dense index on the keywords of a document retrieval system, you can count the number of keyword lists containing some term, for example, "mountain trail," without accessing the documents themselves. (Count information is useful for that application because queriers frequently restrict their query when they discover that their current query would retrieve too many documents.)

[9]In most systems, every pointer has an associated key. However, if many keys have the same value and the key is large, an important optimization for dense indexes is to store a given key value only once followed by a list of pointers. That optimization reduces the space needs of a dense index.

3.6 To Cluster or Not To Cluster

A *clustering index* (sometimes called a *primary* index) on an attribute (or set of attributes) *X* is an index that colocates records whose *X* values are "near" to one another (Figure 3.5).

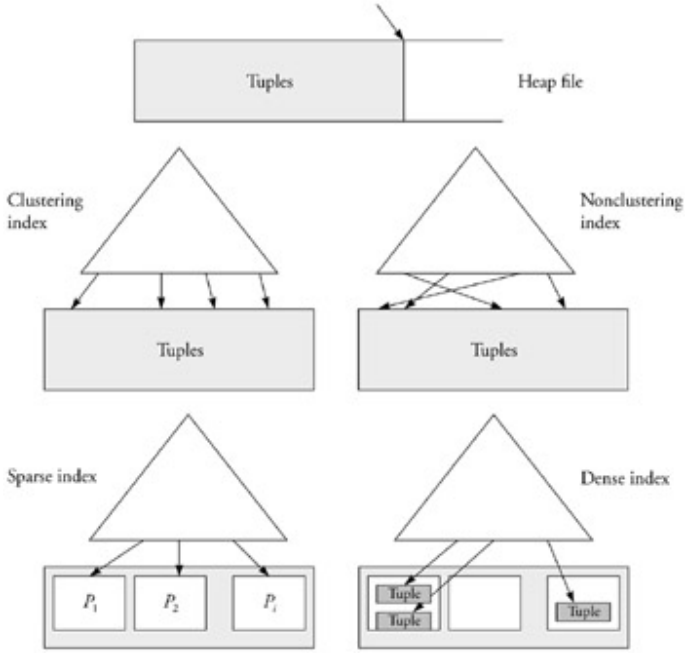


Figure 3.5: Data organization. This diagram represents various data organizations: a heap file (records are always inserted at the end of the data structure), a clustering index (records are placed on disk according to the leaf node that points to them), a nonclustering index (records are placed on disk independently of the index structure), a sparse index (leaf nodes point to pages), and a dense index (leaf nodes point to records). Note that a nonclustering index must be dense, whereas a clustering index might be sparse or dense.

Two records are colocated if they are close to one another on disk. What "near" means depends on the data structure. For B-trees and other sorted structures, two *X* values are near if they are close in their sort order. For example, 50 and 51 are near, as are Smith and Sneed. For hash structures, two *X* values are near only if they are identical. That is why clustered B-

tree structures are good for partial match, range, multipoint, point, and general join queries. By contrast, hash structures are useful only for point, multipoint, and equijoin queries.

Clustering indexes are sparse in some systems (e.g., SYBASE Adaptive Server or SQL Server) and dense in others (e.g., DB2 UDB). Using the sparse alternative can then reduce response time by a factor of two or more.

Because a clustering index implies a certain table organization and the table can be organized in only one way at a time, there can be at most one clustering index per table.
A *nonclustering index* (sometimes called a *secondary* index) is an index on an attribute (or set of attributes) *Y* that puts no constraint on the table organization. The table can be clustered according to some other attribute *X* or can be organized as a heap as we will discuss later. A nonclustering index is always dense—there is one leaf pointer per record. There can be many nonclustering indexes per table.

Just to make sure you understand the distinction between the two kinds of indexes, consider a classical library (the kind with books). A book may be clustered by access number, for example, Library of Congress number. The access number is the book's address in the library. Books with close access numbers will tend to be physically close to one another. In addition, there may be several nonclustering indexes, for example, one based on last name of author and another based on title. These are nonclustering because two books with the same author name may be physically far apart. (For example, the same author may write a book about database tuning and another one about a mathematical detective.)
A *heap* is the simplest table organization of all. Records are ordered according to their time of entry. That is, new inserts are added to the last page of the data structure. For this reason, inserting a record requires a single page access. That is the good news.

The bad news is that

- concurrent insert transactions may lock one another out unless your system spreads the insert points across the heap. If your system offers record locks, then short-term page locks—held only as long as it takes to obtain the record lock—may cause a bottleneck if the insert rate is high enough.[10]
- no index is clustering.

With this introduction, we are now in a position to compare clustering to nonclustering indexes.

3.6.1 Evaluation of Clustering Indexes

A clustering index offers the following benefits compared to a nonclustering one:

- If the clustering index is sparse, then it will store fewer pointers than a dense index. (Recall that nonclustering indexes cannot be sparse.) As we discussed earlier, this can save one disk access per record access if the records are small.
- A clustering index is good for multipoint queries, that is, equality accesses to nonunique fields. For example, a clustering index is useful for looking up names in a paper telephone book because all people with the same last name are on consecutive pages. By contrast, a nonclustering index on the first three digits of subscribers' phone numbers would be worse than useless for multipoint queries. For example, a query to find all subscribers in the 497 exchange might require an access to nearly every page (Figure 3.6).

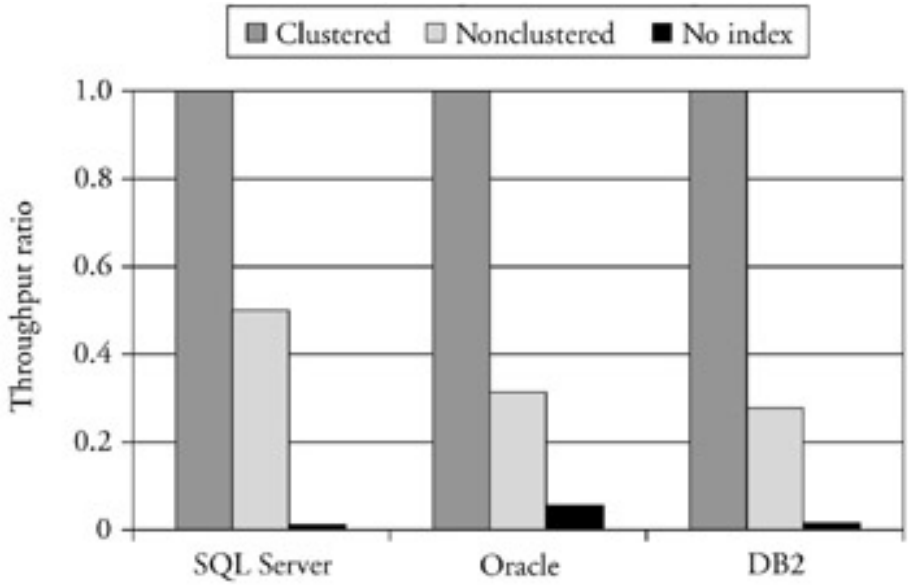


Figure 3.6: Clustering index. For all three systems, a clustering index is twice as fast as a nonclustering index for a multipoint query and orders of magnitude faster than a full table scan (no index). Each multipoint query returns 100 records out of the 1,000,000 that the relation contains. These experiments were performed on DB2 UDB V7.1, Oracle 8i and SQL Server 7 on Windows 2000.

For the same reason, a clustering index will help perform an equality join on an attribute with few distinct values. For example, consider the equality join query on first names:

```
SELECT Employee.ssnum, Student.course
FROM Employee, Student
WHERE Employee.firstname = Student.firstname
```

If the relation Employee has a clustering index on firstname, then for each Student record, all corresponding Employee records will be packed onto consecutive pages.
If the Employee and Student tables both have a clustering index on first-name based on a B-tree structure, then the database management system will often use a processing strategy called a *merge join*. Such a strategy reads both relations in sorted order, thus minimizing the number of disk accesses required to perform the query. (Each page of each relation will be read in once.) This will also work if both relations have a clustering index on firstname based on a hash structure that uses the same hash function.

- A clustering index based on a B-tree structure can support range, prefix match, and ordering queries well. The white pages of a telephone book again provide a good example. All names that begin with ‘St’ will be on consecutive pages.

A clustering index based on these data structures can also eliminate the need to perform the sort in an ORDER BY query on the indexed attribute.
The main disadvantage of a clustering index is that its benefits can diminish if there are a large number of overflow data pages (Figures 3.7, 3.8, 3.9, and 3.10). The reason is that accessing such pages will usually entail a disk seek. Overflow pages can result from two kinds of updates.

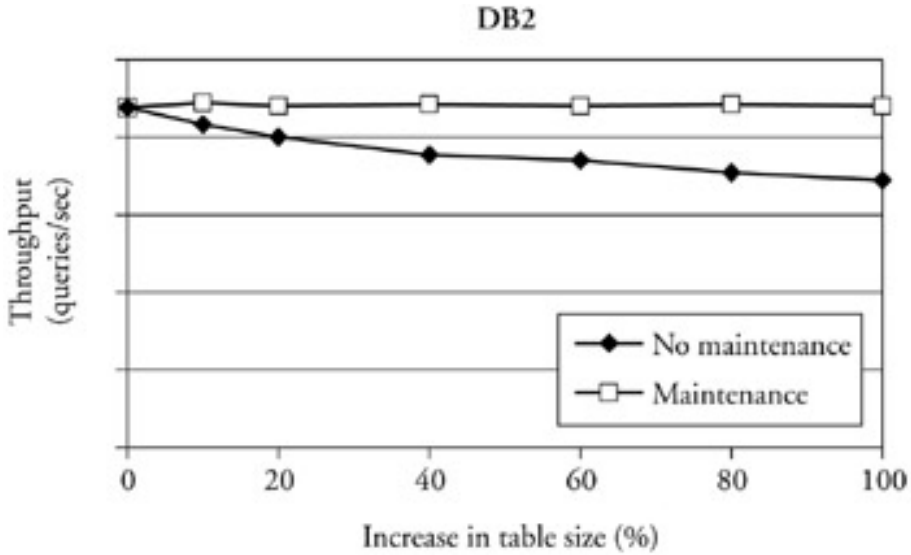


Figure 3.7: Index maintenance after insertions—DB2. The performance of the clustered index degrades with insertions. Once the index is full, additional records are simply appended to the relation. Each access is thus composed of a traversal of the clustering index followed by a scan of the additional records. In this experiment, a batch of 100 multipoint queries is asked. After each table reorganization, the index regains its original performance. This experiment was performed using DB2 UDB V7.1 on Windows 2000.

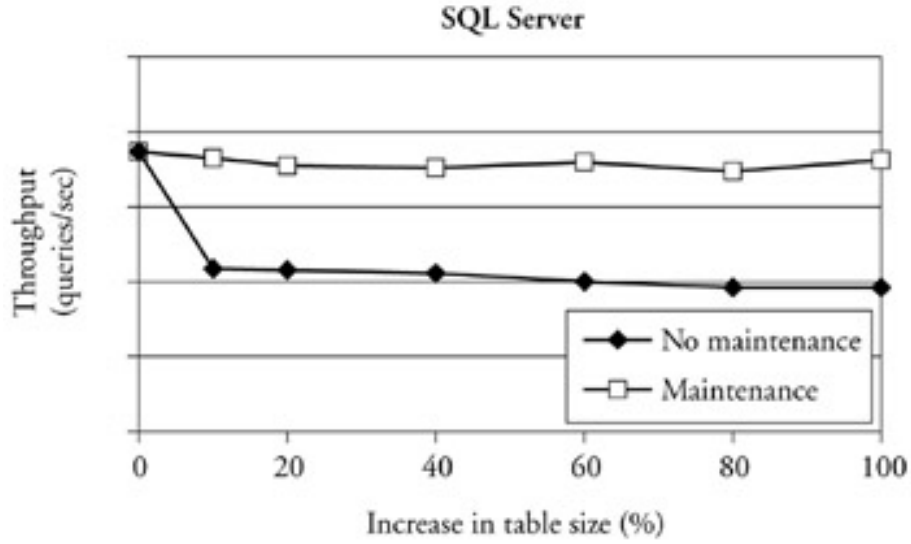


Figure 3.8: Index maintenance after insertions—SQL Server. The performance of the clustered index degrades fast with insertions. Once the index is full, pages are split to accommodate new records in the index structure. For the multipoint query we are running in this experiment, the page split results in extra I/O for each of the 100 queries in our batch. After dropping and re-creating the index, performances are back to what they were before the insertions. This experiment was performed on SQL Server 7 on Windows 2000.

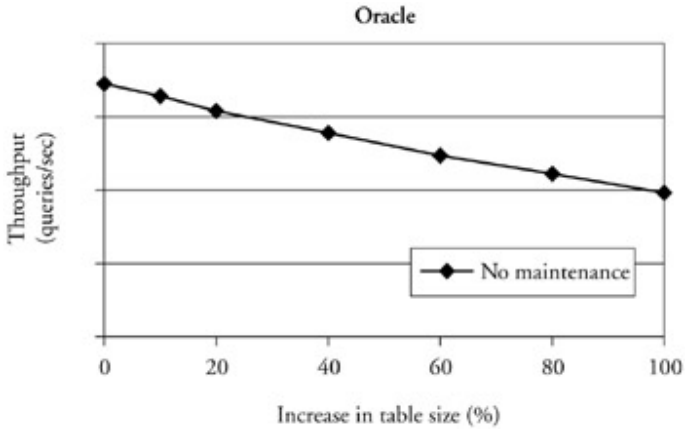


Figure 3.9: Index maintenance after insertions—Oracle. In Oracle, the notion of clustering and indexing are orthogonal. All indexes are nonclustering (except for index-organized tables whose application is restricted to unique indexes on a primary key). In the general case, a clustering index can be approximated by an index defined on a clustered table. There is, however, no automatic physical reorganization of the clustered table when the index is reorganized. The only way to perform maintenance is to export and reimport the table. This experiment was performed on Oracle 8i EE on Windows 2000.

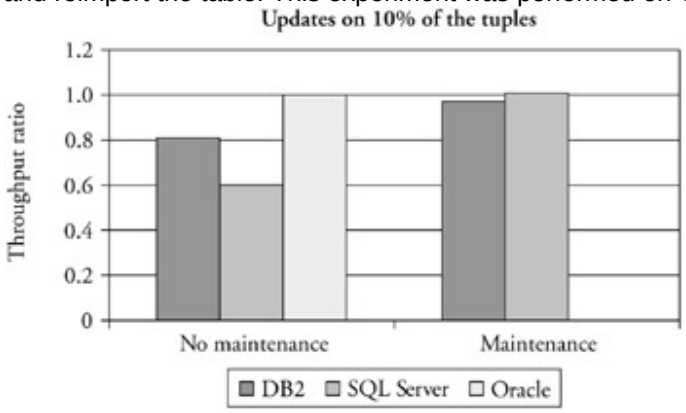


Figure 3.10: Index maintenance after 10% updates. The updates do not concern the key attributes. For DB2 and SQL Server, updates introduce a penalty comparable to the one caused by insertions (see experiments). In Oracle, updates just add data to the existing clusters (by default one page is reserved for each cluster value). In this case, there is no overflow. Consequently, Oracle's index performance is not affected by the updates. These experiments were performed on DB2 UDB V7.1, Oracle 8i, and SQL Server 7 on Windows 2000.

Inserts may cause data pages to overflow.

- Record replacements that increase the size of a record (e.g., the replacement of a NULL value by a long string) or that change the indexed key value will also cause overflows of the data page.

When there are a large number of overflow pages in a clustering index, consider invoking a utility to eliminate them or drop the index and then re-create it.

Redundant tables

Since there can be at most one clustering index per table, certain applications may consider establishing a redundant second table. Once again, you need to look no farther than your telephone book to see an illustration of this idea. The white pages are clustered by name. The

yellow pages contain only a subset of the entries in the white pages, but are clustered by category of goods and services offered (computer dealer, acupuncturist, ski shop, and so on). The two books both contain address and telephone information—hence, they are partially redundant. Adding redundancy works well when there are few updates.

3.6.2 Nonclustering Indexes

Because a nonclustering index on a table imposes no constraint on the table organization, there can be several nonclustering indexes on a given table

1. A nonclustering index can eliminate the need to access the underlying table through covering. For example, suppose there is a nonclustering index on attributes *A*, *B*, and *C* (in that order) of *R*. Then the following query can be answered completely within the index, that is, without accessing the data pages.

```
SELECT B, C
FROM R
WHERE A = 5
```

If your system takes advantage of this possibility (as do most), nonclustering indexes will give better performance than sparse clustering ones (though equal performance to dense clustering ones). Of course, updates would need to access the data pages of the *R* table (Figure 3.11).

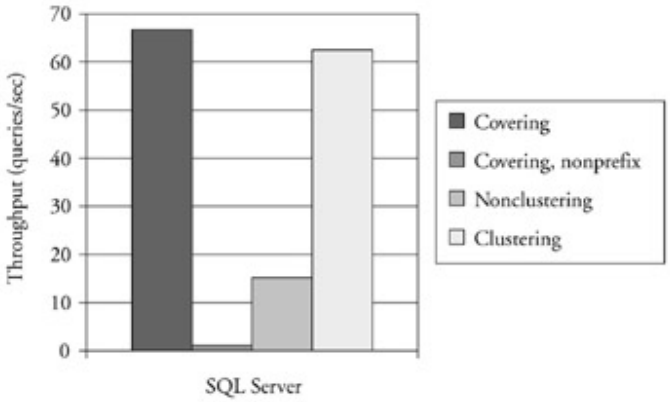


Figure 3.11: Covering index. This experiment illustrates that a covering index can be as good as or even better than a clustering index as long as the prefix match query that is asked matches the order in which the attributes have been declared. If it is not the case, then the composite index does not avoid a full table scan on the underlying relation. A covering index is also significantly faster than a nonclustering index that is not covering because it avoids access to the table records. This experiment was performed with SQL Server 7 on Windows 2000; that is, the clustering index is sparse.

5. For critical tables having few attributes, it may be worthwhile to create several nonclustering indexes so that every desired query on *A*, *B*, and *C* is covered. This works as well as redundant tables.
6. Suppose the query must touch the table *R* through a nonclustering index based on *A*. Let *NR* be the number of records retrieved and *NP* be the number of pages in *R*. If *NR* < *NP*, then approximately *NR* pages of *R* will be logically read. The reason is simple: it is likely that each record will be on a different page.

Thus, nonclustering indexes are best if they cover the query and are good if each query retrieves significantly fewer records than there are pages in the file (Figure 3.12). We use the word *significant* for the following reason: a table scan can often save time by reading many pages at a time, provided the table is stored contiguously on tracks. Therefore, even if the scan and the index both read all the pages of the table, the scan may complete by a factor of two to ten times faster.

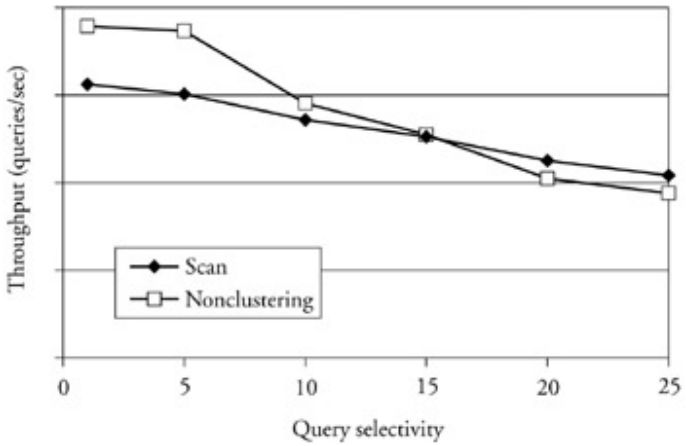


Figure 3.12: Nonclustering index. We use DB2 UDB V7.1 on Windows 2000 for this experiment. We use a range query and observe that the nonclustering index is advantageous when less than 15% of the records are selected. A scan performs better when the percentage of selected records is higher.

Consider the following two examples concerning a multipoint query.

- Suppose a table *T* has 50-byte records and pages are 4 kilobytes long. Suppose further that attribute *A* takes on 20 different values, which are evenly distributed among the records. Is a nonclustering index on *A* a help or a hindrance?
  Evaluation Because attribute *A* may contain 20 different values, each query will retrieve approximately 1/20 of the records. Because each page contains 80 records, nearly every page will have a record for nearly every value of *A*. So, using the index will give worse performance than scanning the table.
- Consider the same situation except each record is 2 kilobytes long.

  Evaluation In this case, a query on the nonclustering index will touch only every tenth page on the average, so the index will help at least a little.

We can draw three lessons from these examples.

1. A nonclustering index serves you best if it can help you avoid touching a data page altogether. This is possible for certain selection, count, and join queries that depend only on the key attributes of the nonclustering index.
2. A nonclustering index is always useful for point queries (recall that these are equality selections that return one record).
3. For multipoint queries, a nonclustering index on attribute *A* may or may not help. A good rule of thumb is to use the nonclustering index whenever the following holds:

4.

5. *number of distinct key index values accessed* < *c* × *number of records per page*
   where *c* is the number of pages that can be prefetched in one disk read.
   This inequality would imply that the use of the nonclustering index would entail fewer disk accesses (where a disk access fetches *c* pages) than scanning all the pages of the relation.

3.6.3 Composite Indexes

A *composite index* (called a concatenated index in some systems) is an index based on more than one attribute. A composite index may be clustering or nonclustering. For example, consider a relation

Person(ssnum, lastname, firstname, age, telnumber, …)

You might specify a composite index on (lastname, firstname) for Person. The white pages of telephone books are organized in this fashion. Thus, a data structure supporting a dense composite index on attributes (*A, B, C,...*) will store pointers to all records with a given *A* value together; within that collection, it will store pointers to all records with a given *B* value together; within that subcollection, it will store pointers to all records with a given *C* value together; and so on.

Composite indexes offer the following benefits compared with single-attribute (i.e., "normal") indexes:

- As mentioned, a dense composite index can sometimes answer a query completely, for example, how many people are there with last name ‘Smith’ and first name ‘John’?
- A query on all attributes of a composite index will likely return far fewer records than a query on only some of those attributes.
- A composite index is an efficient way to support the uniqueness of multiple attributes. For example, suppose that the relation (supplier, part, quantity) records the quantity of a particular part on order from a particular supplier. There may be many records with the same supplier identifier and many records with the same part identifier, but there should be only one record with a given supplier-part pair. This can be supported efficiently by establishing a composite index on (supplier, part) in conjunction with an SQL UNIQUE option.
- Oracle 9i introduces index skip scan to the Oracle product line. It is a form of scan that takes advantage of a composite index to avoid scanning the underlying table, even when a nonprefix of the index key is used. Consider a composite index defined on attributes (*A, B*) and a query with a condition on attribute *B*. The index skip scan searches the composite index to find all the *A* values and looks for those *AB* combinations that satisfy the condition on attribute *B*. This way a composite index can be used not only for prefix match queries but also (though at slower speed) for queries on other attributes.
- A composite index can support certain kinds of geographical queries. For example, suppose we look at the relation City, having to do with cities in the Southern Hemisphere.

City(name, latitude, longitude, population)

If we use a clustering composite index (latitude, longitude), then the query

```
SELECT name
FROM City
WHERE population >= 10000
AND latitude = 22
AND longitude >= 5
AND longitude <= 15
```

will execute quickly because all such cities will be packed as closely together as possible. By contrast, the following similar query will derive much less benefit from the composite clustering index:

```
SELECT name
FROM City
WHERE population >= 10000
AND longitude = 22
AND latitude >= 5
AND latitude <= 15
```

The reason is that all cities at latitude 5 will be packed together no matter what their longitude value is. As a result, a search will access the entire fraction of the database of cities whose latitudes fall within the range between 5 and 15 degrees south. So a composite index is not as good as a multidimensional one for general spatial queries.
When designing a composite index, you must specify the order of the attributes in the index. As illustrated by the last example about latitude and longitude, you should put attribute *A* before attribute *B* if your queries tend to put more constraints on *A* than on *B*. (So, a composite index on latitude, longitude will perform well on queries that specify a single latitude and a range of longitudes but not on queries that specify a single longitude and a range of latitudes.)

There are two main disadvantages to composite indexes:

- They tend to have a large key size. As we saw in the description of B-trees earlier, this can cause B-tree structures to be very large and to contain many levels unless some form of compression is used. Implementing a composite index as a hash structure solves the size problem but doesn't support prefix match or range queries.
- Because a composite index encompasses several attributes, an update to any of its attributes will cause the index to be modified.

[10]Steve Adams, *Oracle 8i Internal Services for Waits, Latches, Locks and Memory*. O'Reilly, 1999. This book describes the wait mechanisms in Oracle.

3.7 Joins, Foreign Key Constraints, and Indexes

Your database allows you to join two tables *R* and *S*. This can be done cleverly or badly. One way is to take each row of *R* and then search through the rows of *S* for matches. Such an approach is called a naive nested loop join. If there is no index on *S*, then the time required is proportional to the size of *R* times the size of *S* (|*R*| × |*S*|). This is bad because even modest million row tables require a trillion comparisons. If there is an index on the join attribute of *S*, then the work done for each row of *R* entails an index lookup for the appropriate rows of *S*, giving time proportional to |*R*| × log(|*S*|). So, an indexed nested loop is far better than a naive nested loop.
As for selections, an index may cover a join condition, in which case the index helps even more. For example, an index on *S.B* would cover the join condition in this case

```
SELECT R.B, R.D
FROM R, S
WHERE R.A = S.B
```

but not in this one

```
SELECT R.C, R.D, S.E
FROM R, S
WHERE R.A = S.B
```

Another approach to equality joining that is used when neither *R* nor *S* has an index is to hash both tables based on the same hash function on the join attributes and then join the corresponding buckets. So, if the join were *R.A* = *S.B*, such a hash join would reorganize *R* into buckets based on H(*R.A*) and *S* based on H(*S.B*) and then would join the first bucket of *R* with the first bucket of *S*, the second bucket of *R* with the second bucket of *S*, and so on. If *A* is a key or nearly a key of *R* (i.e., there are almost as many distinct values of *R.A* as there are rows in *R*) or similarly *B* is a key or nearly a key for *S*, then the time to do this operation is proportional to the sum of |*R*| and |*S*| (|*R*| + |*S*|). By default SQL Server and DB2 UDB use a hash join when no index is present. As Figure 3.13 and 3.14 show, the hash join can be faster than an indexed nested loop join in certain circumstances.

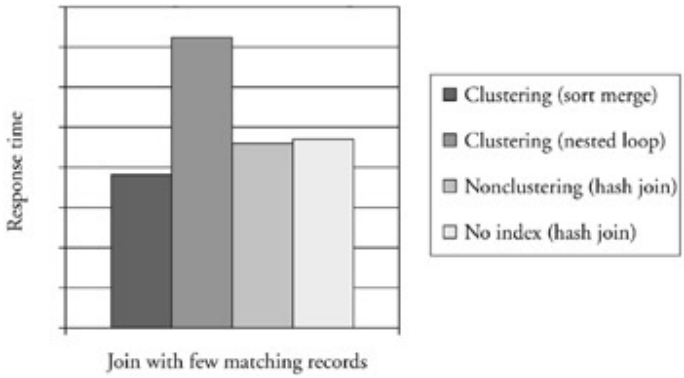


Figure 3.13: Join with few matching records. A hash join with no index is faster than an indexed nested loop join relying on a clustering index on the joining attribute because the clustering index is sparse. The nonclustering index is ignored: a hash join is used. This experiment was performed using SQL Server 7 on Windows 2000.

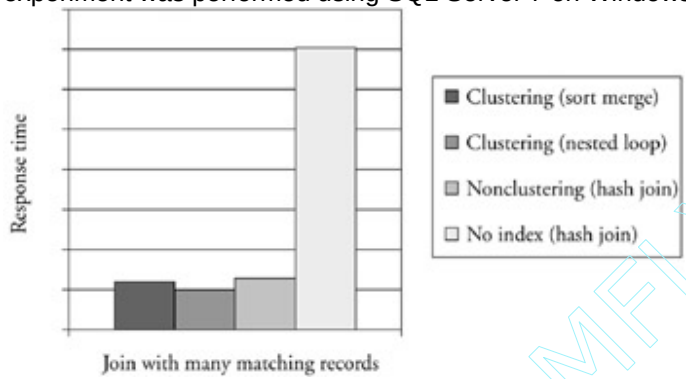


Figure 3.14: Join with many matching records. A hash join with no index performs worse than the other methods because setting up the buckets requires a lot of memory and disk reorganization. This experiment was performed using SQL Server 7 on Windows 2000.

So, the basic lessons for an equijoin (for concreteness, let the predicate be *R.A* = *S.B*) are as follows:

1. An indexed nested loop based on an index on *S.B* works better than a hash join if the number of distinct values of *S.B* is almost equal to the number of rows of *S*.[11] This is the common case because most joins are foreign key joins.
2. The same holds regardless of the number of distinct values of *S.B* if the index covers the join because then the only accesses to *S* data occur within the index.
3. The same holds regardless of the number of distinct values of *S.B* if *S* is clustered based on *B* because all *S* rows having equal *B* values will be colocated.
4. Otherwise, the hash join may be better.

There are two other cases where an index may be particularly helpful.

1. In the case of a nonequijoin (e.g., *R.A* > *S.B*), an index (using an ordered data structure such as a B-tree) on the join attribute avoids a full table scan in the nested loop.
2. To support a foreign key constraint such as that *R.A* must be a subset of *S.B* (e.g., *R.A* may hold the supplier identifier in a supplier-part table and *S* may be the supplier table with id field *B*). In this example, attribute *A* is said to be a foreign key in *R* and *B* is a primary key in *S*. For such a constraint, an index on *S.B* will speed up insertions on table *R*. The system generates the following nested loop: for every record inserted in *R*, check the foreign key constraint on *S*. Similarly, an index on *R.A* will speed up deletions in *S*. The system generates another nested loop: for every record deleted from *S*, check in *R* that there is no foreign key referencing the value being deleted.

[11]It also helps if the number of *R* rows having each *R.A* value is approximately the same. In the extreme case where there are only a few *R.A* values, the indexed nested loop join may not work at all well unless the database management system caches the rows of *S* that join with each *R.A* value. Such nonuniformities can come up. For example, if *R.A* represents bond interest rates, values will cluster around the prime rate.

3.8 Avoid Indexes on Small Tables

Indexes on small tables can do more harm than good. Many system manuals will tell you not to use an index on a table containing fewer than, say, 200 records. However, you should know that this number depends on the size of the records compared with the size of the index key.

- An index search will require reads of at least one and possibly two index pages and one data page. By contrast, if the entire relation is held on a single track on disk, reading it may require only a single physical read (provided you set your prefetching parameter to allow the system to read an entire track at once). An index may hurt performance in this case.
- If each record occupies an entire page, on the other hand, then 200 records may require 200 disk accesses or more. In this case, an index is worthwhile for a point query because it will entail only two or three disk accesses.
- If many inserts execute on a table with a small index, then the index itself may become a concurrency control bottleneck. (Lock conflicts near the roots of index data structures can form a bottleneck. Such conflicts arise when many inserts apply to a small index.)

Systematically avoiding indexes on a small table can also be harmful. In the absence of an index, a small table might become a serialization bottleneck if transactions update a single record. Without an index, each transaction scans through many records before it locks the relevant record, thus reducing update concurrency. We illustrate this point with an experiment (Figure 3.15).

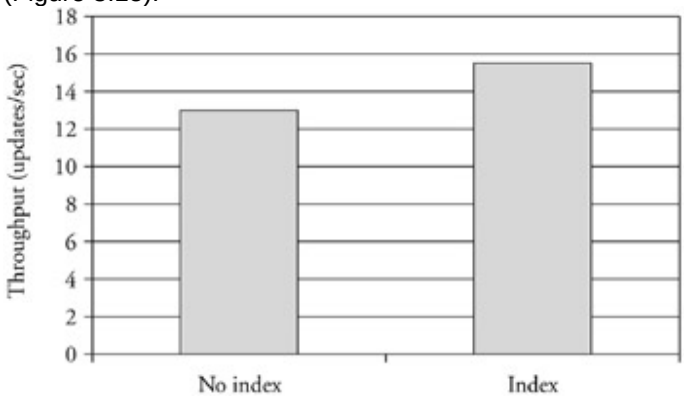


Figure 3.15: Indexes and updates. This graph shows the potential benefits for updates of creating an index on a small table (100 tuples). Two concurrent processes update this small table; each process works for 10 ms before it commits its update. When no index is used, the small table needs to be scanned when performing an update, and locks are requested for all the rows that are traversed by the scan operation. Concurrent updates are thus impossible. On the contrary, the presence of a clustered index on the attribute on which the update condition is expressed permits concurrent updates. This experiment was performed with SQL Server 2000 on Windows 2000.

3.9 Summary: Table Organization and Index Selection

Remember these basic rules and use the experiments to quantify choices.

1. Use a hash structure for equality queries. Use a B-tree if equality and nonequality queries may be used.
2. Use clustering (index-organized tables in Oracle) if your queries need all or most of the fields of each record returned, the records are too large for a composite index on all fields, and you may fetch multiple records per query.
3. Use a dense index to cover critical queries.
4. Don't use an index if the time you will lose during inserts or updates due to the indexes exceeds the time you save in queries (Figures 3.16 and 3.17 illustrate low and high overheads).

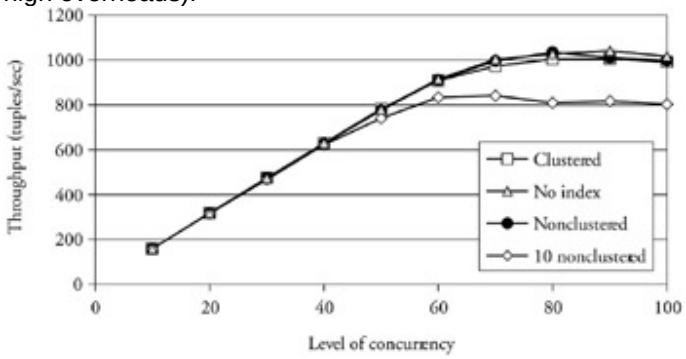


Figure 3.16: Low index overhead for insertion. Using SQL Server 7 on Windows 2000, we insert 100,000 records in the TPC-B table account(number, branchnum, balance). We observe that the cost of inserting data in a single clustering or a single nonclustering index is similar to the cost of inserting data in a heap; the overhead becomes significant when the number of nonclustering indexes increases, and the number of concurrent threads performing the insertions increases.

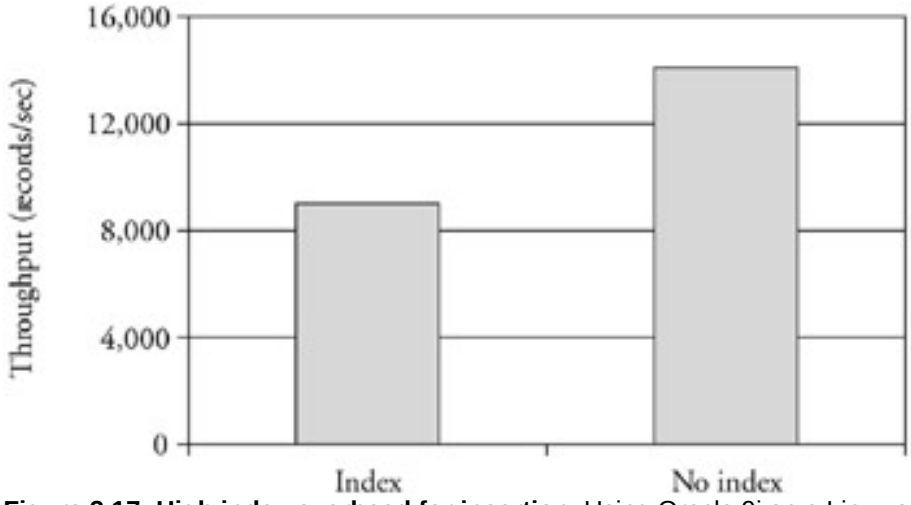


Figure 3.17: High index overhead for insertion. Using Oracle 9i on a Linux server, we insert 100,000 records in the table Order(ordernum, itemnum, quantity, purchaser, vendor). We measure throughput with or without a nonclustered index defined on the ordernum attribute. The presence of the index significantly affects performances.

Use key compression when the following conditions all hold:

- You are using a data structure other than a hash structure.
- Compressing the key will reduce the number of levels in your index.
- Your system is disk bound but has plenty of processor power.
- Updates to the data structure are relatively rare. (Updates to the underlying table that don't affect the index cause no difficulty, but insertions and deletions do.)

Virtually all commercial database management systems offer B-trees, usually with prefix compression. Systems differ in whether they offer other data structures and whether their primary (clustering) indexes are sparse or not. (Recall that secondary indexes must be dense.) Table 3.1 summarizes the facilities offered by some major relational products.

Table 3.1: Indexes offered by some relational vendors

SYSTEM	PRIMARY DATA STRUCTURE	SECONDARY DATA STRUCTURES
IBM DB2 UDB V7.1	B-tree (dense)	B-tree
Oracle 9i EE	B-tree (dense), hash (dense)	B-tree, bitmap, functions
SQL Server 7	B-tree (sparse)	B-tree
SYBASE	B-tree (sparse)	B-tree

What other systems call clustering indexes, Oracle calls index-organized tables. The index used is a B-tree. One limitation is that these indexes apply only to a key, so they support point but not multipoint queries. In order to support multipoint queries, you can use an Oracle cluster (which is a general facility that allows you to group records from one or more tables together). Oracle 8i and higher versions provide bitmap indexes. A bitmap index is a table where columns represent the possible values of the key; for each row in the relation there is one row in the bitmap table. Each entry in the bitmap table is a bit that is set to 1 if the key value in the given row corresponds to the column value and to 0 otherwise (on each row one bit is set to 1 and the rest is set to 0). We describe bitmap indexes further in Chapter 10. Figures 3.18, 3.19, 3.20, and 3.21 compare the index structures provided by Oracle 8i EE.

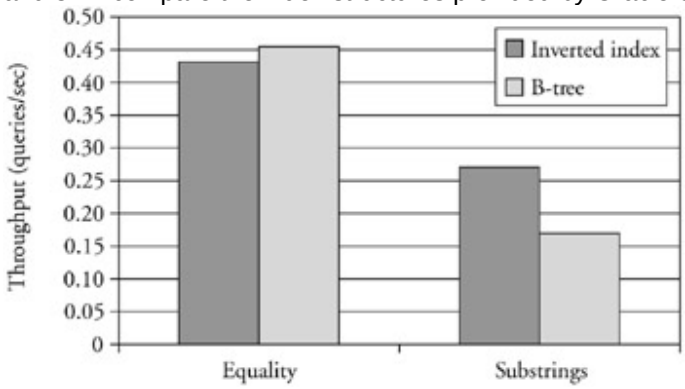


Figure 3.18: Text indexing. Oracle 8i with the Intermedia extension supports inverted indexes for substring. An inverted index defined on a text column is a sequence of (key, pointer) pairs where each word in the text is a key whose associated pointer refers to the record it has been extracted from. This experiment compares the performance of multipoint queries based on equality and substring predicates using an inverted index and a B+ tree. We defined these indexes on the comments attribute of the TPC-H lineitem relation; this attribute is of type varchar(44). Using a B+ tree the equality predicate is =, and the substring predicate is LIKE. When an inverted index is defined, both equality and substring predicates are constructed using

the CONTAINS function that searches through the inverted index. The graph shows that the performance of the inverted index and B+ tree are comparable for exact matches, while the inverted index performs slightly better on substring search. Note that inverted indexes can be defined on large objects(LOBs), whereas B+ trees cannot.

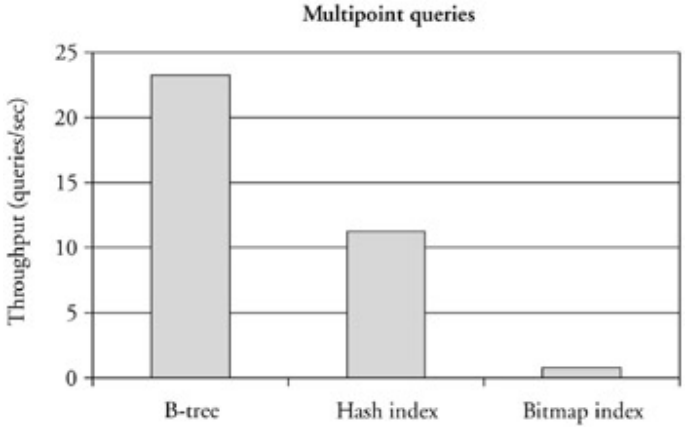


Figure 3.19: Comparison of B-tree, hash, and bitmap index multipoint queries. One hundred records are returned by each equality query. In the hash structure, these 100 records map to the same hash key, thus requiring an overflow chain. The clustered B-tree offers good performance because the records returned are on contiguous pages. The bitmap index does not perform well because it is necessary to traverse the entire bitmap to fetch just a few matching records. This experiment was performed using Oracle 8i EE on Windows 2000.

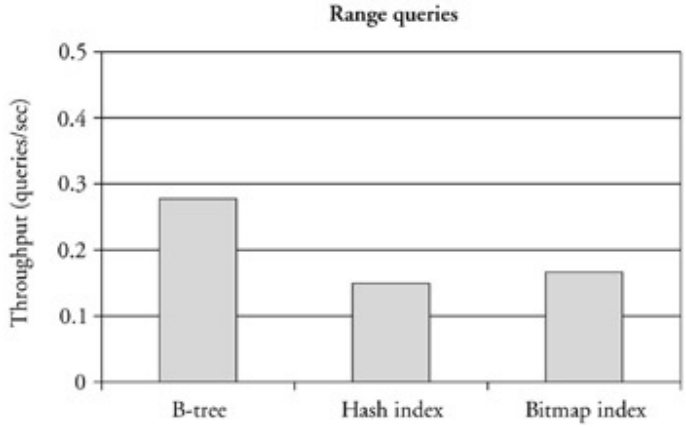


Figure 3.20: Comparison of B-tree, hash, and bitmap index—range queries. As expected, hash indexes don't help when evaluating range queries. This experiment was performed using Oracle 8i EE on Windows 2000.

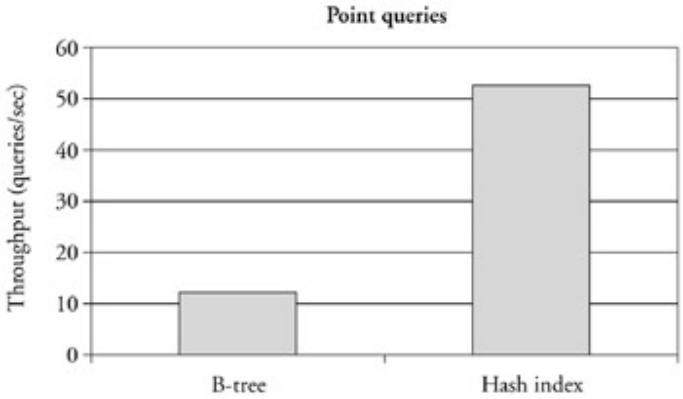


Figure 3.21: Comparison of B-tree and hash index—point queries. Hash index outperforms B-tree on point queries. This experiment was performed using Oracle 8i EE on Windows 2000.

INDEX TUNING WIZARD

[redacted]

SQL Server 7 and higher versions provide an index tuning wizard. The wizard takes as input a schema (with existing indexes) and the description of a workload (trace of SQL statements). It produces as output a set of recommendations: new indexes to be created and possibly existing indexes to be removed.

The wizard relies on the cost-evaluation module of the optimizer to perform its magic. First, the wizard evaluates the best index for each SQL statement in the workload. To do so, it enumerates relevant indexes on one attribute, then two attributes, and so on; computes the cost of the execution plans that would use these indexes (the maintenance overhead is considered for insert/delete statements); and picks the best one. In a second step, the wizard computes the cost of combining the indexes that have been picked up in the first phase (the cost of maintaining existing indexes is considered in addition to the query speed-up). The subset of indexes with the lowest combined cost is used for the final recommendation.

The wizard does a great job most of the time. We collected traces for some of the experiments presented in this chapter (clustering, covering, index on small table), and we let the wizard choose the appropriate index. In each case, the wizard chose the appropriate index: a covering index when a limited number of attributes are involved in a query, a clustering index for multipoint queries, or an index on a small table because updates are performed.

We can see two limitations in the use of the tuning wizard.

1. There is no priority among the different statements in the workload. The wizard computes the cost for the workload globally while we might favor a particular aspect of performance (query speed-up over maintenance cost). For instance, an index might not be recommended because the cost of maintaining it is greater than the performance speed-up. This is valid from a global point of view; however, we might have specific constraints on query response time that would justify the presence of the index while we do not have particular constraints on the insertions that are negatively affected by index maintenance.
2. It is possible that the wizard does not recommend the best index for a query because its maintenance cost is too high, while the second best index for that query would have provided a good speed-up or maintenance trade-off.

The wizard does a very good first-cut job. It is, however, important to keep a critical eye on its recommendations in particular when joins and insert/update/delete are part of the workload.

You can find further information on the tuning wizard in Surajit Chaudhuri and Vivek Narasayya's "An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server," *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

3.10 Distributing the Indexes of a Hot Table

A *hot table* is one that is accessed by many transactions concurrently. If several disks are available, then it may be worthwhile to partition accesses to that table across the disks either in a RAID configuration or otherwise. Depending on the access patterns, you may choose one of two organizations.

- Insert- or delete-intensive applications should move their nonclustering indexes to a disk other than the one containing the table data (Figure 3.22). The reason is that all indexes will have to be updated when an insert takes place. This partitioning balances the load between the disks having the nonclustering indexes and the disks having the clustering index and the table data. It is useful to keep the clustering index with the table data because most read accesses and all insert accesses will use that index.

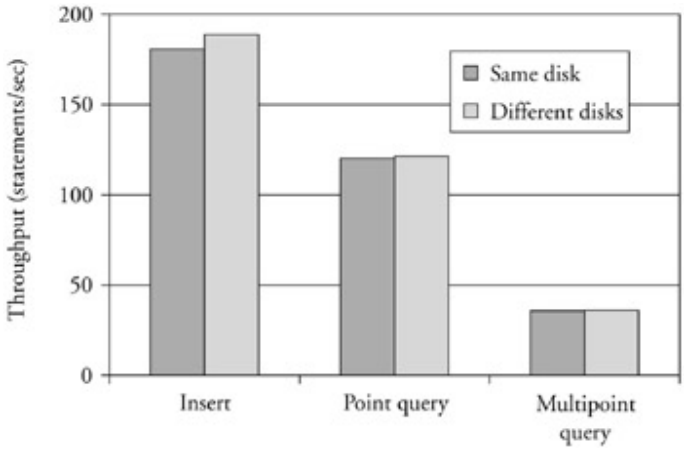


Figure 3.22: Distributing indexes. In this experiment, placing the index on a separate disk provides some advantage when inserting data. Point or multipoint queries do not always benefit from separating data and indexes. We used Oracle 8i on Windows 2000 for this experiment.

- Applications that perform many point queries and multipoint queries with few updates or with updates that modify only nonindexed attributes may consider spreading the table, the clustering index, and the nonclustering indexes over several available disks (possibly in a RAID-5 or similar configuration). This maximizes the number of queries that can execute concurrently (Figure 3.23).

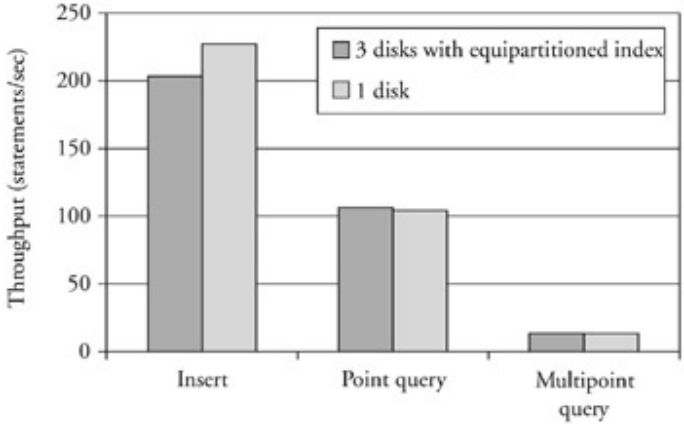


Figure 3.23: Partitioning indexes. Data is range partitioned on three disks; on each disk reside index and data. Insertions are slower compared to a single disk storage. Point

queries benefit slightly from this partitioning. This experiment was performed on Oracle 8i EE on Windows 2000.

3.11 General Care and Feeding of Indexes

Here are some maintenance tips on indexes.

- From time to time, your indexes need the data equivalent of a face lift. For example,
  - In a clustering index on attribute *A*, data page overflows have caused new pages to be placed on a different cylinder from old pages whose records have the same *A* values. (Chapter 7 will tell you how.)
  - A hash structure has long overflow chains.
  - Hash structures showing poor performance for point or multipoint queries may have insufficient space on disk. Recall that some systems suggest that hash spaces be no more than 50% utilized—check your tuning manual.
  - A B-tree has many empty nodes that have not been removed (this occurs in some systems such as DB2 that do not automatically free index nodes that are empty).

  To fix these problems, you can drop the index and rebuild it, perhaps allocating more space. Some systems allow you to modify an index in place. Oracle 9i allows dropping and re-creating an index while queries are running.
- Drop indexes when they hurt performance. If an application performs complicated queries at night and short insert transactions during the day, then it may be a good idea to drop the indexes when the updates occur and then rebuild them when the inserts have completed.
- Run the catalog statistics update package regularly to ensure that the optimizer has the information it needs to choose the proper indexes. When running those statistics, consider the options carefully. Some systems offer histogram options on the statistics update package. The idea is to give the density of values in a given range. For example, if most bonds have interest rates near the prime rate, but only a few have higher rates, then the system should be smart enough to use a nonclustering index for queries on the outlying interest rates but not for the popular interest rates. Choosing many histogram cells will help the query optimizer make the correct decision.
- Use a system utility that tells you how the system processes the query, that is, the system's *query plan*. The query plan will tell you which indexes are being used. Aside from out-of-date catalog statistics, there are several reasons why a query optimizer may not use indexes that might help.
  - The use of a string function. For example, the following query may not use an index on name in some systems:
  - SELECT *
  - FROM Employee
  - WHERE SUBSTR(name, 1, 1) = ‘G’
  - If a bind variable (a variable set by the programming language) has a different type than the attribute to which it is being compared, the index might not be used. So, if you compare a bind variable with a table attribute, ensure they are of the same type. This means integer with integer of same size, float with float of same size, and string with string of the same maximum size.
  - A comparison with NULL.
  - SELECT *
  - FROM Employee
  - WHERE salary IS NULL

- Some systems may not use indexes effectively for certain types of queries, such as nested subqueries, selection by negation, and queries that use ORs.

Bibliography

Surajit Chaudhuri and Vivek R. Narasayya. *An efficient cost-driven index selection tool for Microsoft SQL server.* In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, August 25–29, 1997, Athens, Greece, 146–155. Morgan Kaufmann, 1997.

Volker Gaede and Oliver Günther. *Multidimensional access methods. ACM Computing Surveys*, 30(2):170–231,1998 This survey reviews work on multidimensional index structures.

Theodore Johnson and Dennis Shasha. *Utilization of B-trees with inserts, deletes and modifies.* In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, March 29–31, 1989, Philadelphia, Pennsylvania, 235–246. ACM Press, 1989. This paper studies space utilization in B-trees and shows why free-at-empty gives better performance than marge-at-half for growing databases.

Tobin J. Lehman and Michael J. Carey. *A study of index structures for main memory database management systems.* In *VLDB'86 Twelfth International Conference on Very Large Data Bases,* August 25–28, 1986, Kyoto, Japan, 294–303. Morgan Kaufmann, 1986. This paper introduces T-trees.

C. Mohan. *Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes.* In *16th International Conference on Very Large Data Bases*, August 13–16, 1990, Brisbane, Queensland, Australia, 392–405. Morgan Kaufmann, 1990. Practical algorithm unifying concurrency control for B-trees with recovery considerations.

Oracle Corp. *Performance and scalability in DSS environment with Oracle 9i.* Oracle white paper. April 2001.

Jun Rao and Kenneth A. Ross. *Making B^+ -trees cache conscious in main memory.* In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, May 16–18, 2000, Dallas, Texas, vol. 29, 475–486. ACM, 2000. This paper introduces cache-sensitive B+ trees.

Gottfried Vossen and Gerhard Weikum. *Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001. An encyclopedic treatment of theory and some practice of all research in transaction processing. From workflow to data structure concurrency control, this book has both breadth and depth.

Exercises

The exercises use the following tables:

- Employee(ssnum, name, dept, manager, salary)

- Student(ssnum, name, course, grade, stipend, written_evaluation)

EXERCISE 1

When the Student relation was created, a nonclustering index was created on name. However, the following query does not use that index:

```
SELECT *
FROM Student
WHERE name = 'Bayer'
```

Action. Perhaps the catalog statistics have not been updated recently. For that reason, the optimizer concludes that Student is small and hence the index should not be used. Try updating the statistics.

EXERCISE 2

You discover that the following important query is too slow. It is a simple query, and there is a nonclustering index on salary. You have tried to update the catalog statistics, but that did not help.

```
SELECT *
FROM Employee
WHERE salary/12 = 4000
```

Action. The index is not used because of the arithmetic expression. Reformulate the query to obtain a condition of the form attribute_value = constant. Alternatively, you can use a function-based index if available.

EXERCISE 3

Your customer eliminates the arithmetic expression, yielding the following query:

```
SELECT *
FROM Employee
WHERE salary = 48000
```

However, the system uses the index without improving performance.
Action. The index is not a clustering index. Many employees happen to have a salary of $48,000. Using the index causes many random page reads, nearly one for each employee. The index may be useful for other salaries that are less common, however.

EXERCISE 4

Your system has a 2-kilobyte size page. The Student table records are very long (about 1 kilobyte) because of the length of the written_evaluation field. There is a clustering index on Social Security number, but the table suffers overflow chaining when new written_evaluation data is added.
Action. The clustering index doesn't help much compared to a nonclustering one on Social Security number because of the large record size. The nonclustering index will be easier to maintain.

EXERCISE 5

Suppose there are 30 Employee records per page. Each employee belongs to one of 50 departments. Should you put a nonclustering index on department?
Action. If such an index were used, performance would be worse unless the index covers the query. The reason is that approximately 3/5 (=30/50) of the pages would have Employee

records from any given department. Using the index, the database system would access 3/5 of the pages in random order. A table scan would likely be faster. As Figure 3.12 showed, even when only 10% of the records are selected by a nonclustering index, that index may not be worthwhile.

EXERCISE 6

Suppose there are 30 Employee records per page. However, in this case, there are 5000 departments. Should you put a nonclustering index on department to support multipoint queries on departments?
Action. Each page has only a 30/5000 chance of having an Employee record. Using the index would be worthwhile.

EXERCISE 7

Auditors take a copy of the Employee file to which they wish to apply a statistical analysis. They allow no updates but want to support the following accesses:

1. Count all the employees that have a certain salary (frequent).
2. Find the employees that have the maximum (or minimum) salary within a particular department (frequent).
3. Find the employee with a certain Social Security number (rare).

Initially, there is no index.
Action. A clustering index on salary would give less benefit than a nonclustering index because the first query can be answered solely based on the nonclustering index on salary. That is, a dense index will cover the query. By contrast, the same query using a clustering index may have to access the Employee records if the clustering index is sparse. A nonclustering composite index on (dept, salary) using a B-tree would be useful to answer the second style of query. A sparse clustering index on Social Security number will help if the Employee records are small because a sparse index may be a level shorter than a dense one.

EXERCISE 8

Suppose that the student stipends correspond to monthly salaries, whereas the employee salaries are yearly. To find out which employees are paid as much as which students, we have two choices.

```
SELECT *
FROM Employee, Student
WHERE salary = 12*stipend
```

or

```
SELECT *
FROM Employee, Student
WHERE salary/12 = stipend
```

Which is better?
Action. Many systems will use an index on salary but not on stipend for the first query, whereas they will use an index on stipend but not on salary for the second query. If there is an index on only one of these, it should be used.

If there are indexes on both, then the situation is a bit complicated.

- If the index on the larger table is clustering, then arrange your query to use it. This will avoid reading the larger relation.

- Even if the index on the larger table is nonclustering, but the larger table has more pages than the smaller table has records, then arrange your query to use the index on the larger table. Again, this will avoid reading all the pages of the larger relation. This will be a common case.
- By contrast, suppose the index on the larger table is nonclustering and you suspect that every page of the larger table will have at least one record that matches some record of the smaller table. In that case, using the index on the larger table may cause the query to access the same page of the larger table several times. So, it may be worthwhile to arrange your query to use the index on the smaller table.

The ideal would be that the two fields be placed on the same basis, that is, both monthly or both yearly. Then the optimizer would make the choice.

EXERCISE 9

A purchasing department maintains the relation Onorder(supplier, part, quantity, price). The department makes the following queries to Onorder:

1. Add a record, specifying all fields (very frequent).
2. Delete a record, specifying supplier and part (very frequent).
3. Find the total quantity of a given part on order (frequent).
4. Find the total value of the orders to a given supplier (rare).

Action. If there is time to eliminate overflow pages at night, then a clustering composite index on (part, supplier) would work well. Part should come first in the composite index because then a prefix match query on part will answer query 3. (Because of this prefix match query, the data structure in the clustering composite index should be a B-tree rather than a hash structure.)

EXERCISE 10

Consider a variant of Exercise 9 in which the queries access an archival read-only Onorder relation.
Action. Once again, a clustering composite index on (part, supplier) would work well. To support query 4, a nonclustering index on supplier alone would be helpful though a separate table holding the aggregated totals per supplier would be even better.

EXERCISE 11

A table has a clustering B-tree index on Social Security number and performs simple retrievals and updates of records based on Social Security number. The performance is still not good enough. What should be done?
Action. B-trees are not as fast as hash structures for point queries.

EXERCISE 12

Ellis Island is a small island south of Manhattan through which flowed some 17 million immigrants to the United States beteen the late 1800s and the mid-1900s. Immigration workers filled in some 200 fields on each immigrant, containing information such as last name (as interpreted by the immigration worker), first name, city of origin, ship taken, nationality, religion, arrival date, and so on. You are to design a database management system to allow the approximately 100 million descendants of these 17 million to retrieve the record of their ancestors.

To identify an immigrant, the querier must know the last name of the immigrant as well as some other information (the more the merrier). Most queriers will know the last name and either the first name or the year of arrival. What is a good structure?

Action. Once the records are loaded, this database is never updated. So, a B-tree structure having high space utilization is a good possibility. Because nearly all queriers know the last name of their ancestor and many know the first name as well, a clustering composite index on (last name, first name) would help. Indexes on other single fields would probably not help because they would not be selective enough. However, a composite index on (last name, year of arrival) might be helpful. Because there are no updates, the only cost to indexes is space. Indeed, one radical solution is to have many composite indexes, one for each likely combination of attributes.

EXERCISE 13

An airline manages 1000 flights a day. In their database, there is a table for each flight (called Flight) containing a flight identifier, a seat identifier, and a passenger name. There is also a master table (called Totals) that has the total number of passengers in each flight. Each reservation transaction updates a particular Flight table and the Totals table. They have a performance bottleneck whose symptom is high lock contention. They try to resolve this by breaking each reservation transaction into two: one that updates the appropriate Flight table and the other that updates the Totals table. That helps but not enough.
Action. The Totals table may be a serialization bottleneck for all transactions because every flight reservation transaction must update that table and the table is small, possibly only a few pages in size. Therefore, there is going to be contention on that table. Even if your system has record-level locking, an update to the Totals table might have to scan through many records to arrive at the one relevant to a given flight. This may also lead to lock contention. Consider reorganizing Totals to establish an index on that table in order to avoid the need to scan. Figure 3.15 illustrates this principle.

Notice that making Totals a view on Flight would not be useful because it would not be updatable (data derived from an aggregate never is).

Chapter 4: Tuning Relational Systems

4.1 Goal of Chapter

Nearly all the database systems sold today are relational—not bad for a data model that was dismissed as totally impractical when Ted Codd first introduced it in the early 1970s (shows you how much to trust critics). The relational model offers a simple, more or less portable, expressive language (usually SQL) with a multitude of efficient implementations.
Because of the spectrum of applications that relational systems cover, making them perform well requires a careful analysis of the application at hand. Helping you do that analysis is the goal of this chapter. The analysis will have implications for lower-level facilities, such as indexes and concurrency control. This chapter, however, discusses higher-level facilities (Figure 4.1). The discussion will concentrate on four topics.

- Table (relation) design—trade-offs among normalization, denormalization, clustering, aggregate materialization, and vertical partitioning. Trade-offs among different data types for record design.
- Query rewriting—using indexes appropriately, avoiding DISTINCTs and ORDER BYs, the appropriate use of temporary tables, and so on.
- Procedural extensions to relational algebra—embedded programming languages, stored procedures, and triggers.
- Connections—to conventional programming languages.

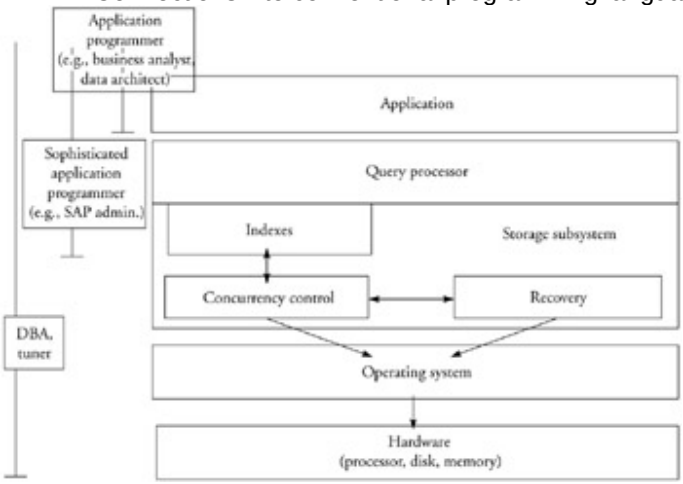


Figure 4.1: Database system architecture. Responsibility of people with different skills.

4.2 Table Schema and Normalization

One of the first steps in designing an application is to design the tables (or relations) where the data will be stored. Once your application is running, changing the table design may require that you change many of your application programs (views don't work for most updates). So, it is important to get the design right the first time.

Normalization is a rational guide to the design of database tables.

4.2.1 Preliminary Definitions

A *relation schema* is a relation name and a set of attribute names. If you think of a relation as a table name, then its schema is the table name plus the column headers of that table. A *relation instance* for a relation *R* is a set of records over the attributes in the schema for *R*. Informally, a relation instance is the set of records (or rows or tuples, if you wish) in the table.
For example, the table in Figure 4.2 is called Purchase and has attributes name, item, price, quantity, suppliername, and year. The records are the four bottom rows.

NAME	ITEM	PRICE	QUANTITY	SUPPLIERNAME	YEAR
Bolt	4325	15	60	Standard Part	2001
Washer	5925	13	60	Standard Part	2002
Screw	6324	17	54	Standard Part	2003
Nut	3724	15	80	Metal Part	2001

Figure 4.2: Purchase table.

4.2.2 Some Schemas Are Better Than Others

Consider the following two schema designs concerning information relating suppliers and their addresses to the parts on order and the quantity ordered of each part from each supplier.

Schema design I:

- Onorder1(supplier_ID, part_ID, quantity, supplier_address)

Schema design II:

- Onorder2(supplier_ID, part_ID, quantity)
- Supplier(supplier_ID, supplier_address)

Let us compare these two schema designs according to three criteria. Assume that there are 100,000 orders outstanding and 2000 suppliers altogether. Further assume that the supplier_ID is an 8-byte integer (64-bit machine) and the supplier_address requires 50 bytes.

1. *Space*: The second schema will use extra space for the redundant supplier_ID = 2000 × 8 = 16,000 bytes. The second schema stores will save space, however, by storing 2000 supplier addresses as opposed to 100,000 supplier addresses in the first schema. So, the second schema consumes 98,000 × 50 = 4,950,000 bytes less space for address information. Adding these two effects together, the second schema actually saves 4,934,000 bytes under the given assumptions. We admit that 5 megabytes has no importance, but the principle holds even if the number of rows was increased by 1000-fold, in which case the space difference becomes 5 gigabytes.
2. *Information preservation*: Suppose that supplier QuickDelivery has delivered all parts that were on order. The semantics of the relation Onorder1 dictate that the records pertaining to QuickDelivery should be deleted since the order has been filled. The problem is that the database would lose any record of QuickDelivery's address. Using the second schema, you would not lose the address information because it would be held in the Supplier relation.
3. *Performance*: Suppose you frequently want to know the address of the supplier from where a given part has been ordered. Then the first schema, despite the problems mentioned so far, may be good, especially if there are few updates. If, however, there are many new orders, that is, many insertions, then including the supplier address in every Onorder1 record requires extra data entry effort (which is likely to introduce dirty data) or entails an extra lookup to the database system for every part ordered from the supplier. So, these two schemas present a tuning trade-off that we will revisit later.

Relation Onorder2 is *normalized*, whereas Onorder1 is *unnormalized*. What do these terms mean?

Intuitively, the problem of the first schema is that the relationship between supplier_ID and supplier_address is repeated for every part on order. This wastes space and makes the presence of a given supplier's address dependent on the presence of an open order from that supplier.

We can formalize this intuition through the notion of functional dependencies. Suppose X is a set of attributes of a relation R, and A is a single attribute of R. We say that *X determines A* or that the functional dependency $X \rightarrow A$ holds for R if the following is true: for any relation

instance *I* of *R*, whenever there are two records *r* and *r´* in *I* with the same *X* values, they have the same *A* values as well. The functional dependency $X \rightarrow A$ is *interesting* (or, to use the term of the research community, nontrivial) if *A* is not an attribute of *X*.
Suppose you discover that each supplier is to be associated with a single address. This implies that any two records with the same supplier_ID value have the same supplier_address value as well. Thus, supplier_ID → supplier_address is an interesting functional dependency.
Having defined a functional dependency, we can now define a *key*. Attributes *X* from relation *R* constitute a key of *R* if *X* determines every attribute in *R* and no proper subset of *X* determines every attribute in *R*. For example, in the relation Onorder1, supplier_ID and part_ID together constitute a key. In the relation Supplier, supplier_ID by itself constitutes a key. So a key of a relation is a minimal set of attributes that determines all attributes in the relation.[1]

Check yourself with the following:

- Do you see why supplier_ID by itself does not constitute a key of relation Onorder1?
  Answer. Because supplier_ID does not determine part_ID.
- Do you see why supplier_ID and supplier_address together do not constitute a key of Supplier?
  Answer. Because supplier_ID by itself determines all the attributes. That is, supplier_ID and supplier_address do not constitute a minimal set of attributes that determines all attributes, but supplier_ID by itself is minimal.

A relation *R* is *normalized* if every interesting functional dependency $X \rightarrow A$ involving attributes in *R* has the property that *X* is a key of *R*.

Relation Onorder1 is not normalized because the key is supplier_ID and part_ID together, yet supplier_ID by itself determines supplier_address. It does not matter that supplier_ID is part of the key. For the relation to be normalized, supplier_ID would have to be the whole key.

Relation Onorder2 and Supplier are both normalized.

4.2.3 Normalization by Example

Practice Question 1 Suppose that a bank associates each customer with his or her home branch, that is, the branch where the customer opened his or her first account. Each branch is in a specific legal jurisdiction, denoted jurisdiction. Is the relation (customer, branch, jurisdiction) normalized?

Answer to Practice Question 1 Let us look at the functional dependencies. Because each customer has one home branch, we have

customer → branch

Because each branch is in exactly one jurisdiction, we have

branch → jurisdiction

So, customer is the key, yet the left-hand side of the functional dependency branch → jurisdiction is not customer. Therefore, (customer, branch, jurisdiction) is not normalized. Its problems are exactly the problems of Onorder1 earlier.

Relation (customer, branch, jurisdiction) will use more space than the two relations (customer, branch) and (branch, jurisdiction). Further, if a branch loses its customers, then the database loses the relationship between branch and jurisdiction. (The bank directors may not care, but information scientists dislike losing information by accident.)

Practice Question 2 Suppose that a doctor can work in several hospitals and receives a salary from each one. Is the relation (doctor, hospital, salary) normalized?

Answer to Practice Question 2 The only functional dependency here is doctor, hospital → salary. Therefore, this relation is normalized.

Practice Question 3 Suppose that we add the field primary_home_address to the previous question. Each doctor has one primary_home_address, but many doctors may have the same primary home address. Would the relation (doctor, hospital, salary, primary_home_address) be normalized?

Answer to Practice Question 3 We have a new functional dependency.

doctor → primary_home_address

Unfortunately, doctor is a proper subset of the key of the relation so the relation is unnormalized. The key is doctor and hospital together. A normalized decomposition would be

(doctor, hospital, salary)

(doctor, primary_home_address)

Practice Question 4 Suppose that a new law forbids doctors from working in more than one hospital. However, nothing else changes. In that case, is (doctor, hospital, salary, primary_home_address) normalized?

Answer to Practice Question 4 In this case, we have the following functional dependencies:

doctor → primary_home_address

doctor → hospital

doctor → salary

So, doctor by itself is the key, and the relation is normalized.

Practice Question 5 Suppose we have the situation of the previous question, but we add the hospital_address associated with each hospital. In that case, is (doctor, hospital, hospital_address, salary, primary_home_address) normalized?

Answer to Practice Question 5 To the functional dependencies that we have already, we would add

hospital → hospital_address

Because doctor is still the key of the relation, we have a functional dependency involving the attributes of a relation in which the left-hand side is not the key of the relation. Therefore, the relation is not normalized.

A possible decomposition would yield the relations

(doctor, hospital, salary, primary_home_address)

(hospital, hospital_address)

Notice that these two relations would probably occupy less space than the single unnormalized one because the hospital address would be represented only once per hospital.

4.2.4 A Practical Way to Design Relations

There are algorithms to design normalized relations given functional dependencies. In practice, such algorithms are far more complicated than necessary. An easier strategy is to find the *entities* in the application. Intuitively, entities make up what the application designer considers to be the "individuals" of the database. For example, doctors, hospitals, suppliers, and parts would be the entities of the applications we've spoken about so far (Figure 4.3). Usually, an

entity has *attributes* that are properties of the entity. For example, hospitals have a jurisdiction, an address, and so on. There are two formal constraints on attributes.

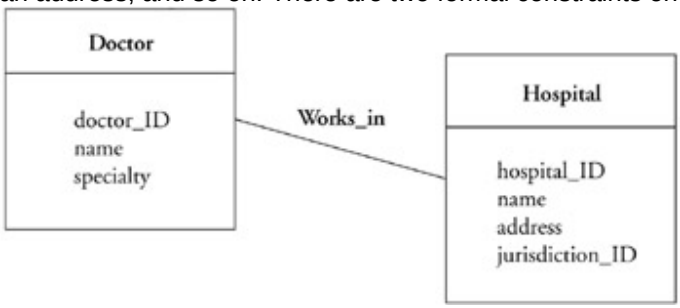


Figure 4.3: Entities. This entity-relationship diagram represents the hospital and doctor entities in our sample hospital application as well as the works_in relationship connecting those entities.

- An attribute cannot have attributes of its own. Only entities can have attributes.
- The entity associated with an attribute must functionally determine that attribute. (For example, there must be one value of address for a given hospital.) Otherwise, the attribute should be an entity in its own right.

Each entity with its associated attributes becomes a relation. For example, the entity doctor with attributes salary, primary_home_address, and so on becomes a relation. Hospital is not part of that relation according to this design methodology because hospital is an entity in its own right with attributes hospital_address and so on.
To that collection of relations, add relations that reflect *relationships* between entities. For example, there is a relationship *works_in* between doctors and hospitals. Its schema might be (doctor_ID, hospital_ID).

Three relations would result from this design strategy for an application concerning doctors and hospitals.

1. Doctor(doctor_ID, primary_home_address, …)
2. Hospital(hospital_ID, hospital_address, …)
3. Works_in(doctor_ID, hospital_ID)

Most CASE (computer-aided software engineering) tools include something similar to this entity-relationship design strategy.

4.2.5 Functional Dependency Test

In most cases, your relations will be normalized after you have used this entity-relationship methodology. To be sure, you should identify the functional dependencies and see. The two main conditions to check are that (1) each relation is normalized, and (2) the attributes constituting each "minimal" functional dependency are a subset of the attributes of some relation. For example, if the functional dependency $AB \rightarrow C$ holds and is minimal because neither $A \rightarrow C$ nor $B \rightarrow C$ holds, then attributes A, B, and C should all be attributes of some relation R.
Intuitively, a functional dependency is minimal if it is not implied by other functional dependencies and it does not have too many attributes on the left-hand side. For an example of a redundant functional dependency, consider the three functional dependencies $A \rightarrow B$, $B \rightarrow C$, and $A \rightarrow C$. The third one is implied by the other two. That is, any table satisfying the first two will also satisfy the last one. Given these functional dependencies, we would construct two

relations having schemas (*A*, *B*) and (*B*, *C*), but no relation having schema (*A*, *B*, *C*) or (*A*, *C*) because $A \rightarrow C$ is not minimal in this context.
Now let's see how a functional dependency can have too many attributes on its left-hand side. Suppose there are three functional dependencies $A \rightarrow B$, $ABF \rightarrow C$, and $BD \rightarrow E$. Then the second one has an unnecessary attribute on its left-hand side. The reason is that any table satisfying the first two functional dependencies will also satisfy $AF \rightarrow C$. (If two rows of the table have the same *AF* values, then they have the same *B* values because of $A \rightarrow B$ and the same *C* values because of $ABF \rightarrow C$.) On the other hand, the third functional dependency has no extra attributes on the left-hand side. So, the three corresponding minimal functional dependencies are $A \rightarrow B$, $AF \rightarrow C$, and $BD \rightarrow E$. This leads to the schemas *(A, B)*, *(A, F, C)*, and *(B, D, E)*, but not *(A, B, C, F)*.

If you are a mathematician at heart, you may want to read the rigorous but clear discussion of the preceding procedure in the freely available book by Serge Abiteboul, Rick Hull, and Victor Vianu on database theory.[2]

4.2.6 Tuning Normalization

Different normalization strategies may guide you to different sets of normalized relations, all equally good according to our criteria so far. Which set to choose depends on your application's query patterns.

Scenario: Different normalization sets 1

Suppose that we have three attributes: account_ID, balance, and address. The functional dependencies are

- account_ID → address

and

- account_ID → balance

There are two normalized schema designs in this case.

- (account_ID, address, balance)

and

- (account_ID, address)
- (account_ID, balance)

The question is which design is better? Let us look at the query patterns.

In most U.S. banks, the application that sends a monthly statement is the principal user of the address of the owner of an account. By contrast, the balance is updated or examined much more often, possibly several times a day. In such a case, the second schema might be better because the (account_ID, balance) relation can be made smaller.[3] Small size offers three benefits.

1. A sparse clustering index on the account_ID field of the (account_ID, balance) relation may be a level shorter than it would be for the (account_ID, balance, address) relation because the leaves of such an index will have only one pointer per data page, and there will be far fewer data pages.
2. More account_ID-balance pairs will fit in memory, thus increasing the hit ratio for random accesses.

3. A query that must scan a significant portion of account_ID-balance pairs will read relatively few pages.
In this case, two relations are better than one even though the two-relation solution requires more space. This principle is illustrated in Figure 4.4.

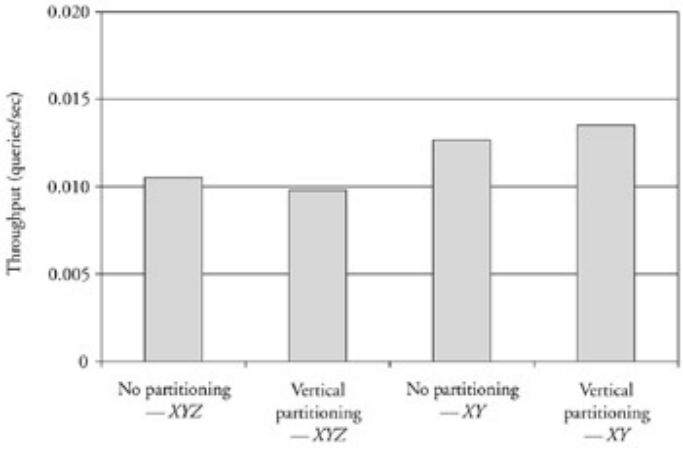


Figure 4.4: Vertical partitioning and scan queries. A relation *R* is defined with three attributes *X* (integer), *Y*, and *Z* (large strings); a clustered index is defined on *X*. This graph compares the performance of scan queries that access all three attributes, or only two of them (*X* and *Y*) depending on whether vertical partitioning is used or not. Vertical partitioning consists of defining two relations R_1 and R_2: R_1 has two attributes *X* and *Y* with a clustered index on *X*, whereas R_2 has two attributes *X* and *Z* with a clustered index on *X*, too. As expected, the graph shows that vertical partitioning exhibits poorer performance when all three attributes are accessed (vertical partitioning forces a join between R_1 and R_2 as opposed to a simple lookup on *R*) and better performances when only two attributes are accessed (fewer pages need to be scanned). This graph was obtained with SQL Server 2000 on Windows 2000.

Scenario: Different normalization sets 2

Suppose that the address field in scenario 1 was actually divided into two fields, a street address and a zip code. Would the following normalized schema design make sense?

- (account_ID, street address)
- (account_ID, zip code)
- (account_ID, balance)

Because street address and zip code are accessed together or not at all in this application, dividing the address information between two relations hurts performance and uses more space.

In this case, having one relation (account_ID, street address, zip code) will give better performance than having two (account_ID, street address) and (account_ID, zip code).
The preceding examples show that the choices among different normalized designs depend critically on the application's access patterns. Usually, a single normalized relation with attributes *XYZ* is better than two normalized relations *XY* and *XZ* because the single relation design allows a query to request *X*, *Y*, and *Z* together without requiring a join. The two-relation design is better if and only if the following two conditions hold:

- User accesses tend to partition between the two sets most of the time. If user accesses do not partition between the two sets, then, as Figure 4.5 shows, the join overhead can make the vertically partitioned layout worse than the nonpartitioned layout, at least for range queries.

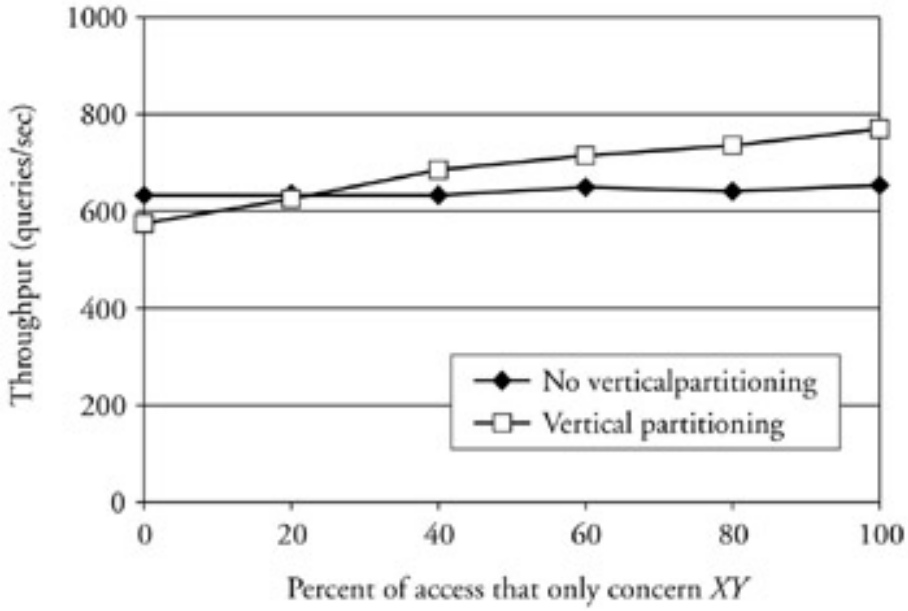


Figure 4.5: Vertical partitioning and single point queries. A relation *R* is defined with three attributes *X* (integer), *Y*, and *Z* (large strings); a clustered index is defined on *X*. A mix of point queries access either all three attributes, or only *X* and *Y*. This graph shows the throughput for this query mix as the proportion of queries accessing only *X* and *Y* increases. We compare query execution with or without vertical partitioning of *R*. Vertical partitioning consists of defininin two relations R_1 and R_2: R_1 has two attributes *X* and *Y* with a clustered index on *X*, whereas R_2 has two attributes *X* and *Z* with a clustered index on *X*, too. The graph is obtained by running five clients that each submit 20 different point queries either on *X, Y, Z* (these require a join if vertical partitioning is used) or only on *X, Y*. The graph shows that vertical partitioning gives better performances when the proportion of queries accessing only *X* and *Y* is greater than 20%. In this example the join is not really expensive compared to a simple lookup—indeed the join consists of one indexed access to each table followed by a tuple concatenation. This graph was obtained with SQL Server 2000 on Windows 2000.

- Attribute *Y* values or attribute *Z* values or both are large (one-third the page size or larger).

Bond scenario

In certain cases, you may start with a vertically partitioned schema and then perform what may be called *vertical antipartitioning*. Here is an example.

Some brokers base their bond-buying decisions on the price trends of those bonds. The database holds the closing price for the last 3000 trading days. However, statistics regarding the 10 most recent trading days are especially important. (If your trading system requires microstructure—that is, intraday trading histories—the principles are exactly the same.)

Consider therefore the following two schemas:

- (bond_ID, issue_date, maturity, …)—about 500 bytes per record
- (bond_ID, date, price)—about 12 bytes per record

versus

- (bond_ID, issue_date, maturity, today_price, yesterday_price, … 10dayago_price)—about 600 bytes per record

- (bond_ID, date, price)

Suppose we arrange for the second schema to store redundant information. That is, both relations will store the data from the last 10 days of activity. This will cost a little extra space, but will be much better for queries that need information about prices in the last ten days because it will avoid a join and will avoid fetching 10 price records per bond. Even if (bond_ID, date, price) is clustered by bond_ID and date, the second schema will save at least one disk access for such queries. It may save more depending on the number of overflow pages needed to store all the price records associated with a given bond.

It is possible to avoid redundancy by storing records in the (bond_ID, date, price) table only after they become more than 10 days old, but then queries that want the price on a given day or the average price over a range of 50 consecutive days (given as a parameter) become very difficult to write.

4.2.7 Tuning Denormalization

Denormalization means violating normalization. Given all the disadvantages of such a move, it has only one excuse: performance.

- Even this excuse applies only to some situations. The schema design
- Onorder1(supplier_ID, part_ID, quantity, supplier_address)

is bad for performance when inserts are frequent. As mentioned earlier, the data entry operator must either enter address information each time or look up the supplier address when performing an insertion.
That schema design would help a query that returns the parts available from suppliers having some zip code, however. Figure 4.6 illustrates this principle on an example based on the TPC-H schema.

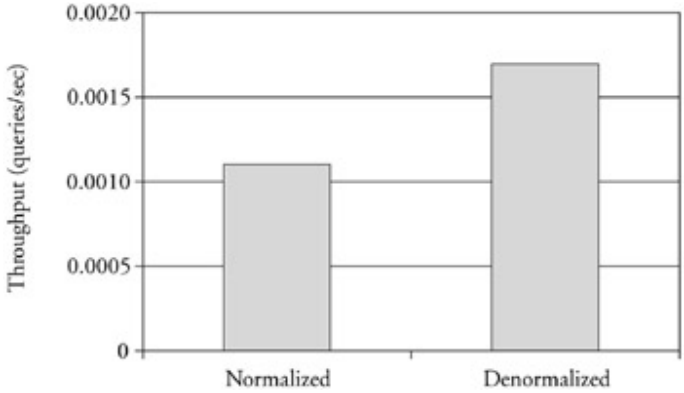


Figure 4.6: Denormalization. We use the TPC-H schema to illustrate the potential benefits of denormalization. This graph shows the performance of a query that finds all lineitems whose supplier is in Europe. With the normalized schema, this query requires a four-way join between lineitem, supplier, nation, and region. If we denormalize lineitem and introduce the name of the region each item comes from, then the query is a simple selection on lineitem. In this case, denormalization provides a 30% improvement in throughput. This graph was obtained with Oracle 8i EE running on Windows 2000.

As a general rule, denormalization hurts performance for relations that are often updated. However, denormalization may help performance in low-update situations. For that reason, some applications denormalize their archival data while keeping their online data normalized.

[1]There is an unfortunate confusion in database terminology regarding the word *key*. The key of an index is the attribute or sequence of attributes by which you can search the index. There may be many records in a given relation with the same value of a given index key. In normalization

theory, a key of a relation *R* is a set of attributes such that no two records of any instance of *R* have the same values on all the attributes of that key. You must always use context to determine the sense in which "key" is meant. This chapter uses "key" in the sense of normalization theory.

[2]S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[3]In making this statement about size, we are assuming that the table is stored one row at a time. That is the conventional storage approach. If the table is stored one column at a time (as is done, for example, in KDB), then splitting this table into two has no benefit.

4.3 Clustering Two Tables

Oracle offers the possibility to cluster two tables together based on the key of one of the tables. This can be considered a compromise between normalization and denormalization because it groups rows together that would be joined in forming a denormalized table. For example, let us take our supplier/order example.

- Supplier(supplier_ID, supplier_address)
- Onorder(supplier_ID, part_ID, quantity)

Oracle clustering would intermix these two tables, so there would be a single Supplier record followed by all the Onorder records that match that supplier. In that case, supplier_ID is said to be a *cluster key*. Here is how the layout might appear.

- Supplier 1 record
  - Onorder 235 record
  - Onorder 981 record
  - Onorder 112 record
- Supplier 2 record
  - Onorder 239 record
  - Onorder 523 record
  - Onorder 867 record

Clustering tables has its good and bad points.

- Queries on the cluster key are fast. Thus, a query that wants to find all orders having to do with a particular supplier will execute quickly.
- Point queries on either Supplier or Onorder that use indexes will behave as well as point queries using nonclustering indexes on the standard layout.
- Full table scans of the Onorder table will be somewhat slower than in a standard layout. Full table scans of the Supplier table will be much slower than in a standard layout because there may be many Onorder records between neighboring Supplier records.
- Insertions may cause overflow chaining, slowing the performance of cluster key searches. In fact, if there are enough insertions, application performance may degrade badly. It is mostly for this reason that Oracle consultants rarely recommend clustering of two tables. In fact, the documentation advertises clustering mainly for the use of a hash index on a single table.

DIGRESSION REGARDING DEFINITIONS

It is important to understand the similarities and differences between table clustering and index clustering.

- Both concepts force a certain organization on table records.
- A clustering index forces an organization onto the records of a single table and provides an index to access the records of that table. Table clustering forces an intermixing of the records between two different tables based on an attribute, essentially precomputing a join.
- As far as the larger table is concerned, the performance effect of table clustering is similar to having a clustering index on the cluster key field. For example, clustering Onorder with Supplier on supplier_ID gives similar performance to a clustering index on Onorder.supplier_ID.

4.4 Aggregate Maintenance

In reporting applications, aggregates (sums, averages, and statistics) are frequently used to make the data more intelligible. For such queries, it may be worthwhile to maintain special tables that hold those aggregates in precomputed form. Even though we discuss strategies for constructing such "materialized views" in Chapter 9, the rationale for such tables applies even to transactional databases, so we present the basic ideas here.

Consider the following example. The accounting department of a chain of convenience stores issues queries every 20 minutes to discover the total dollar amount on order from a particular vendor and the total dollar amount on order by a particular store. They have a normalized schema.

- Order(ordernum, itemnum, quantity, purchaser, vendor)
- Item(itemnum, price)
- Store(store, vendor)

Order and Item each has a clustering index on itemnum. Store is small; thus no index is defined.

Both "total amount" queries execute very slowly and disrupt normal insert processing into these relations. The database administrator believes he must denormalize Order to include the price. What do you suggest?
Action. Denormalizing would eliminate one join. However, updating the price of an item would be complicated. What's more, every insertion into Order may require a lookup in the (still present) Item table to discover the price.

Another approach is to consider temporal partitioning. If the queries can be executed during off-hours, then they should be.

If the queries must be executed often, then consider creating two additional relations.

- VendorOutstanding(vendor, amount), where amount is the dollar value of goods on order to the vendor, with a clustering index on vendor.
- StoreOutstanding(store, amount), where amount is the dollar value of goods on order by the store, with a clustering index on store.

This approach, known as *aggregate maintenance*, requires updates to VendorOutstanding and StoreOutstanding on every update of the Order table, which makes it more expensive to insert Order records. The benefit is that the "total amount" queries require a simple lookup with this approach. Thus, aggregate maintenance is worthwhile if the time saved for the lookup queries is greater than the extra time required by all updates between two lookups (Figures 4.7, 4.8, and 4.9).

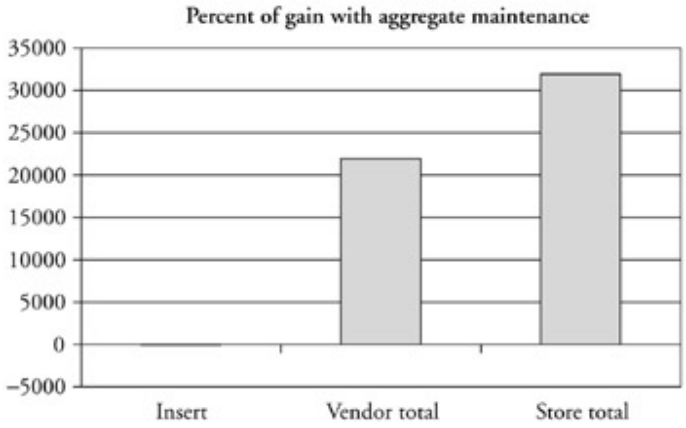


Figure 4.7: Aggregate maintenance. We implemented the *total amount* example given in Section 4.4 to compare the benefits and costs of aggregate maintenance using SQL Server 2000 on Windows 2000. The aggregate maintenance solution relies on triggers that update the VendorOutstanding and StoreOutstanding relations whenever a new order is inserted. The graph shows the gain obtained with aggregate maintenance. For inserts this gain is negative: the execution of the triggers slows down insertions by approximately 60%. For queries, however, the gain is spectacular: response time is reduced from 20 seconds to 0.1 second. The reason is that the "total amount" queries are simple scans on very small relations when aggregate maintenance is used, whereas they are three- or four-way joins on the large order relations otherwise.

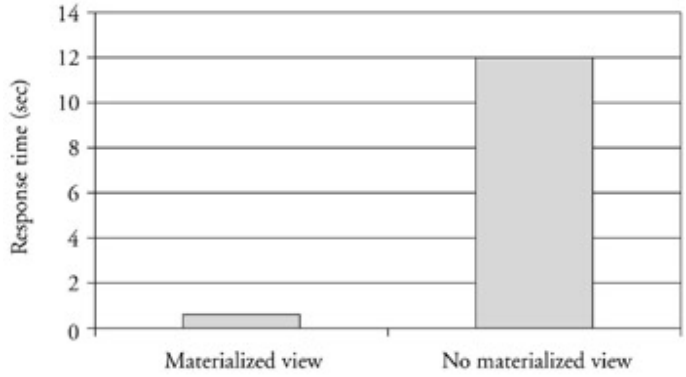


Figure 4.8: Aggregate maintenance with materialized views (queries). We implemented the *total amount* example given in Section 4.4 to compare the benefits and costs of aggregate maintenance using Oracle 9i on Linux. Materialized views are transparently maintained by the system to reflect modifications on the base relations. The use of these materialized views is also transparent; the optimizer rewrites the aggregate queries to use materialized views if appropriate. We use a materialized view to define VendorOutstanding. The speed-up for queries is two orders of magnitude.

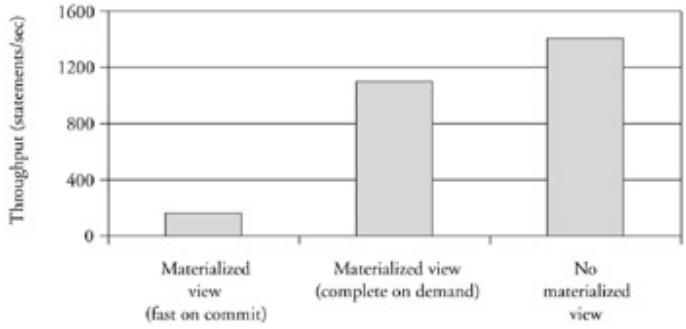


Figure 4.9: Aggregate maintenance with materialized views (insertions). There are two main parameters for view materialization maintenance: (1) the materialized view can be updated in the

transaction that performs the insertions (ON COMMIT), or it can be updated explicitly after all insert transactions are performed (ON DEMAND); (2) the materialized view is recomputed completely (COMPLETE) or only incrementally depending on the modifications of the base tables (FAST). The graph shows the throughput when inserting 100,000 records in the orders relation for FAST ON COMMIT and COMPLETE ON DEMAND. On commit refreshing has a very significant impact on performance. On demand refreshing should be preferred if the application can tolerate that materialized views are not completely up to date or if insertions and queries are partitioned in time (as it is the case in a data warehouse).

USE OF REDUNDANCY TO ENHANCE PERFORMANCE

Denormalization and aggregate maintenance rely on redundancy to enhance performance:

- As the schema Onorder1(supplier_ID, part_ID, quantity, supplier_address) showed, denormalization creates redundant relationships (many copies of a supplier_ID, supplier_address pair) that slow down insertions and updates, but speed join queries.
- Maintaining aggregates as in the VendorOutstanding and Store-Outstanding tables is again a trade-off that slows down insertions and updates, but speeds up aggregate queries.
- Defined but "unmaterialized" views have no influence on performance. They are the database equivalent of macros whose definition is expanded at query time.

4.5 Record Layout

Once you have determined the attributes of each relation, you face the far easier, but important task of choosing the data types of the attributes. The issues are fairly straightforward.

- An integer is usually better to use than a float because floats tend to force selections to perform range queries. For example, if attribute *A* is computed based on some floating-point expression, then a selection to find records having value 5 in attribute *A* may require the qualification
- WHERE A >= 4.999
- AND A <= 5.001

  to avoid problems having to do with different machine precisions. Therefore, you should choose integer values for data such as salaries (record the salary in pennies) and stock prices.
- If the values of an attribute vary greatly in size and there are few updates, then consider using a variable-sized field.

  On most systems, a variable-sized field will be only a few bytes longer than the value it represents. A fixed-sized field has to be long enough for the longest field value. So, variable-sized fields give better space utilization. This makes them better for scans and for clustering indexes. The main cost is that the overhead of doing storage management for variable-sized fields can become significant if there are many updates. Especially disruptive is an update that makes a certain field much larger, for example, changing the address of an Employee record from NULL to a rural postal address. Such an update may cause that field to be placed on another page or may move the entire containing record to a new page depending on your database system. As discussed in Chapter 2, there is a close relationship between your page utilization parameters and your application's use of variable-length fields.
- Modern database systems provide support for Binary Large OBjects (BLOBs): variable-length values of up to 2 or 4 gigabytes. These attribute values are not stored with the rest of a row, but rather on separate pages (it is often a good idea to store BLOBs on separate disks to enable sequential scanning of the non-BLOB part).[4] Oracle even allows values of type BFILE to be stored in the file system outside the database system. Each row contains a pointer to the location of its BLOB's attributes. In IBM UDB V7.1, the size

of the pointer depends on the maximum size declared for the BLOB. It varies from 72 bytes for a BLOB of less than 1 kilobyte to 316 bytes for a BLOB of 2 gigabytes. This improves space utilization if the maximum size of BLOBs is known beforehand.

[4]An alternative to using BLOBs is often to store pointers to files in the database. The following article discusses the example of an image database. It recommends storing pointers to files as VARCHAR in the database instead of images that would be stored as BLOBs.

Andy Rosebrock and Stan Schultes, "Store Images in Your Database," *Visual Basic Programmer's journal*, February 2001, vol. 11, no. 2. Fawcette Technical Publications, Palo Alto, Calif.

4.6 Query Tuning

The first tuning method to try is the one whose effect is purely beneficial. Most changes such as adding an index, changing the schema, or modifying transaction lengths have global and sometimes harmful side effects. Rewriting a query to run faster has only beneficial side effects.

There are two ways to see that a query is running too slowly.

1. It issues far too many disk accesses, for example, a point query scans an entire table.
2. You look at its query plan and see that relevant indexes are not used. (The *query plan* is the the method chosen by the optimizer to execute the query.)

Refer to Chapter 7 on monitoring methods. Here are some examples of query tuning and the lessons they offer. The examples use three relations.

- Employee(ssnum, name, manager, dept, salary, numfriends)

Clustering index on ssnum and nonclustering indexes on name and dept each. Ssnum and name each is a key.

- Student(ssnum, name, course, grade)

Clustering index on ssnum and a nonclustering index on name. Ssnum and name each is a key.

- Techdept(dept, manager, location)

Clustering index on dept. Dept is the key. A manager may manage many departments. A location may contain many departments.

Let us discuss DISTINCT first.

1. In most systems, DISTINCT will entail a sort or other overhead, so should be avoided.

Query. Find employees who work in the information systems department. There should be no duplicates.

```
SELECT DISTINCT ssnum
FROM Employee
WHERE dept = 'information systems'
```

There is no need for the keyword DISTINCT since ssnum is a key of Employee so certainly is a key of a subset of Employee. (Because of the index on ssnum, this particular query may not encounter extra overhead as a result of the DISTINCT keyword, but some cases are not so obvious as we will see later.)

2. Many subsystems handle subqueries inefficiently.

Query. Find employee Social Security numbers of employees in the technical departments. There should be no duplicates.

```
SELECT ssnum
```

```
FROM Employee
WHERE dept IN (SELECT dept FROM Techdept)
```

This query might not use the index on Employee dept in some systems. Fortunately, the query is equivalent to the following one, which would use the index on Employee dept:

```
SELECT ssnum
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
```

Note that if employees could belong to several departments, then the second query would require DISTINCT, whereas the first one would not. Since ssnum is a key of Employee and dept is a key of Techdept, neither query needs DISTINCT for reasons we will see shortly.

3. The unnecessary use of temporaries can hurt performance for two reasons. First, it may force operations to be performed in a suboptimal order. Second, in some systems, the creation of a temporary causes an update to the catalog, perhaps creating a concurrency control hot spot.
Query. Find all information department employees who earn more than $40,000.

```
SELECT * INTO Temp
FROM Employee
WHERE salary > 40000

SELECT ssnum
FROM Temp
WHERE Temp.dept = 'information'
```

Not only is there overhead to create the temporary, but the system would miss the opportunity to use the index on dept. A far more efficient solution would be the following:

```
SELECT ssnum
FROM Employee
WHERE Employee.dept = 'information'
AND salary > 40000
```

4. Complicated correlation subqueries may execute inefficiently, so perhaps should be rewritten. In that case, temporaries may help.
Query. Find the highest paid employees per department.

```
SELECT ssnum
FROM Employee e1
WHERE salary =
(SELECT MAX(salary)
 FROM Employee e2
 WHERE e2.dept = e1.dept
)
```

This query may search all of e2 (that is, all of the Employee relation) for each record of e1 (or at least all the records in each department). In that case, this query should be replaced by the following query that uses temporaries:

```
SELECT MAX(salary) as bigsalary, dept INTO Temp
FROM Employee
GROUP BY dept
```

```
SELECT ssnum
FROM Employee, Temp
WHERE salary = bigsalary
AND Employee.dept = Temp.dept
```

Observe that you would not need DISTINCT because dept is a key of Temp. In the terminology that we will develop later, Temp "reaches" Employee.

5. Temporaries may also help avoid ORDER BYs and scans when there are many queries with slightly different bind variables.
Queries. For the salary ranges, $40,000 to $49,999, $50,000 to $59,999, $60,000 to $69,999, and $70,000 to $79,999, order the employees by ssnum. Thus, there are four queries.

Each query would have the form (with 40000 and 49999 replaced appropriately)

```
SELECT ssnum, name
FROM Employee
WHERE salary >= 40000
AND salary >= 49999
ORDER BY ssnum
```

That is, each would require a scan through Employee and a sort of the records that survive the qualification on salary. A better approach would do the following:

```
SELECT ssnum, name, salary INTO Temp
FROM Employee
WHERE salary >= 40000
AND salary <= 79999
ORDER BY ssnum
```

A typical query would then have the form

```
SELECT ssnum, name
FROM Temp
WHERE salary >= 40000
AND salary >= 49999
```

The reformulation would require only

- A single ORDER BY of the records whose salaries satisfy the constraints of the four queries.
- Four scans without an ORDER BY statement of these selected records.

The big savings comes from avoiding a scan of the entire Employee relation for each query. This will reduce the necessary number of disk accesses.

6. It is a good idea to express join conditions on clustering indexes. Failing that, prefer a condition expressing numerical equality to one expressing string equality.
Query. Find all the students who are also employees.

```
SELECT Employee.ssnum
FROM Employee, Student
WHERE Employee.name = Student.name
```

In this case, the join is correct because name is a key, but we can make it more efficient by replacing the qualification as follows:

```
SELECT Employee.ssnum
```

```
FROM Employee, Student
WHERE Employee.ssnum = Student.ssnum
```

This will speed up the query by permitting a merge join, since both relations are clustered on ssnum.

7. Don't use HAVING when WHERE is enough. For example, the following query finds the average salary of the information department, but may first perform the grouping for all departments.

```
8.  SELECT AVG(salary) as avgsalary, dept
9.  FROM Employee
10. GROUP BY dept
11. HAVING dept = 'information';
```

The following will first find the relevant employees and then compute the average.

```
SELECT AVG(salary) as avgsalary
FROM Employee
WHERE dept = 'information'
GROUP BY dept;
```

12. Study the idiosyncrasies of your system. For example, some systems never use indexes when different expressions are connected by the OR keyword.
Query. Find employees with name Smith or who are in the acquisitions department.

```
SELECT Employee.ssnum
FROM Employee
WHERE Employee.name = 'Smith'
OR Employee.dept = 'acquisitions'
```

Check the query plan. If no index is used, then consider using a union.

```
SELECT Employee.ssnum
FROM Employee
WHERE Employee.name = 'Smith'

UNION

SELECT Employee.ssnum
FROM Employee
WHERE Employee.dept = 'acquisitions'
```

13. Another idiosyncrasy is that the order of tables in the FROM clause may affect the join implementation though this does not usually take effect until more than five tables are involved.

If one relation is much smaller than the other, it is better to scan the smaller one. So, the ordering of the tables in the query can be important for performance (though the set of records in the answer won't change).

14. Views may cause queries to execute inefficiently. Suppose we create a view Techlocation as follows:

```
15. CREATE VIEW Techlocation
16. AS SELECT ssnum, Techdept.dept, location
17. FROM Employee, Techdept
18. WHERE Employee.dept = Techdept.dept
```

The view definition can be read as if it were a table. For example,

```
SELECT location
FROM Techlocation
WHERE ssnum = 452354786
```

In this case, the database management system will process such a query based on the definition of Techlocation. That is, the system will execute

```
SELECT location
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
AND ssnum = 452354786
```

Thus, the use of a view cannot give better performance than a query against base tables. (If this query is performed frequently and updates are rare, then it may be worthwhile to maintain the unnormalized relation Techlocation as a redundant base table. However, a view does not do that.)

A view can easily lead you to write inefficient or even incorrect queries, however. For example, consider a similar query:

```
SELECT dept
FROM Techlocation
WHERE ssnum = 452354786
```

This will be expanded to a formulation having a join:

```
SELECT dept
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
AND ssnum = 452354786
```

Because dept is an attribute of Employee, the following less expensive query is possible:

```
SELECT dept
FROM Employee
WHERE ssnum = 452354786
```

The query against the view might lead to an incorrect response if the given employee does not work in a technical department.

Figure 4.10 shows the performance impact of some of the rewriting techniques already described. We can make the following observations. First, some of these cases are identified during optimization, and the systems perform the rewriting on their own (subquery on Oracle and DB2, correlated subqueries on SQL Server, HAVING on SQL Server). Second, removing DISTINCTs when they're not required helps though only a little. Third, the performance of correlated subqueries is spectacularly poor on Oracle and DB2. On these systems, it is about 100 times faster to introduce a temporary relation. They both perform an aggregate on the inner branch of a nested loop join. SQL Server, on the other hand, does a good job at handling the correlated subquery: a hash join is performed.[5] Fourth, using numerical attributes as opposed to string attributes to perform a join helps significantly. Fifth, Oracle suffers a significant performance penalty when a query uses HAVING even though a WHERE would do. Note finally that the use of statistics makes a significant difference. Outdated statistics lead the optimizer to choose an execution plan that performs significantly worse than the execution plan chosen with updated statistics.

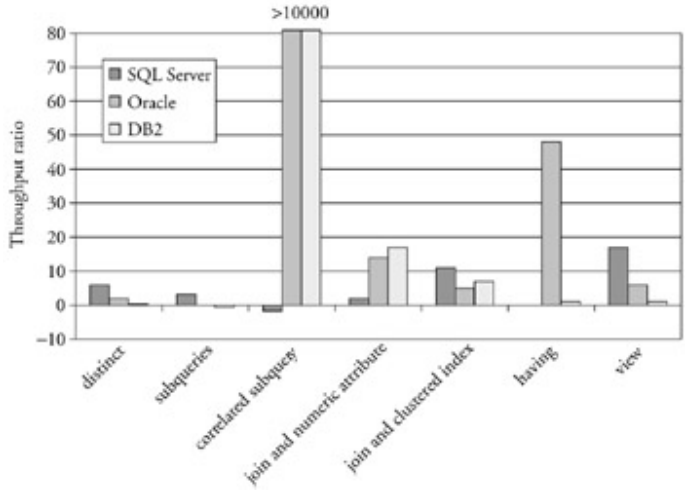


Figure 4.10: Query tuning. This graph shows the percent increase in throughput between the original query and the rewritten query for the rewriting techniques we have presented in this section: distinct refers to the suppression of unnecessary DISTINCTs, subquery refers to the use of joins instead of uncorrelated subqueries without aggregates, correlated subquery refers to the decomposition of correlated subqueries using intermediate tables, join and numeric attribute refers to the use of a numeric attribute rather than an equivalent string attribute as the joining attribute, join and clustered index refers to the use of attributes on which a clustered index is defined on join attributes, having refers to the incorporation of selection conditions in the WHERE clause rather than in the HAVING clause, and view refers to the use of a selection on a base table instead of a view expanded to a join. These experiments were performed using IBM UDB V7.1, Oracle 8i, and SQL Server 2000 on Windows 2000.

4.6.1 Minimizing DISTINCTs

The interrelated questions of minimizing DISTINCTs and eliminating certain kinds of nested queries can be subtle. This subsection concerns DISTINCT.

In general, DISTINCT is needed when

- the set of values or records returned should contain no duplicates.
- the fields returned do not contain (as a subset) a key of the relation created by the FROM and WHERE clauses.

In the query

```
SELECT ssnum
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
```

DISTINCT is not needed because an Employee record *e* will survive the join with Techdept only if *e* has the same department as some Techdept record *t*. Because dept is a key of Techdept, there will be at most one such record *t*, so *e* will be part of at most one record of the join result. Because ssnum is a key of Employee, at most one record in Employee will have a given ssnum value.
Generalizing from this example, we conclude that if the fields returned constitute a key of one table *T* and all other tables perform an equijoin with *T* by their keys, then the values returned will contain no duplicates, so DISTINCT will be unnecessary.

Technical generalization: Notion of reaching

In fact, DISTINCT is unnecessary in more general situations. To describe those situations, we must resort to a little mathematics and graph theory. Call a table *T privileged* if the fields returned by the select contain a key of *T*.
Let *R* be an unprivileged table. Suppose that *R* is joined on equality by its key field to some other table *S*, then we say that *R reaches S*. We define *reaches* to be transitive. So, if R_1 reaches R_2 and R_2 reaches R_3, then we say that R_1 reaches R_3.

There will be no duplicates among the records returned by a selection, even in the absence of DISTINCT, if the following two conditions hold:

- Every table mentioned in the select line is privileged.
- Every unprivileged table reaches at least one privileged one.

The reason this works is the following. If every relation is privileged, then there are no duplicates even without any qualification. Suppose some relation *T* is not privileged but reaches at least one privileged one, say, *R*. Then the qualifications linking *T* with *R* ensure that each distinct combination of privileged records is joined with at most one record of *T*.

Here are some examples to train your intuition.

- Note that the following slight variation of the query above would return duplicates:
- SELECT ssnum
- FROM Employee, Techdept
- WHERE Employee.manager = Techdept.manager

The reason is that the same Employee record may match several Techdept records (because manager is not a key of Techdept), so the Social Security number of that Employee record may appear several times. The formal reason is that the unprivileged relation Techdept does not reach privileged relation Employee.

- If the preceding example were changed slightly to make Techdept privileged
- SELECT ssnum, Techdept.dept
- FROM Employee, Techdept
- WHERE Employee.manager = Techdept.manager

then the problem goes away because each repetition of a given ssnum value would be accompanied by a new Techdept.dept since Techdept.dept is the key of Techdept.

- In fact, the qualification isn't even necessary. That is, the following query would have no duplicates either:
- SELECT ssnum, Techdept.dept
- FROM Employee, Techdept

- If Techdept were not privileged in a query, however, then the query might produce duplicates as in the following example:
- SELECT ssnum, Techdept.manager
- FROM Employee, Techdept

- Finally, the reaches predicate may go through an intermediate relation yet still ensure that there are no duplicates. Recall here that name is a key of Employee. (It also happens to be a key of Student though even if it weren't, there would be no duplicates in the result of the following query.)
- SELECT Student.ssnum
- FROM Student, Employee, Techdept
- WHERE Student.name = Employee.name
- AND Employee.dept = Techdept.dept

The formal reason is that both Employee and Techdept reach Student. However, let us try to show this directly. If a given Student's ssnum appeared more than once, then there would be some Student record *s* that survived twice in the qualification. That is, there are Employee records *e* and *e′* and Techdept records *t* and *t′* such that *s, e, t* and *s, e′, t′* both survive the qualification. Because name is the key of Employee, however, *e* = *e′*. Because dept is the key of Techdept, only one record can join with *e*, so *t* = *t′*.

- Can you see why the following might have duplicates?
- SELECT Student.ssnum
- FROM Student, Employee, Techdept
- WHERE Student.name = Employee.name
- AND Employee.manager = Techdept.manager

4.6.2 Rewriting of Nested Queries

It is unfortunate but true that most query optimizers perform much less well on some types of nested queries than on the corresponding nonnested ones. The four major kinds of nested queries are

1. Uncorrelated subqueries with aggregates in the inner query
2. Uncorrelated subqueries without aggregates
3. Correlated subqueries with aggregates
4. Correlated subqueries without aggregates

Since the first three are the most common, we concentrate on them.

Uncorrelated subqueries with aggregates

Consider the following example: "Find all employees who earn more than the average employee salary."

```
SELECT ssnum
FROM Employee
WHERE salary > (SELECT AVG(salary) FROM Employee)
```

Virtually all commercial systems would compute the average employee salary first and then insert the result as a constant in the outer query. This type of query causes no particular performance problem.

Uncorrelated subqueries without aggregates

Recall the example from earlier: "Find all employees in departments that are also in the Techdept relation."

```
SELECT ssnum
FROM Employee
WHERE dept IN (SELECT dept FROM Techdept)
```

This query might not use the index on dept in employee in many systems. Consider the following transformation:

1. Combine the arguments of the two FROM clauses.
2. AND together all the where clauses, replacing IN by =.

3. Retain the SELECT clause from the outer block.

This yields

```
SELECT ssnum
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
```

The transformation will work for nestings of any depth, but sometimes we have to use the formalism concerning privilege and reaching. Consider, for example, the following:

```
SELECT AVG(salary)
FROM Employee
WHERE dept IN (SELECT dept FROM Techdept)
```

This will be equivalent to

```
SELECT AVG(salary)
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
```

because the same number of salary values result from the join as from the nested query. The reason is that the salary value from a given Employee record will be included at most once in both queries because Techdept "reaches" (as we saw) Employee in this query.

By contrast, the following query

```
SELECT AVG(salary)
FROM Employee
WHERE manager IN (SELECT manager FROM Techdept)
```

could yield a different value from

```
SELECT AVG(salary)
FROM Employee, Techdept
WHERE Employee.manager = Techdept.manager
```

because the second one may include an Employee record several times if that Employee's manager is the manager of several Techdepts. (As you can see, the reaches formalism is quite useful when duplicates matter. Duplicates do not matter for aggregates like MIN and MAX.)

The best solution would be to create a temporary relation, say, Temp with key field manager. Then,

```
SELECT DISTINCT manager INTO Temp
FROM Techdept


SELECT AVG(salary)
FROM Employee, Temp
WHERE Employee.manager = Temp.manager
```

Correlated subqueries

Let us modify slightly the uncorrelated subquery from the preceding example: "Find employees who earn exactly the average salary in their department where their department is a technical one."

```
SELECT ssnum
```

```
FROM Employee e1, Techdept
WHERE salary = (SELECT AVG(e2.salary)
        FROM Employee e2, Techdept
        WHERE e2.dept = e1.dept
        AND e2.dept = Techdept.dept)
```

In most cases, this query will be quite inefficient so should be transformed to

```
SELECT AVG(salary) as avsalary, Employee.dept INTO Temp
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
GROUP BY Employee.dept

SELECT ssnum
FROM Employee, Temp
WHERE salary = avsalary
AND Employee.dept = Temp.dept;
```

This transformation can be characterized as follows:

1. Form a temporary based on a GROUP BY on the attribute (or attributes) of the nested relation that is (or are) correlated with the outer relation. (The correlation must be equality for this to work.) In the example, the attribute was dept. Use the uncorrelated qualifications from the subquery in the construction of the temporary. In the example, that was the qualification
2. Employee.dept = Techdept.dept
3. Join the temporary with the outer query. A condition on the grouped attribute replaces the correlation condition. In the example, the condition is
4. Employee.dept = Temp.dept

A condition between the comparing attribute and the dependent attribute of the grouping replaces the subquery condition. In the example, that corresponds to the replacement of

```
WHERE salary = (SELECT AVG(e2.salary)
```

by

```
WHERE salary = avsalary
```

All other qualifications in the outer query remain. There were none in this example.

By definition, the GROUP BY attribute(s) will constitute a key of the resulting temporary. Therefore, the temporary relation will always reach the outer relation. This eliminates any problem about duplicates. The correlation predicate was

```
WHERE e2.dept = e1.dept
```

in the example. Can you show by example why the transformation could be incorrect if the correlation predicate were not equality? (*Hint:* The grouping operator could create the wrong aggregate values.)

There is one more problem concerning empty sets and counting aggregates. Consider the following slight variation of the preceding query: "Find employees whose number of friends equals the number of employees in their department where their department is a technical one."

```
SELECT ssnum
FROM Employee e1
```

```
WHERE numfriends = COUNT(SELECT e2.ssnum
FROM Employee e2, Techdept
WHERE e2.dept = e1.dept
AND e2.dept = Techdept.dept)
```

Construct a similar transformation to the preceding one.

```
SELECT COUNT(ssnum) as numcolleagues, Employee.dept INTO Temp
FROM Employee, Techdept
AND Employee.dept = Techdept.dept
GROUP BY Employee.dept


SELECT ssnum
FROM Employee, Temp
WHERE numfriends = numcolleagues
AND Employee.dept = Temp.dept
```

Can you see why the result of this transformation is not equivalent? (*Hint:* Consider an employee Helene who is not in a technical department.)

In the original query, Helene's friend's list would be compared with the count of an empty set, which is 0, so her record would survive the selection provided she has no friends. In the transformed query, her dept would not correspond to a dept value in Temp (because it would not correspond to a department value in Techdept). So she would not appear in the output of the second query no matter how few friends she has.

In the first query, this would not have been a problem because Helene's salary would never be equal to the average salary of an empty set (which is NULL in most systems).

For more details, see the papers of Richard A. Ganski and Harry K. T. Wong[6] and of Won Kim.[7] The Ganski and Wong paper extends the Kim paper and corrects a few bugs. These transformations can be very tricky.

[5]The tehniques implemented in SQL Server for optimizating subqueries and aggregates are described in "Orthogonal Optimization of Subqueries and Aggregates" by Cesar A. Galindo-Legaria and Milind Joshi, SIGMOD, 2001.

[6]R. A. Ganski and H. K. T. Wong, "Optimization of Nested SQL Queries Revisited," ACM SIGMOD Conference 1987, 23–33.

[7]Won Kim, "On Optimizing an SQL-Like Nested Query," *Transactions on Database Systems*, vol. 7, no. 3, 443–469, September 1982.

4.7 Triggers

A *trigger* is a stored procedure that executes as the result of an event.[8] In relational systems, the event is usually a modification (insert, delete, or update) or a timing event (it is now 6 A.M.). The event that causes a trigger to execute its modification is called the *enabling event*, and it is said to *fire* that trigger. The trigger executes as part of the transaction containing the enabling event.

4.7.1 Uses of Triggers

There are three main reasons to use a trigger, only one of which has to do with performance.

- A trigger will fire regardless of the application that enables it. This makes triggers valuable for auditing purposes or to reverse suspicious actions. For example, the following trigger rolls back transactions that try to update salaries on weekends. (We use SYBASE syntax here.)

```
CREATE TRIGGER nosalchange
ON Employee
FOR update
AS
IF update(salary)
AND datename(dw, getdate()) IN ('Saturday,' 'Sunday')
BEGIN

  roll back transaction
  PRINT 'Nice try, buster!'
END
```

- A trigger can maintain integrity constraints that the application is unaware of. Referential integrity is the constraint that every value in some column *A* of table *T* should be present in column *A′* of table *T′*. For example, the following trigger deletes all accounts from a deleted branch:

```
CREATE TRIGGER killaccounts
ON Branch
FOR delete
AS
DELETE Account
FROM Account, deleted
WHERE Account.branch_ID = deleted.branch_ID
```

- The third use of a trigger is to respond to events generated by a collection of applications. In this case, triggers can provide substantial performance benefits. Consider an application that must write into a table Monitortable the latest data inserted into a table Interestingtable. Without triggers the only way to obtain this data is to query the table repeatedly, that is, to *poll*. The query might be of the form

```
INSERT Monitortable
SELECT *
FROM Interestingtable
WHERE Inserttime > @lasttimeIlooked
Update @lasttimeIlooked based on current time.
```

Polling introduces a tension between correctness and performance. If your application polls too often, then there will be unnecessary queries in addition to concurrency conflicts with inserters to Interestingtable. If your application polls too seldom, you may miss important data (perhaps some transactions delete recently inserted records from Interestingtable).
A better approach (known as an *interrupt-driven* approach) is to use a trigger to put the data directly into Monitortable when an insertion occurs. The trigger might have the form

```
CREATE TRIGGER tomonitor
ON Interestingtable
```

```
FOR insert
AS
INSERT Monitortable
SELECT *
FROM inserted
```

No tuning is necessary, and no inserts will be missed.

Triggers do have one serious disadvantage: they can lead to a maintenance nightmare for two related reasons.

1. In the presence of triggers, no update can be analyzed in isolation because the update may cause some trigger to cause a further modification.
2. The interaction between triggers can be confusing once the number of triggers becomes large. This is an issue because several triggers may be enabled by the same modification. The system must choose which one to execute first and whether to execute the others at all. (A system could, in principle, execute all enabled triggers simultaneously, but it is not clear what that would mean if several of the triggers modify the same data.) A useful software tool would alert system administrators to possible conflicts among triggers as well as possible cascades of different triggers.

4.7.2 Trigger Performance

For most purposes, improving a trigger's performance entails the same analysis as improving any query's performance. For example, if the trigger updates the branch balance each time an account balance is updated, then you might consider an index on branch ID. Trigger performance tuning presents two unique aspects, however.

- A trigger occurs only after a certain combination of operations. Sometimes, you can make use of that information to suggest a special strategy. For example, suppose every update to the Sale relation requires a check on whether the customer credit is good. It may, then, be a good idea to create an index on the Customer table to ensure that the resulting join is fast enough.
- A naive application developer may often write a trigger that executes too often or returns too much data. For example, consider a trigger that writes records to table Richdepositor each time an account balance increases over $50,000.

```
CREATE TRIGGER nouveauriche
ON Account
FOR update
AS
BEGIN
  INSERT Richdepositor
  FROM inserted
  WHERE inserted.balance > 50000
END
```

Writing the trigger this way has two problems. First, it will try to write into Richdepositor whether or not a balance has been updated. Second, suppose that some depositor Ariana Carlin already had a balance greater than $50,000 before the update took place. This trigger will then write Ariana Carlin into the Richdepositor table unnecessarily. It would be better to write the trigger this way:

```
CREATE TRIGGER nouveauriche
ON Account
FOR update
AS
IF update(balance)
BEGIN
  INSERT Richdepositor
  FROM inserted, deleted
  WHERE inserted.balance > 50000
  AND deleted.balance < 50000
  AND deleted.account_ID = inserted.account_ID
END
```

Bibliography

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995. This is the classic book for the theory underlying good database design.

Joe Celko. *SQL for Smarties: Advanced SQL Programming.* Morgan Kaufmann, 2000. This book contains case studies of SQL programming techniques presented as puzzles.

Ceptsar A. Galindo-Legaria and Milind Joshi. *Orthogonal optimization of subqueries and aggregation.* In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, Calif., May 21–24, 2001, 571–581. ACM Press, 2001. This paper describes the techniques implemented in SQL Server 7 for optimizing correlated subqueries and aggregates.

Richard A. Ganski and Harry K. T. Wong. *Optimization of nested SQL queries revisited.* In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference*, San Francisco, May 27–29, 1987, 23–33. ACM Press, 1987. This paper describes the infamous count bug (involving the transformation of counting aggregates in the presence of empty sets).

Won Kim. *On optimizing an SQL-like nested query. TODS*, 7(3):443–469, 1982. Original paper on nested subqueries optimization.

Exercises

EXERCISE 1

An airline database stores information about airplanes and flights as follows:

- Airplane(airplane_ID, dateofpurchase, model), with key airplane_ID

Flight(airplane_ID, flight, seat, occupant), with key airplane_ID, flight, seat

Would there be any performance-related advantage to denormalizing to

- (airplane_ID, flight, seat, occupant, dateofpurchase, flight, model)

for the online portion of the database?
Action. No. Denormalizing is unlikely to help. The number of updates on flights is likely to be much higher than queries relating specific flight-seat pairs to the date of purchase.

EXERCISE 2

Suppose you are given the relation

- Purchase(name, item, price, quantity, suppliername, year)

with a clustering index on name. You want to compute the cost of items based on a first-in, first-out ordering. That is, the cost of the first purchase of an item should be accounted for before the cost of a later item. Consider Figure 4.11, which contains all the widget records in the Purchase relation. To account for the purchase of 98 widgets, we should account for 60 from the year 2001 (costing $900) and 38 from the year 2002 (costing $494) for a total cost of $1394.

NAME	ITEM	PRICE	QUANTITY	SUPPLIERNAME	YEAR
Widget	4325	15	60	Standard Widget	2001
Widget	5925	13	60	Standard Widget	2002
Widget	6324	17	54	Standard Widget	2003
Woosit	3724	15	80	Standard Widget	2001

Figure 4.11: Purchase table.

We want to do this for all the data items. So, for each such data item @x, we return the data sorted by year.

```
SELECT *
FROM Purchase
WHERE item = @x
ORDER BY year
```

There are 10,000 different items, and we discover that each query scans the entire Purchase table and then performs a sort based on year. The application runs too slowly. What do you do?
Action. If there are fairly few records for each item, then one possibility is to construct a nonclustering index on item. The query style won't have to change. Another strategy is to sort all the records into a temporary by (item, year). Then go through the table sequentially using your favorite programming language.

[8]We discuss the performance uses of stored procedures in the context of reducing unnecessary interactions between clients and servers in the next chapter.

Chapter 5: Communicating With the Outside

5.1 Talking to the World

Databases must communicate with programming languages that do specialized manipulations, with outside data sources, and with application servers.

In most settings, the underlying hardware can make this communication fast, but poor use of the software interface tools can slow your system down significantly. This chapter suggests methods to make the interfaces fast.

Most applications interact with database systems in one of two ways: by using a system-specific fourth-generation language(4GL) or by using a programming language (such as Java, Visual Basic, or C) that relies on a call-level interface.

Different vendors offer different 4GLs, so using a 4GL hinders portability, but they execute at a high level so often result in small programs. Call-level interfaces are libraries of functions that enable a program to connect to a database server, run SQL statements, fetch query results, and commit or abort transactions. There are a myriad of call-level interfaces. The most popular are ODBC for C/C++ and JDBC for Java.

ODBC (Open DataBase Connectivity) was originally designed by Microsoft; it has become a standard interface for database systems on Windows platforms (Oracle and DB2 provide ODBC drivers for Windows).[1] JDBC is a call-level interface for Java that was defined by Sun. JDBC is very similar to ODBC in its architecture and in the functions it supports; we thus focus on ODBC in the rest of this section.

An application using the ODBC interface relies on a driver manager (integrated with Windows) to load a driver that is responsible for the interaction with the target database system. The driver implements the ODBC functions and interacts with the underlying database server. The driver manager can also be configured to log all ODBC calls. This is useful for debugging but hurts performances, so should be disabled in production environments.
SQL Server, Oracle, and DB2 UDB servers all have specific ODBC drivers. Companies such as DataDirect Technologies[2] or OpenLink [3] provide *generic* ODBC drivers. ODBC drivers can be characterized as follows:

1. *Conformance level.* For portable applications, it is safe to limit the calls to level 1 ODBC functions (e.g., core functions for allocating and deallocating handles for database environment, connections, and statements; for connecting to a database server; for executing SQL statements; for receiving results; for controlling transactions; and for error handling).
2. *SQL dialect.* Generic ODBC drivers transform SQL statements that follow ODBC's definition of the SQL grammar into the dialect of a particular database system. System-specific ODBC drivers assume that SQL statements follow the dialect of the system they are connected to.
3. *Client-server communication mechanism.* Generic ODBC drivers rely on a database-independent communication layer, whereas database-specific ODBC drivers exploit the particular mechanisms supported by the database server they connect to (net8 for Oracle 8, ODS for SQL Server, and Client Application Enabler for DB2 UDB). Database-specific ODBC drivers can run significantly faster as a result.
Generic ODBC drivers provide transparent portability but worse performance and some loss of features (level 1 conformance in general). Database-specific ODBC drivers allow a program to run on different database systems as long as queries are expressed in the dialect of the target system. If portability is not an issue, or if a program needs to access specific functions of a given database system that are not supported by ODBC (most ODBC drivers do not support bulk load, for instance), then a programmer should consider using a native call-level interface

to access a specific database system (CLI for DB2 UDB and OCI for Oracle). Figure 5.1 compares the performance of the ODBC driver and of the native call-level interface for Oracle 8i EE. The native call-level interface is the most frequently chosen option for C/C++ programs on Unix platforms.

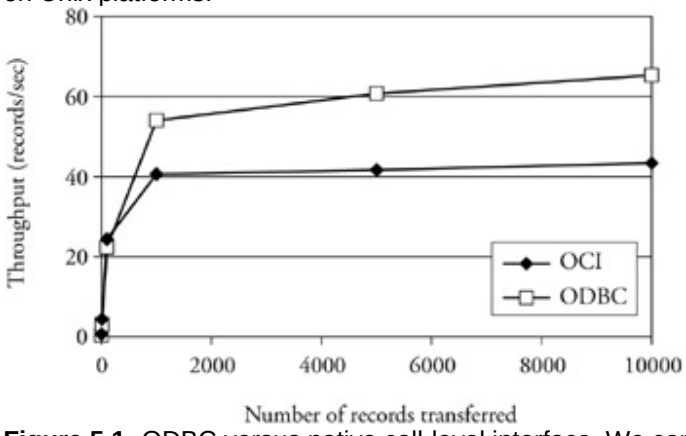


Figure 5.1: ODBC versus native call-level interface. We compare the throughput obtained using ODBC and OCI (the Oracle call-level interface) to retrieve records from the database server into the application. Note that the ODBC driver we use for this experiment is implemented on top of OCI. The results show (1) that the connection and query preparation overhead is much lower with OCI than ODBC, almost twice as low, and (2) that the fetching overhead is lower for ODBC than OCI. When the number of records fetched from the result set increases, the throughput increases more with ODBC than with OCI. The reason is that the ODBC driver does a good job at implicitly prefetching records compared to our straightforward use of OCI.

[1]As fo mid-2001, ODBC drivers are available for most database systems on Unix platforms.

[2]http://www.datadirect-technologies.com

[3]http://www.openlinksw.com

5.2 Client-Server Mechanisms

The database server produces results that are consumed by the application client. Because the client is asynchronous with respect to the server, this communication is mediated by a buffer, usually one per connection (Figure 5.2). There are two main performance issues related to this client-server mechanism:

- If a client does not consume results fast enough, then the server will stop producing results and hold resources such as locks until it can output the complete result to the query. Thus a client program executing on a very slow machine can affect the performance of the whole database server. This is one motivation for creating three-tier applications: the middle tier takes the database results fast and buffers them for a potentially slow outer tier.
- Data is sent from the server to the client either when the communication buffer on the server side is full or when a batch (one or several queries and stored procedures) is finished executing. A large communication buffer on the server side thus will delay the first rows transmitted to the client (lengthening the perceived response time). A small buffer entails traversing the network stack more frequently when transferring large volumes of data. Finally, the buffer should fit into a frame of the underlying transport layer in order to avoid unnecessary fragmentation overhead. The maximum transmission unit for TCP, for instance, is 1500 bytes, so in order to avoid unnecessary fragmentation the size of the communication buffer added to the header introduced by the communication layer should be a multiple of 1500 on an Ethernet network. Fortunately, sufficient network bandwidth

makes most of these considerations irrelevant. In our experiments, varying the server side buffer or the packet size had *no effect* on a 100-megabit network. If you have a slower network or a lot of contention on your fast network, then please consider using these experiment scripts to guide your specification of server side buffer and packet size.

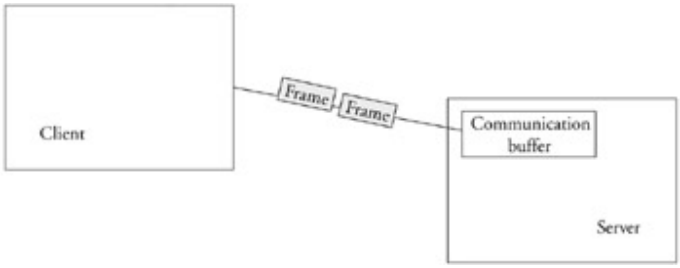


Figure 5.2: Client-server connection. Client-server communication is mediated by a buffer on the server site, usually one per connection.

5.3 Objects, Application Tools, and Performance

Every software developer has experienced the scenario: an innocent design decision brings performance to its knees. The difference can be the refactoring of an object-oriented design over a backend database or the use of an application development tool for a tangential application. Here are a few instructive examples.

5.3.1 Beware of Small Objects

Object-oriented databases have never achieved wide acceptance because object-relational systems stole most of their comparative advantage. Nevertheless, object-oriented programming languages such as C++ and Java have captured a large portion of the application development market. See the box for a quick review of object orientation.

BASIC CONCEPTS OF OBJECT ORIENTATION

An *object* is a collection of data, an identity, and a set of operations, sometimes called *methods*. For example, a newspaper article object may contain zero or more photographs as constituent objects and a text value. Typical operations will be to edit the article and display it at a certain width: *edit(), display(width)*. The system prevents programs using that object (perhaps with the exception of a privileged few) from accessing the internal data (e.g., the text) of the object directly. Instead, such programs must access the operations—or, in object orientese, "invoke the methods"—of the object (e.g., edit). Hiding the representation of an object is known as *encapsulation*.

A *class* is the definition of a data description (analogous to the relational notion of schema) and operations that will characterize a set of objects. For example, there may be a newspaper article class whose data description would consist of text, and a set of photographs whose operation description would consist of the code for edit and display. There may be many objects belonging to class newspaper article, each denoting a different article.

Finally, many object-oriented systems have a concept of *inheritance* that permits class X to derive much of its code and attributes from another class Y. Class X will contain the data attributes and operations of class Y plus additional ones. For example, there may be a sports article class that derives its data description and operations from newspaper article, but then adds a table (of scores) and the operations *enlarge headline* and *display score*.

In summary, an object-oriented programming language represents pieces of data called objects. Objects belong to classes that define the operations on those objects. Classes are related to one another through an inheritance hierarchy.

Object-oriented encapsulation allows the implementation of one class to be modified without affecting the rest of the application, thus contributing greatly to code maintenance. Encapsulation sometimes is interpreted as "the specification is all that counts," however, and then can cause terrible harm to performance. The problem begins with the fact that the most natural object-oriented design on top of a relational database is to make records (or even fields) into objects. Fetching one of these objects then translates to a fetch of a record or a field. So far, so good.

But then the temptation is to build bulk fetches from fetches on little objects. The net result is a proliferation of small queries instead of one large query.

Consider, for example, a system that delivers and stores documents. Each document type (e.g., a report on a customer account) is produced according to a certain schedule that may differ from one document type to another. Authorization information relates document types to users. This gives a set of tables of the form

```
authorized(user, documenttype)
documentinstance(id, documenttype, documentdate)
```

When a user logs in, the user wants to know which document instances the user can see. This can easily be done with the join

```
select documentinstance.id, documentinstance.documentdate
from documentinstance, authorized
where documentinstance.documenttype = authorized.documenttype
and authorized.user = <input>
```

But if each document type is an object and each document instance is another object, then you may be tempted to issue one query to find all the document types for the user:

```
select documentinstance.documenttype
from authorized
where authorized.user = <input>
```

and then for each such type `t` to issue the query

```
select documentinstance.id, documentinstance.documentdate
from documentinstance
where documentinstance.documenttype = t
```

This is much slower. The join is performed in the application and not in the database server.

The point is not that object orientation is bad. Encapsulation contributes to maintainability. The point is that programmers should keep their minds open to the possibility that accessing a bulk object (e.g., a collection of documents) should be done directly rather than by forming the member objects individually and then grouping them into a bulk object on the application side.

5.3.2 Beware of Application Development Tools

Some application development packages get in the way of good performance for other reasons. Judy Smith, a DBA, reports the following: "The application group uses an application toolkit that encapsulates stored procedures rather than makes direct sql calls. Performance was quite poor. However, performance was quite acceptable when the underlying stored procedure was

run using a command-line tool (isql). The Unix 'snoop' command captured the packets passed to the server via the toolkit. It turned out that toolkit would fetch the metadata about a stored procedure prior to calling it. That is, it would fetch each column name and column type (whether or not the default was to be used) that the stored procedure had as a parameter. This was happening on every call creating additional sql with large parameter lists. Turning off this feature helped substantially. As a result of such situations, I have gotten into the practice of running ‘suspect’ sql directly on the machine where the database engine is running without using layered interfaces. This helps to determine if it is really the sql, and not factors such as the network or the API, that is causing the performance problem."

5.4 Tuning the Application Interface

Poorly written applications can result in poor performance. Here are some principles. Justification and experiments follow.

1. Avoid user interaction within a transaction.
2. Minimize the number of round-trips between the application and the database server.
3. Retrieve needed columns only.
4. Retrieve needed rows only.
5. Minimize the number of query compilations.
6. Consider large granularity locks.

5.4.1 Avoid User Interaction Within a Transaction

As discussed in the concurrency control section of Chapter 2, a transaction should not encapsulate user interactions (e.g., manual screen updates) because such interactions lengthen the transaction significantly, thereby causing concurrency problems. Imagine, for example, that the operator decides to go to lunch while holding important locks. To avoid such nightmares, the application should divide an update into a read transaction, a local update (outside transactional boundaries), and a write transaction. See Appendix B on transaction chopping to understand when it is possible to do this while maintaining correctness.

5.4.2 Minimize the Number of Round-Trips Between the Application and the Database Server

Crossing the interface between the application and the database server is costly. Here are some techniques to minimize the number of crossings.

1. Application programming languages offer looping facilities. Embedding an SQL SELECT statement inside the loop means that there will be application-to-database interaction in every iteration of the loop. A far better idea is to retrieve a significant amount of data outside the loop and then to use the loop to process the data.
Along the same lines, it is a far better idea to gather all insertions into one statement rather than issuing an INSERT statement for each row to be inserted. Figure 5.3 illustrates this point.

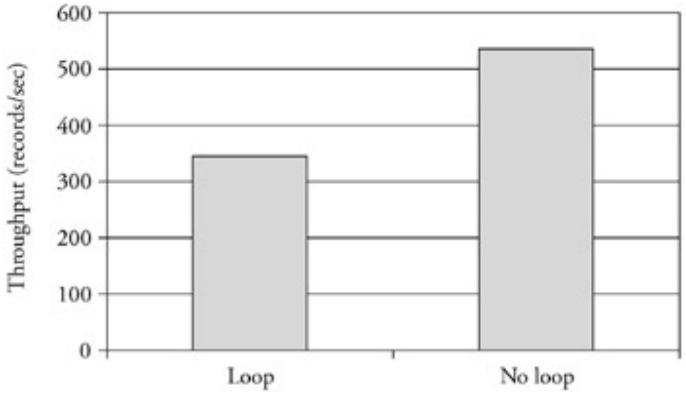


Figure 5.3: Loop constructs. This graph compares two programs that obtain 2000 records from a large table (lineitem from TPC-H). The *loop* program submits 200 queries to obtain this data, whereas the no loop program submits only one query and thus displays much better performance. This graph was obtained using SQL Server 2000 on Windows 2000.

2. Because interactions between a conventional programming language and the database management system are expensive, a good tuning strategy is to package a number of SQL statements into one interaction. The embedded procedural language that many systems offer includes control flow facilities such as if statements, while loops, goto's, and exceptions.

   Here is an example in Transact SQL-like syntax that determines whether Carol Diane is an ancestor of Nicholas Bennet. For the purposes of the example, assume a genealogical database containing at least the relation Parental(parent, child).

```
create table Temp1 (parent varchar(200))
create table Temp2 (parent varchar(200))
create table Ancestor (person varchar(200))
/* Temp2 will hold the latest generation discovered. */
INSERT INTO Temp1
SELECT parent

FROM Parental
WHERE child = 'Nicholas Bennet';
WHILE EXISTS(SELECT * FROM Temp1)
BEGIN
  INSERT INTO Ancestor
  SELECT * FROM Temp1;

  INSERT INTO Temp2
  SELECT * FROM Temp1;

DELETE FROM Temp1;

INSERT INTO Temp1
SELECT Parental.parent
```

```
FROM Parental, Temp2
WHERE Parental.child = Temp2.parent;

DELETE FROM Temp2;

END
IF EXISTS (
    SELECT *

    FROM Ancestor
    WHERE person = 'Carol Diane'
  )
  PRINT 'Carol Diane is an ancestor of Nicholas Bennet.'
ELSE
  PRINT 'Carol Diane is not an ancestor of Nicholas Bennet.'
```

Using the embedded procedural language reduces the number of calls from the application program to the database management system, saving significant overhead.

The main disadvantage is that many such languages are product specific, so they can reduce the portability of your application code.

3. Object-relational systems such as SQL Server 2000, Oracle 8i (and higher), and DB2 UDB V7.1 (and higher) support user-defined functions (UDFs). Scalar UDFs can be integrated within a query condition: they are part of the execution plan and are executed on the database server. UDFs are thus a great tool to reduce the number of round-trips between the application and the database system. Figure 5.4 illustrates the benefits of user-defined functions.

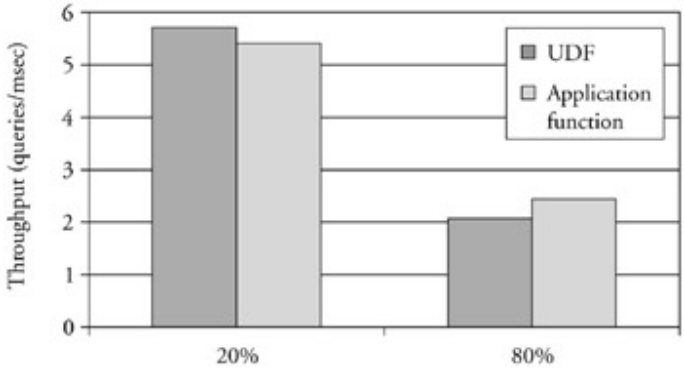


Figure 5.4: User-defined functions. We compare processing a function on the client site (retrieving all data + executing the function) with executing a function as a UDF within a query. The function computes the number of working days between two dates; the query selects the records in the lineitem table where the number of working days between the date of shipping and the data the receipt was sent is greater than five working days (80% of the records) or smaller than five working days (20% of the records). Using the UDF reduces the amount of data transferred, but applying the function at the application level happens to be faster when there are many records. This graph was obtained using SQL Server 2000 on Windows 2000.

4. Call-level interfaces such as ODBC allow so-called positioned updates. A positioned update consists in updating the rows that are obtained as the result of a query. The application thus iterates over a set of rows and updates each row in turn. This forces

the processing of updates one row at a time. This might be required in some cases, but unsophisticated programmers might be tempted to overuse this feature. In fact, set-oriented update statements should be used as much as possible in order to minimize the number of round-trips between the application and the database server.

5.4.3 Retrieve Needed Columns Only

There are two reasons why this is usually a good idea, one obvious and one less so.

1. The obvious reason is that retrieving an unneeded column will cause unnecessary data to be transferred.
2. A subtle reason is that retrieving an unneeded column may prevent a query from being answered within (being covered by) the index (Figure 5.5). For example, if there is a dense composite index on last name and first name, then a query that asks for all the first names of people whose last name is ‘Codd’ can be answered by the index alone, provided no irrelevant attributes are also selected.

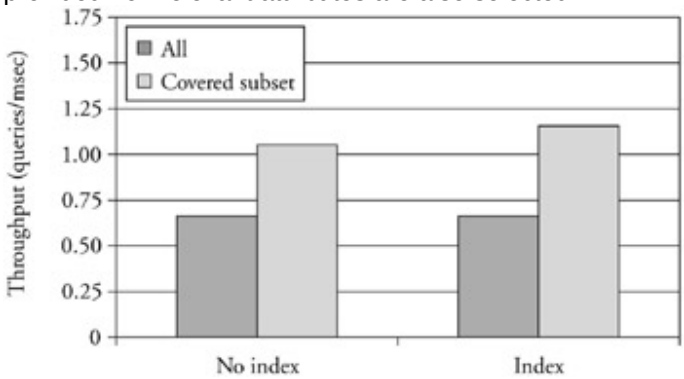


Figure 5.5: Retrieve needed columns only. This graph illustrates the impact on performance of retrieving a subset of the columns as opposed to retrieving all columns. In this experiment, we compare retrieving one-fourth of the columns with retrieving all columns using *select* *. We performed this experiment in two situations: (1) without indexes and (2) with a nonclustering index covering the projected columns. The overhead of crossing the database interface with larger amounts of data is significant. Using covered indexes yields an additional performance boost to the carefully written query. This experiment was run on Oracle 8i on Windows 2000.

5.4.4 Retrieve Needed Rows Only

Here is how to minimize the amount of data that crosses the interface between the application and the database server.

1. In case users end up viewing only a small subset of a very large result set, it is only common sense to transfer just the subset of interest.[4] Sometimes the subset is just a few rows of a larger selection. The user just wants a "feel" for the data in the same way that Web users want a "feel" for the responses to a search engine query. In that case, we recommend that you use a construct such as TOP or FETCH FIRST to fetch just a few rows.
You should avoid cursors, however, since they are terribly slow in almost all systems (Shasha once had the experience of rewriting an 8-hour query having nested cursors into a cursor-free query that took 15 seconds). We illustrate this with an experiment (Figure 5.6).

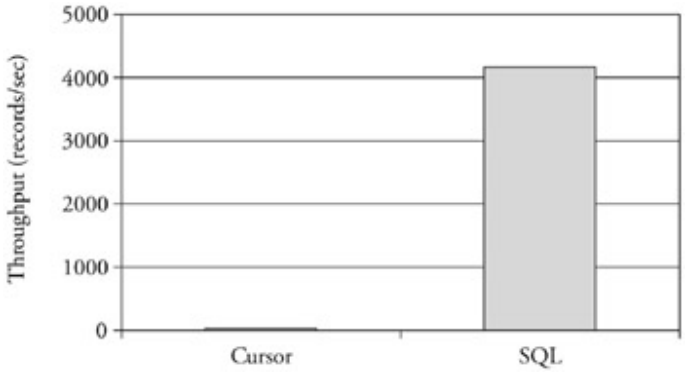


Figure 5.6: Beware of cursors. This experiment consists in retrieving 200,000 rows from the table Employee (each record is 56 bytes) using a set-oriented formulation (SQL) or a cursor to iterate over the table contents (cursor). Using the cursor, records are transmitted from the database server to the application one at a time. This has a very significant impact on performance. The query takes a few seconds with the SQL formulation and more than an hour using a cursor. This experiment was run on SQL Server 2000 on Windows 2000.

2. Applications that support the formulation of ad hoc queries should permit users to cancel those queries. Query cancellation avoids holding resources when a user realizes that he or she made a mistake in the query formulation or when query execution takes unusually long to terminate. Query cancellation can be implemented by maintaining in the application a supervisor thread that signals the querying thread to stop.

5.4.5 Minimize the Number of Query Compilations

Query parsing and optimization is a form of overhead to avoid. There are two aspects to this cost.

1. The compilation of simple queries requires from 10,000 to 30,000 instructions; compiling a complicated query may require from 100,000 to several million instructions. If this seems surprisingly high, remember that compilation requires parsing, semantic analysis, the verification of access privileges, and optimization. In fact, for simple queries that use indexes, compilation time can exceed execution time by a factor of three or more.
2. Compilation requires read access to the system catalog. This can cause lock-induced blocking if other transactions modify the catalog concurrently.

For these reasons, a well-tuned online relational environment should rarely if ever perform query compilations. Moreover, the *query plan* resulting from a compilation (telling which indexes will be used and which tables will be scanned) should be rapidly accessible to the query processor. All systems offer a procedure cache for this purpose. You should make sure that your cache is big enough for frequently executed queries. Figure 5.7 illustrates the benefits of precompiled queries.

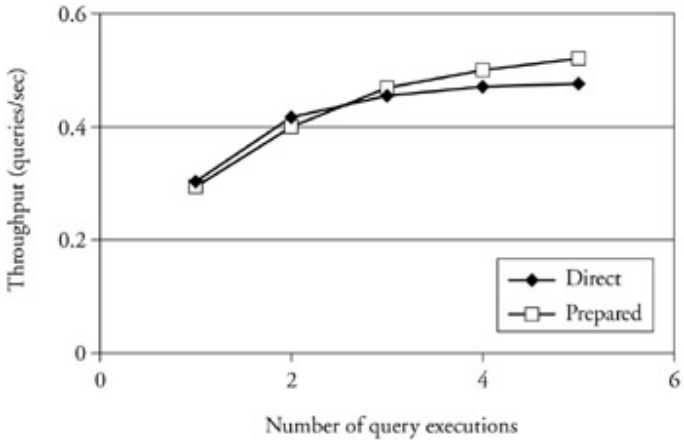


Figure 5.7: Benefits of precompiled queries. This graph illustrates a benefit of precompiled queries. We have run a simple query (uncorrelated subquery without aggregate) several times either by submitting the query each time (using ODBC direct execution) or by compiling it once (using ODBC prepare command) and executing it repeatedly. The results show that precompilation is advantageous when the query is executed more than twice.

Optimal query plans may become suboptimal if the relations upon which they work change significantly. Here are some changes that should induce you to recompile a query (after you update the catalog statistics):

- An index is added on an attribute that is important to the query. The query should be recompiled, so the new query plan will take advantage of this index. Some systems do this automatically.
- A relation grows from nearly empty to a significant size. The query plan for the nearly empty relation will not use any index on the grounds that it is cheaper to scan a nearly empty relation than to use an index to access it. So, once the relation grows, the query should be recompiled so it will use the index.

Defining a stored procedure is a way to precompile a query. Using a call-level interface, it is also possible to prepare a query and execute it several times. Note that SQL Server version 7 and higher maintains a cache of execution plans for all queries that are asked; as a result all already submitted queries (whose plan fits in the cache) can be considered precompiled.

[4]For a psychological reason whose roots in early childhood remain mysterious, Java and C++ programmers seem to prefer to pull everything from the database and then perform selections in a loop of their favorite language. We call this disease *loopphlia.* If you discover this, then the proper bedside manner is to ask in a nonthreatening way: "Could it be that your application throws away most of what it receives from the database? Is there any restriction we could add to the where clause to bring back only what you need?"

5.5 Bulk Loading Data

The partitioning principle suggests loading large volumes of data into a database while there are no applications accessing it. In most organizations, this state of affairs holds at night. In that case, the loading process operates under a time constraint: it has to be finished before database users come back to work in the morning.

Database systems provide bulk loading tools that achieve high performance by bypassing some of the database layers that would be traversed if single row INSERT statements were used.

- SQL Server 7 provides a bulk loading tool called bcp and a Transact-SQL command called BULK INSERT. The T-SQL command is directly run within the SQL engine and bypasses the communication layer of SQL Server 7. BULK INSERT is therefore much more efficient when loading data from a local disk. BULK INSERT can also be configured to avoid logging so that rows are loaded full extent at a time instead of row at a time.
- SQL * Loader is a tool that bulk loads data into Oracle databases. It can be configured to bypass the query engine of the database server (using the direct path option). SQL * Loader can also be configured so that it does not update indexes while loading the data (SKIP_INDEX_MAINTENANCE option). Deferring index updates to the end is often much faster.
- The Load utility from DB2 UDB loads data into a table one page at a time and deactivates triggers and constraints while the data is being loaded (the database administrator has to check which constraints have been violated after data has been loaded). The DB2 loading utility can be configured to skip the collection of statistics.

An important factor influencing the performance of a bulk loading tool is the number of rows that are loaded in each transaction as a batch. The DB2 loading utility loads rows one page at at time. The SQL Server BULK INSERT command and SQL*Loader allow the user to define the number of rows per batch or the number of kilobytes per batch. The minimum of both is used to determine how many rows are loaded in each batch. There is a trade-off between the performance gained by minimizing the transaction overhead and the work that has to be redone in case a failure occurs. Figure 5.8 illustrates the performance benefits of large batches, Figure 5.9 illustrates the benefits of bypassing the SQL engine, and Figure 5.10 illustrates the impact of various storage engine parameters on load performance.

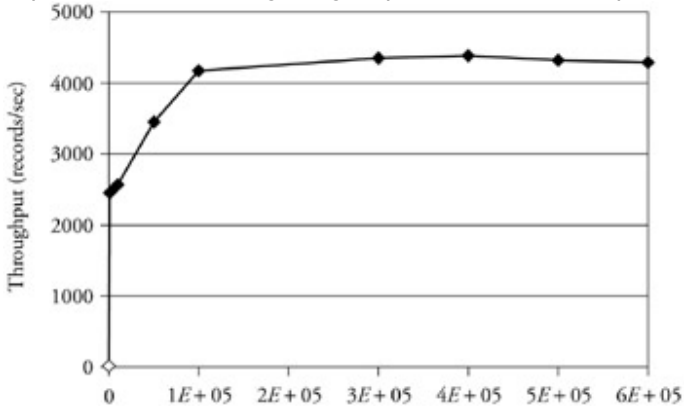


Figure 5.8: Batch size. This graph shows the influence of the batch size on performance. We used the BULK INSERT command to load 600,500 tuples into the lineitem relation on SQL Server 2000 on Windows 2000. We varied the number of tuples loaded in each batch. The graph shows that throughput increases steadily until batch size reaches 100,000 tuples, after which there seems to be no further gain. This suggests that a satisfactory trade-off can be found between performance (the larger the batches the better up to a certain point) and the amount of data that has to be reloaded in case of a problem when loading a batch (the smaller the batches the better).

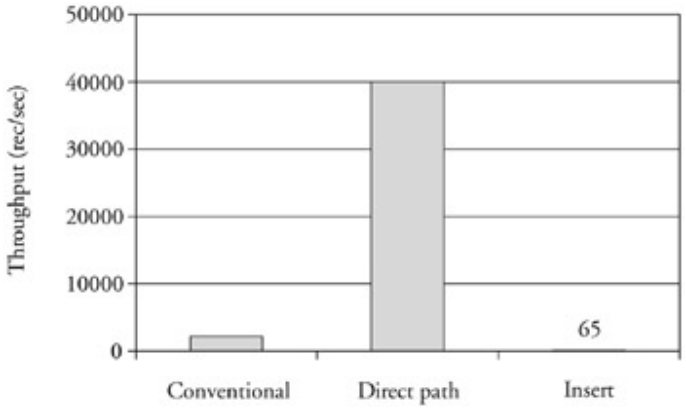


Figure 5.9: Direct path. This graph illustrates the performance benefits obtained by bypassing the SQL engine (conventional usage of SQL * Loader with a commit every 100 records) and the storage manager (direct path option of SQL * Loader) compared to the performance of inserts (using one thread and a commit after each insertion). These results were obtained by inserting 600,500 tuples into the lineitem relation on Oracle 8i on Windows 2000.

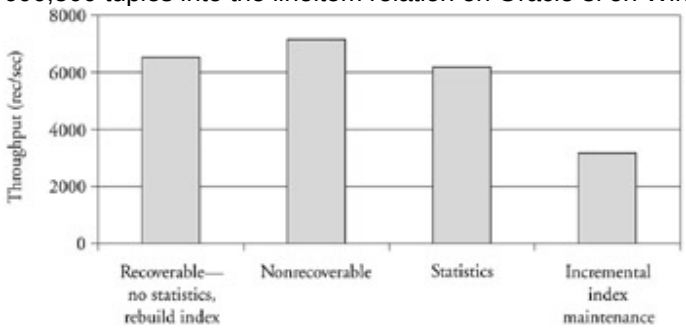


Figure 5.10: Storage engine parameters. This graph illustrates the influence of three parameters on the performance of the DB2 UDB data loading utility. We first loaded 600,500 records in the lineitem relation into DB2 UDB V7.1 on Windows 2000 using the recoverable option (before images are maintained so that the original relation can be restored), no statistics were collected, and a clustering index was rebuilt after data was loaded. We varied in turn each of these parameters. As expected, performing a nonrecoverable load increases throughput, whereas collecting statistics decreases throughput. The impact of these parameters on performance is, however, not dramatic. Incremental index maintenance (as opposed to rebuilding the index after the load has terminated) decreases throughput significantly.

Bulk loading tools allow an application to transform the format of the inserted data by executing SQL functions before actually inserting data into the tables. If such functions are used during the load, then the SQL engine cannot be bypassed. A better alternative is to perform the bulk load without changing the format and then to issue a query to create a new format.

5.6 Accessing Multiple Databases

As companies acquire new database servers or are reorganized (say, after a merger), they have to deal with various legacy systems. As a consequence, they create data warehouses from disparate data sources following a more or less honest attempt at integration. Whereas the problem of integration lies outside the scope of this book, you should know that there is a never-ending demand for reconciling databases, entailing data cleaning and semantic reconcilation. Data cleaning is needed, for example, to unify two databases based on names and addresses—rarely are formats or spellings consistent. Semantic reconcilation is required when two databases have different understandings of who an employee is; perhaps the organizational chart database views consultants as employees, whereas the human resources database does not.

When forming a data warehouse costs too much, organizations must allow business applications to access multiple heterogeneous databases. The goal is to make this reasonably easy and efficient.

Modern database servers provide transparent access to external relational or nonrelational data sources (generic connectivity and transparent gateways in Oracle 8i and higher versions, IBM DataJoiner or DB2 relational connect in DB2 UDB V7.2, and linked servers in SQL Server 7.0 and higher). They support

- *distribution transparency*. The client-server mechanisms normally used between an application and the database server are reused to connect the database server with external data sources. The database server executes SQL statements on the external data source and obtains result sets. Because these client-server mechanisms are often not standard, extra components are needed as interfaces between the client-server layer and the remote sources. These extra components either are encapsulated within ODBC drivers (Oracle generic connectivity or SQL Server linked servers) or are specially tailored for popular data sources (i.e., Oracle open gateways or DB2 relational connect).
- *heterogeneity transparency*. The database server acts as a frontend to the remote sources. It translates the queries submitted by the application into the dialect of the external data sources and performs postprocessing in case the external data source has limited capabilities (e.g., the database server may perform time or math functions).

Performance and tuning considerations are similar to those for application interfaces as described earlier. Here are a few differences.

- *Use shared connections to reduce start-up cost*. Connections to the remote data sources are implicit. In Oracle, for instance, a connection is established the first time a remote data source is contacted by an application; the connection remains open until the application disconnects from the database server. Oracle allows a set of database server processes to share connections. The benefit of shared connections is that existing connections can be reused thus avoiding the start-up cost of connection establishment. The disadvantage is that there might be conflicts accessing the shared connection if several processes submit statements to the same external data sources at the same time. Thus individual connections should be chosen when you expect a lot of concurrent access across the connections.
- *Use pass-through statements when performance is CPU bound*. All interconnection mechanisms support *pass-through* SQL statements, written in the dialect of an external data source. Pass-through statements are submitted to an external data source without any translation or postprocessing. These pass-through statements break the heterogeneity transparency, but they can improve performance if the translation of the result turns out to be time consuming or if the external data source has some special capability (e.g., a time series accelerator).
- *Transfer larger blocks of data when performance is network bound*. The interface components between the client-server mechanism and the external data sources control the size of the data being transferred when returning a result set. This parameter should be tuned to minimize the number of round-trips between the database server and the remote data source. In particular, a larger block size should be favored when large volumes of data are transferred.

Bibliography

Kalen Delaney. *Inside MS SQL Server 2000.* Microsoft Press, 2000. This book is a revision of the SQL Server 7 edition.

Jason Durbin and Lance Ashdown. *Oracle8i Distributed Database Systems, Release 2 (8.1.6).* Oracle Corporation, 1999. This document describes the Oracle heterogeneous services for accessing multiple database systems.

IBM. *DB2 UDB Administration Guide Version 7.* IBM Corporation, 2000. This document covers installation and planning issues for IBM DB2. In particular, it covers issues concerning the application interface and access to multiple databases.

Kevin Loney and Marlene Thierault. *Oracle9i DBA Handbook.* Oracle Press, 2001.

Richard Niemiec. *Oracle8i Performance Tuning.* Oracle Press, 1999. A section of this book covers the tuning of JDBC applications. These principles apply to call-level interfaces in general.

Ron Soukup and Kalen Delaney. *Inside MS SQL Server 7.0.* Microsoft Press, 1999. A section of this book covers the details of the SQL Server application interface using ODBC, OLE DB, or ADO.

Chapter 6: Case Studies From Wall Street

One reason we believe expert human database tuners will always be in demand is that many of the biggest performance improvements come from viewing an application in a new light. Tools can, of course, help, but they would have to confront an enormous variety of situations, some of which we outline here.

Many of these examples come from our work on Wall Street, but we don't think the lessons are specific to that environment. For the sake of concreteness, however, we have left the context intact.

6.1 Techniques for Circumventing Superlinearity

As in many other applications, financial data often must be validated when it enters. This often happens in the context of a data warehouse, slet us study such a situation. You will see that small rewritings can yield big results (Figure 6.1).

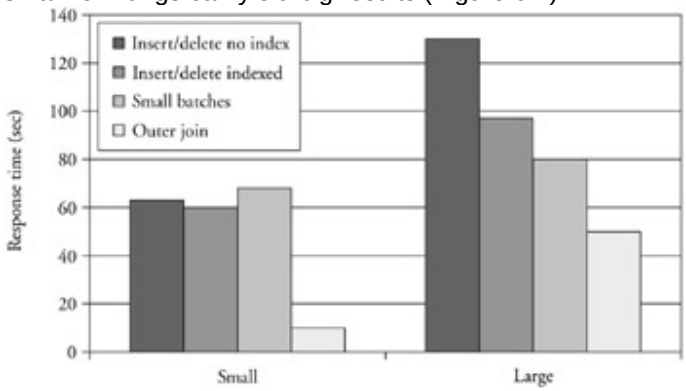


Figure 6.1: Circumventing superlinearity. This graph compares the four techniques that we describe for circumventing superlinearity: (a) insertion followed by a check for deletions, (b) same as (a) with an index on the table used to check for deletions, (c) inserting sales and checking for deletions in small batches, and (d) using outer join. We use the unsuccessful sales example given in the text with two configurations of the data: small (500,000 sales, 400,000 items, 400,000 customers and 10,000 stores, and 400,000 successful sales) and large (1,000,000 sales, 800,000 items, same customers and stores tables as for the small workload, and around 800,000 successful sales). The experiment is performed using SQL Server 2000 on Windows 2000. Using the small workload, the minibatch approach does not provide any benefit. Indeed, the successfulsales table is small enough so that the overhead of the successive iterations (insertions/deletions in a temporary table) is high compared to the benefit of checking the deletion condition on a reduced number of records. There is a benefit in using an index on successful sales (SQL Server 2000 uses an index nested loop in that case instead of a hash join in the absence of an index). The outer join approach is very efficient because the detection of unsuccessful sales is performed using a selection (itemtest is null, or storetest is null, or customertest is null) as opposed to a join with the successfulsales table. The large workload illustrates the benefit of the batch approach. The outer join approach still gives the best performance.

There is a fact table

sales(id, itemid, customerid, storeid, amount, quantity)

The itemid field must be checked against the item table, the customerid field against the customer table, and the storeid field against the store table. With dense indexes on each of the dimension tables item, customer, and store, this can be done in linear time without even touching those tables.

Create an empty table.

successfulsales(id, itemid, customerid, storeid, amount, quantity)

Then issue the query.

```
insert successfulsales
select *
from sales
where itemid in (select itemid from item)

and customerid in (select customerid from customer)
and storeid in (select storeid from store)
```

Depending on your system, this may be faster or slower than

```
insert successfulsales
select sales.id, sales.itemid, sales.customerid, sales.storeid,
sales.amount, sales.quantity
from sales, item, customer, store
where sales.itemid = item.itemid
and sales.customerid = customer.customerid
and sales.storeid = store.storeid
```

Now comes the difficult part. We want to create a table unsuccessfulsales that did not make it into successfulsales possibly with a reason saying why they didn't make it.

One way to do this is to initialize the unsuccessfulsales table with all sales and then to delete the successful ones.

```
create unsuccessfulsales
as select * from sales

delete from unsuccessfulsales
where id in (select id from successfulsales)
```

The trouble is that neither unsuccessfulsales nor successfulsales is indexed, so the result will take time proportional to the square of the number of sales (assuming most sales are successful). The time is thus superlinear with respect to the input time. So, one solution is to index successful sales. That will help enormously.

An alternative solution is to use an outer join and to detect rows having nulls.

```
insert successfulsales
select sales.id, sales.itemid, sales.customerid, sales.storeid,
sales.amount, sales.quantity,
item.itemid as itemtest,
customer.customerid as customertest,
store.storeid as storetest
from
```

```
((sales left outer join item on sales.itemid = item.itemid)
(left outer join customer on sales.customerid =
 customer.customerid)
left outer join store on sales.storeid = store.storeid);

create unsuccessfulsales as
select *
from successfulsales
where itemtest is null
or customertest is null
or storetest is null

delete
from successfulsales
where itemtest is null
or customertest is null
or storetest is null
```

6.2 Perform Data Integrity Checks at Input Time

Consider the TPC/B schema

- account(id, balance, branchid,…)
- branch(branchid,…)

It is better to make sure the branchid in the account row is correct when putting in the account row rather than each time an account balance is updated.

This may seem obvious, but we so often see extra checks for integrity being thrown in that we wonder whether people have been reading too many surgeon general warnings.

6.3 Distribution and Heterogeneity

In finance as in other international businesses, distributed databases reflect two organizational cliches: workflow from one department to another (e.g., sales to back office) and physical distribution of cooperating departments (e.g., global trading).

6.3.1 Interoperability with Other Databases

As in many industries, financial data must travel from clients to midoffice servers to a variety of back-office servers and risk management systems. Acknowledgments and corrections travel back.

Case You have a front-office system that records trade for profit and loss and risk management purposes. You want to send trades to the back office for final clearing. You don't want to lose any trades because of the financial loss implied. How should you solve this problem?

A replication server is one approach, but it has a few problems:

- Commits at the source aren't guaranteed to flow to the destination.
- Sometimes the destination must respond, and this is awkward with replication servers.
- It is hard to determine where a piece of data is in the chain from source to destination.

The standard academic recommendation is to use two-phase commit, but this is widely mistrusted on Wall Street because of horror stories about blocking (the situation in which the failure of one machine causes another one to stop). In addition, there is the serious problem that many database applications fail to provide an interface to the first phase of two-phase commit.
A good strategy is to implement a *buffer table* on the source system side that holds information in the denormalized form required by the destination system (Figure 6.2).

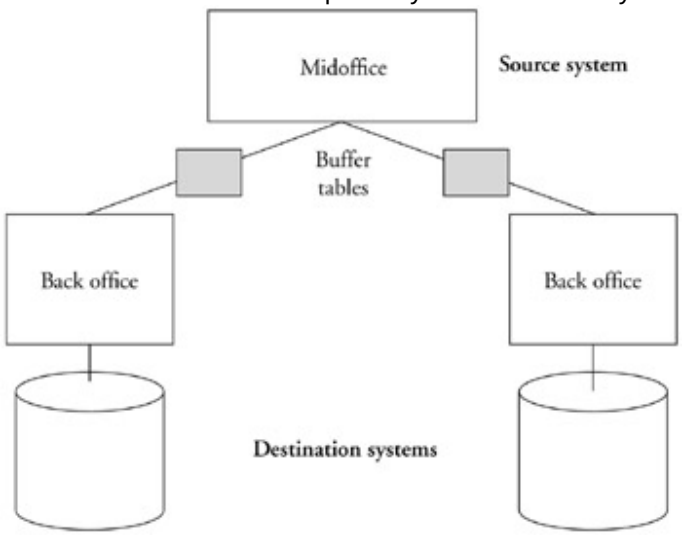


Figure 6.2: Buffer table. Source system transaction write to buffer tables and back-office systems read from them. If back-office systens must respond, then either build a clustering index on the buffer tables or use a second response table written by the back-office system.
A process from the destination database reads a new tuple *t*, from the buffer table at the source site. Then, in a second transaction, that process puts the tuple into a buffer table on the destination site. A second process, as one transaction, processess the tuple on the destination site and marks the tuple as "deletable" in the destination site buffer. A third process issues a transaction to delete the tuple from the source buffer, then it performs a second transaction to remove the deletable tuple from the buffer table on the destination site. (The order is important to avoid processing tuples twice at the destination. Recovery can always repeat the third process for any tuples marked "deletable" at the destination.) Such a scheme frequently runs into lock contention because both the source and the destination are scanning the buffer table for update purposes. What would you do about this problem?

Here are two approaches.

- The destination process can mark a table other than the buffer table to avoid update-update conflicts on the buffer table.
- If the exact order of updates at the destination site is not important, then the primary site can use a clustering index based on a hash of a key. The destination process can then check one bucket or another of the hash on each of its polling steps to the buffer table. This reduces the likelihood of conflict.

Case Another typical scenario is that a single transaction in a master database system performs complete transactions on several other foreign databases. If the transaction on the

master aborts, some of the transactions on the foreign databases may have committed. When the master transaction restarts, it should not redo already committed foreign database transactions.

Avoiding this requires a judicious use of a "breadcrumb table" at each foreign database. That is, design your system so that each update on the foreign database has an identifier that can be determined from its arguments. This identifier must be unique for all time. For example, you can generate a unique key given the date and time of a trade. Write this unique identifier into the breadcrumb table in the same transaction as the actual update. That is how to keep track of how much of the global transaction has completed. This is a special case of "workflow." In our experience, this is such a common case that it constitutes a design pattern.

6.3.2 Global Systems

Most large and many small businesses today are global. Information businesses have global inventories that are traded around the clock on different markets. (Markets have, in fact, converted most atom businesses into bit businesses, to use Nicholas Negroponte's terminology. Oil, for example, is traded several times while a tanker makes a single voyage.) In finance, stocks and bonds are traded nearly 24 hours per day (there is a small window between the time New York closes and Tokyo opens).

Global trading of this sort suggests two solutions: have a centralized database that traders can access from anywhere in the world via a high-speed interconnect. This works well across the Atlantic, but is expensive across the Pacific. It also makes people feel psychologically uncomfortable since they are dependent on a far-away data source.

Replication is delicate for concurrency control reasons, however. If updates on all data can happen anywhere, then serialization errors can occur.
Example At 11:59 A.M. account *X* has a cash position of $10 million at sites A and B (say, in New York and Tokyo). Suppose that site A increases the cash position of account *X* by $2 million and site B increases it by $1 million at 12 noon. A replication server translates the update at site A to a delete and insert sent to site B and similarly for the update at site B one second later. Two seconds later, site A writes the value that B had written at noon (yielding $11 million) and B writes the value that A had written at noon (yielding $12 million). The values are inconsistent with one another and with the truth ($13 million).[1] This is a classic "lost update" problem and is avoided in centralized systems that make use of serialized transactions.

Gray et al. have examined the lost update and related problems in a replication server setting from a probabilistic point of view. The error probability increases superlinearly as the number of sites increase. In the idealized model of that paper, a factor of 10 increase in sites leads to more than a 1000-fold increase in the probability of error.

Sometimes, you can exploit knowledge of the application to avoid a problem. If you know, for example, that all updates are additions and subtractions, you can send the update operation itself to all sites, and they will all eventually converge to $13 million. Such a replication strategy must, unfortunately, be explicitly programmed by the user if the database management system sends delete-insert pairs to reflect each update. The problem is, in general, difficult and argues for centralized databases whenever possible.

Sometimes application semantics helps, however.

Case A trading group has traders in eight locations accessing six servers. Access is 90% local. Trading data (trades made and positions in securities) are stored locally. Exchange rate data, however, is stored centrally in London. Rate data is read frequently but updated seldom (about 100 updates per day). Wide-area network performance is a big issue and slows things down a lot; 64-kilobit links between the sites (even not all possible pairs) costs $50,000 per month.

Here are the two problems.

- For traders outside London, getting exchange rates is slow. It would be nice if exchange rates could be replicated. But consistency is required in at least the following senses. (1) If a trader in New York changes a rate and then runs a calculation, the calculation should reflect the new rate. This consistency requirement precludes a replication strategy that updates a primary in London and then does an asynchronous replication to all other sites because the replication has an unpredictable delay. (2) After a short time, all sites must agree on a new exchange rate. Before reading on, what would you do?
  Here is one possibility. Synchronize the clocks at the different sites. Put a database of exchange rates at each site. Each such database will be read/write. Attach a timestamp to each update of an exchange rate. Each exchange rate in the databases will be associated with the timestamp of its latest update. An update will be accepted at a database if and only if the timestamp of the update is greater than the timestamp of the exchange rate in that database. This scheme will guarantee consistency requirement (1) and will cause all sites to settle eventually to the most recent update, satisfying (2).
- Trade data is mostly local, but periodically traders collect baskets of securities from multiple sites. The quantity available of each security must be known with precision. Right now, retrieving this information is slow largely because of WAN delays. The current implementation consists of an index that maps each security to its home database and retrieves necessary data from the home site.
  A possible improvement is to maintain a full copy of all data at all sites. Not all this data will be up to date ("valid") at all times, however. When a market closes, all its trades for the day will be sent to all other sites. When receiving these updates, a site will apply them to its local database and declare the securities concerned to be "valid." When a site *s*, opens for trading, all other sites will declare securities belonging to *s*, to be "invalid."

6.3.3 Managing Connections Socialistically in a Distributed Setting

As CICS designers knew in the 1960s, batching connections to databases is a good idea. If we have an application server that communicates to many clients, then that application server should be a funnel to the database. Each client state is represented by a data structure. A program managing a single server connection can visit each client data structure, access the server on behalf of that client, and then go on to the next one.
Whereas this conserves connections, it is sometimes slow because too many calls to the database are made. For this reason, it is sometimes helpful to collect the requests from many clients and send them all in one query to the database system (Figure 6.3). This is obvious if every client requires the same information, for example, they all need to know the table of exchange rates. This holds even if that isn't the case. For example, if there are 10 requests for account balances, it is better to ask for all 10 in a single request having an in clause than to issue 10 single selects.

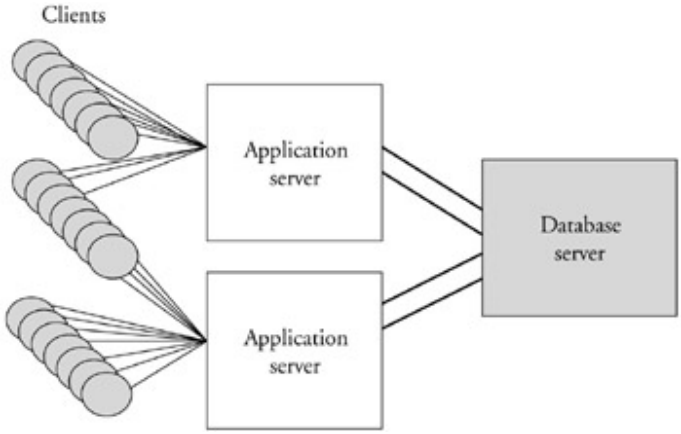


Figure 6.3: Managing connections socialistically. Instead of opening a database connection for each client, the application server serves as a funnel to the database.

[1]If your replication server is smart enough to send updates, then imagine that B had performed an update by 10% instead of an increase of $1 million. The result would still be inconsistent.

6.4 Trading Space for Time in History-Dependent Queries

Money is an important motivator on Wall Street. Applications that determine bonuses enjoy a lot of attention.

Case An application calculates the position of each trader per security (either a stock or a bond) broken down by lots, where each lot corresponds roughly to a date of purchase. Traders are penalized for having old lots. Selling securities from a position to an outsider (including someone at another trader's desk) reduces the position in a FIFO manner (reduces from the oldest lot first). Selling securities to someone at the same desk reduces the position from the newest lot first.

The process of calculating the position of a trader at the end of a day runs into the following complication: a cancel of a trade from several days ago may change the lots held by various people. For example, suppose

1. day 1—Glen buys 10 of security S from outsider Judy
2. day 2—Glen buys 5 of security S from outsider Vivian
3. day 3—Glen sells 6 of security S to Dennis who is on the same desk
4. day 4—Dennis sells 3 of security S to Ying who is on the same desk

Then Dennis holds 2 shares of day 2 securities and 1 share of day 1 securities. Ying holds 3 shares of day 2 securities. If the day 2 sale is canceled, however, Dennis owns only day 1 securities and similarly for Ying. One way to deal with this is to replay all trades on a given security from the beginning of time after removing canceled trades. The application then requires all night to run.
Algorithmic optimization: Trade space for time. Have the application maintain the position of each trader in each security for each lot for all (or most) days. In the case that a given security has cancels against it, the oldest affected trade will be identified. Suppose that is on day *d*.

History will be rolled forward from the position of day $d-1$ (or the most recent position available preceding *d*). This reduced the time from eight hours to one hour.

A critical component of this roll-forward operation is finding all the relevant trade tuples. It turned out that one of the tables needed to calculate these tuples had no indexes on it. Adding an index reduced the time from 1 hour to 20 minutes.

6.5 Chopping to Facilitate Global Trades

Case When the trading day is over, many operations must be done to move trades to the back office, to clear out positions that have fallen to zero. This "rollover" procedure is a single-threaded application. It causes no problems provided no trades are hitting the database at the same time. In a global trading situation, however, rollover in New York may interfere with trading in Tokyo.
It is therefore of great interest to "chop" the rollover transaction into smaller ones. See Appendix B for a discussion of the theory of chopping. The conditions for chopping are that the ongoing trades should not create cycles with the rollover pieces. In point of fact, these trades don't conflict at all with the rollover pieces. They appear to because they touch some of the same records. Further, the rollover pieces are *idempotent* so if one aborts because of deadlock, it can be rerun without any harm.

Typical such operations (1) send trades that haven't yet gone to the back office, (2) clear out positions having balance 0, and (3) compute statistics on each trader's profit and loss.

Separating these transactions into separate ones decreases the time during which there is a lock conflict.

6.6 Clustering Index Woes

Many optimizers will use a clustering index for a selection rather than a nonclustering index for a join. All else being equal, this is a good idea. The trouble is that if a system doesn't have bit vectors, it may use only one index per table.

Case Bond is clustered on interestRate and has a nonclustered index on dealid. Deal has a clustered index on dealid and a nonclustered index on date.

```
select bond.id
from bond, deal
where bond.interestRate = 5.6
and bond.dealid = deal.dealid
and deal.date = '7/7/1997'
```

The query optimizer might pick the clustered index on interestRate. Unfortunately, interestRate is not normally very selective because most bonds have about the same interest rate. This prevents the optimizer from using the index on bond.dealid. That in turn forces the optimizer to use the clustered index on deal.dealid. This leaves many tuples to search through. By causing deal to use the nonclustering index on date (it might be more useful to cluster on date, in fact) and the nonclustering index on bond.dealid, the query used around 1/40 of the original reads.

6.7 Beware the Optimization

Case Position and trade were growing without bound. Management made the decision to split each table by time (recent for the current year and historical for older stuff). Most queries concern the current year, so should run faster.
What happened: A query involving an equality selection on date goes from 1 second with the old data setup to 35 seconds in the new one. Examining the query plan showed that it was no longer using the nonclustered index on date. Why?

The optimizer uses a histogram to determine whether to use a nonclustering index or not. The histogram holds around 500 cells, each of which stores a range of, in this example, date values. The ranges are designed so that each cell is associated with the same number of rows (those in the cell's date range).

The optimizer's rule is that a nonclustering index may be used only if the value searched fits entirely in one cell. When the tables became small, an equality query on a single date spread across multiple cells. The query optimizer decided to scan.

The fix is to force the use of the index (using hints) against the advice of the optimizer.

6.8 Disaster Planning and Performance

If you take a database course in school, you will learn a model of recovery that features a convenient fiction: stable storage. The idea is that failures affect only volatile storage, whereas stable storage survives failures unscathed. In some lecture halls, disks soon acquire the character of stable storage. Discussions that admit the possibility of disk failure posit the use of a disk mirror or a RAID disk to do away with this problem. Do you believe this fiction?

Case A server is used for trading bond futures having to do with home mortgages. The application needs to be up only a few days per month, but the load is heavy during those few days. During a weekend batch run, 11 out of 12 disks from a single batch from a single vendor failed. Nothing terrible happened, but everyone was quite spooked.

The system architect didn't want a backup machine because his understanding was that the backup could catch up to the primary only after 20 minutes. His traders couldn't stop trading for that long, so would have reverted to calculations by hand, at which point they would be unwilling to reenter trades to the system.

In this organization, policy requires that backups be remote because an entire building may suffer from a fire or bomb scare. (This happens in real life. Credit Lyonnais's Paris office burned up a few years ago. Smoke has shut down the New York Stock Exchange's principal computers. Even the destruction of the World Trade Center caused no loss of data. In those cases, remote backups allowed continued operation.) So, as a first case study, how would you design a remote backup? Here are some of the high-availability choices typically made on Wall Street.

- *Shared disk high-availability servers*—A pair of shared memory multiprocessors are attached to RAID disks. If the primary multiprocessor fails, the backup does a warm start from the disks. If a disk fails, the RAID parity prevents any error from being visible (and the faulty disk is replaced by a hot spare). This configuration survives any single disk plus any single processor failure, but does not survive site disasters or correlated failures. So, it is not a remote backup solution (Figure 6.4).

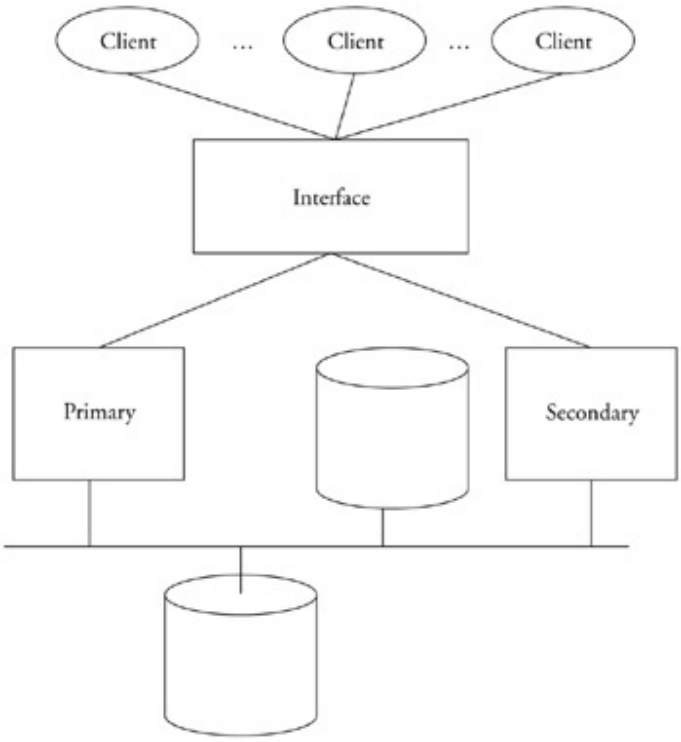


Figure 6.4: High-availability disk subsystem. Writes go to the primary and into the high-availability disk subsystem. This subsystem is normally a RAID device, so it can survive one or more disk failures. If the primary fails, the secondary works off the same disk image (warm start recovery). This architecture is vulnerable if the high-availability disk subsystem fails entirely.

- *Dump and load*—A full dump of the database state is made to a backup site nightly and then incremental dumps (consisting of the changed page images) are made at regular intervals (e.g., every 10 minutes) during the day. If the primary site fails, the backup can perform a warm start from the last incremental dump, but committed transactions since the last incremental dump will be lost. Dump and load imposes little overhead on the primary server, but the loss of committed transactions can be fatal for some applications. It also assumes the traders will be patient during the several-minute delay before the system comes up, not always a likely assumption.
- *Replication server*—As in dump and load, a full dump of the database state is made to the backup site nightly, and then all operations done to the primary are sent to the secondary (typically, as SQL operations) after they are committed on the primary (Figure 6.5). With a replication server, the backup is within a few seconds of being up to date. So, a replication server loses fewer transactions than dump and load. Further, a replication server allows decision support queries to the backup machine. Finally, the backup machine can become the primary in a matter of seconds. A replication server can be a huge administrative headache, however, requiring an administrator to keep table schemas in synchrony, to ensure that triggers are disabled at the backup site, and to be ready when the network connection breaks though the vendors have become much better at helping the administrators with the bookkeeping.

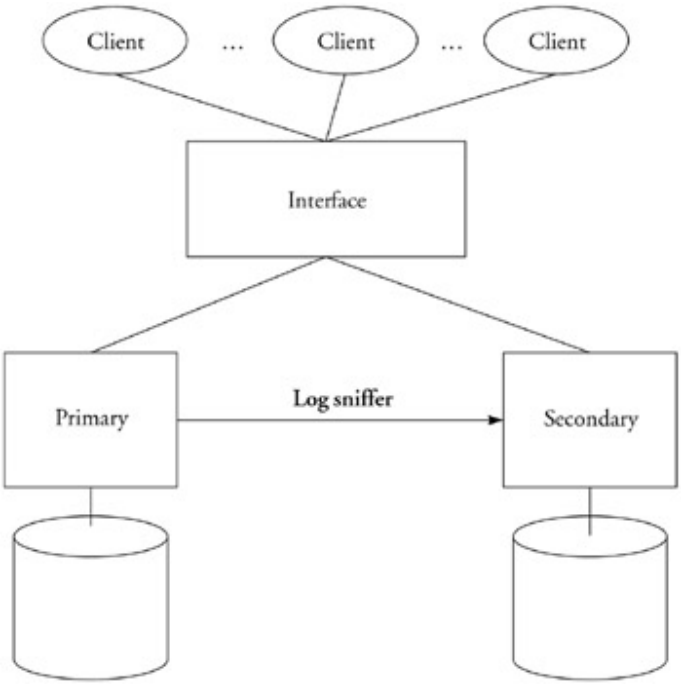


Figure 6.5: Replication server. The backup reads operations after they are completed on the primary. Upon failure, the secondary becomes the primary by changing the interface file configuration variables. This architecture is vulnerable if there is a failure of the primary after commit at the primary but before the data reaches the secondary.

- *Remote mirroring*—Writes to local disks are mirrored to disks on a remote site. The commit at the local machine is delayed until the remote disks respond. This ensures that the remote site is up to date, but may stop the primary in case the remote disks do not respond or the network link fails. To overcome this availability problem, special subsystems will buffer the transmission from the local to the remote, in which case, the result is similar to a replication server but easier to administer since they take place at the hardware rather than the database administration level.
- *Two-phase commit*—Commits are coordinated between the primary and backup. If one fails, blocking can occur (Figure 6.6). This fact scares off many people on Wall Street.

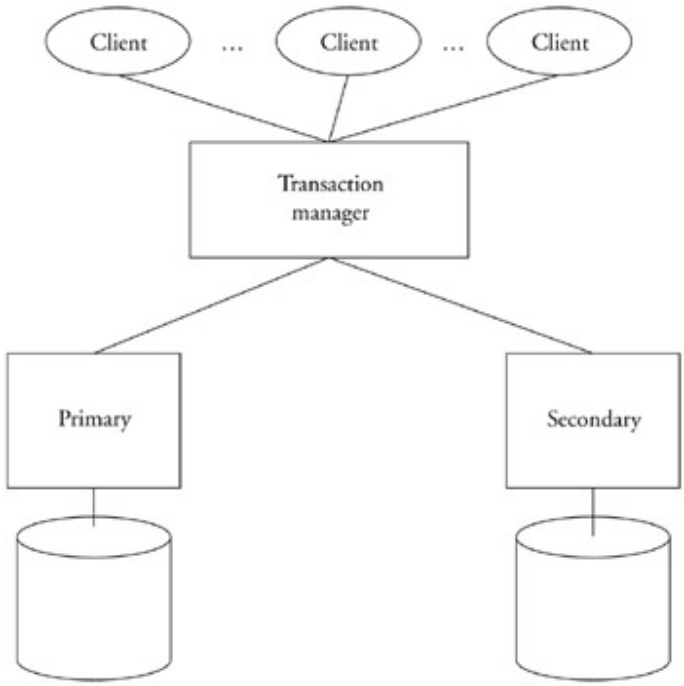


Figure 6.6: Two-phase commit. The transaction manager ensures that updates on the primary and secondary are commit consistent. This ensures that the two sides are in synchrony. This architecture might block the primary in case of delays at the secondary or failure of the transaction manager.

- *Wide-area quorum approaches*—Compaq, Hewlett-Packard, IBM, and others offer some subset of the following architecture. Primary and secondary servers are interconnected via a highly redundant wide-area cable. Clients can be connected to any server since the servers execute concurrently with their interaction managed by a distributed lock manager. Disks are connected with the servers at several points and to one another by a second wide-area link. Heartbeats monitor the connectivity among the various disks and processors. If a break is detected, one partition holding a majority of votes continues to execute. Other partitions don't. In this architecture, backups are completely up to date. Any single failure of a site, processor, or disk is invisible to the end users (except for a loss in performance). The same holds for many multiple-failure scenarios. Most money transfer applications and most major exchanges use this architecture (Figure 6.7).

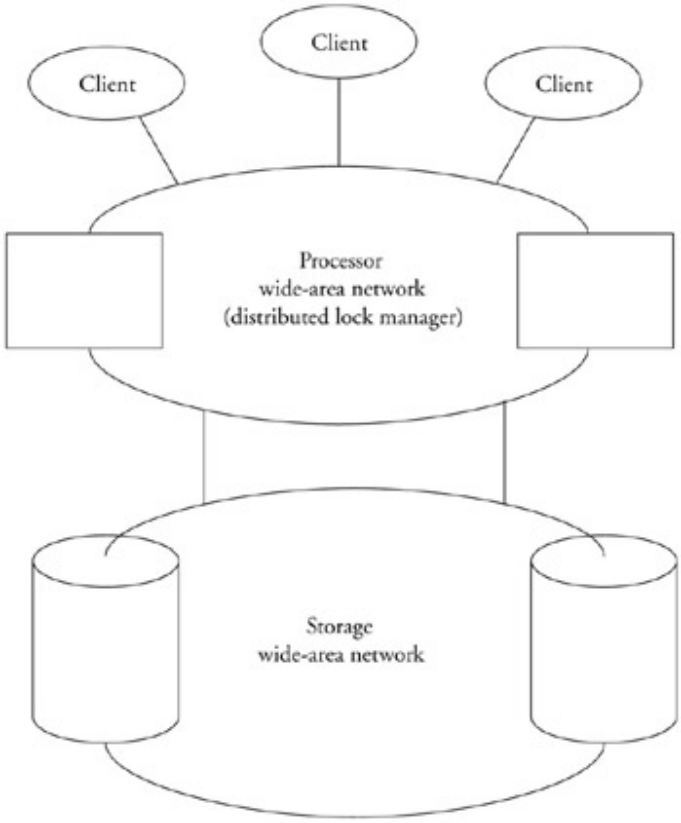


Figure 6.7: Quorum approach. The quorum approach is used in most stock and currency exchanges. It survives processor, disk network, and site failures.

People who don't use this architecture cite the following reasons:

- The shared lock manager is a potential bottleneck.
- Partitioning may occur improperly.
- It locks them into a proprietary and expensive architecture.
- Specialized software priests must minister to its needs.

Whatever backup system you decide on, (1) don't buy disks in batches, especially for the primary and backup (such a practice undermines any hope of failure independence); and (2) never use a scheme in which a problem at the backup can slow down the primary.

6.9 Keeping Nearly Fixed Data Up to Date

Lookup information such as symbol-name mappings must be at a trader's fingertips. Relational systems or the connections to users' personal computers are often too slow, so the information is held outside the database. The problem is that this information is updated from time to time (e.g., when new securities or new customers are added).

So, a critical question is what to do with the updates. The following have all been tried:

- Ignore updates until the next day.
- Program clients to request refresh at certain intervals.
- Have the server hold the state of the clients. Send messages to each client when an update might invalidate an out-of-date copy or simply refresh the screens. One efficient implementation uses triggers.

Which would you choose? The stateful server approach gives the most up-to-date approach, but requires the most programming effort.

6.10 Deletions and Foreign Keys

Case Data is extracted from a production system and transferred to a data ware-house, then it is deleted. Indexes are dropped on the tables where a large number of deletions are anticipated to avoid the overhead of deleting index entries. Let us take as an example the TPC-H schema, where lineitem.l_suppkey has supplier.suppkey as a foreign key.
What happened: The deletion of 300,000 records from the lineitem relation is faster than the deletion of 500 tuples from the supplier relation. Why?
There is overhead to check foreign key constraints. When tuples are deleted from a relation *R*, the system accesses the relations that define foreign keys on *R* in order to verify the constraint.

In case the attribute that is referenced as a foreign key is not a prefix of the key of an index on the referencing relation, the verification of a foreign key constraint implies scanning the referencing relation. In our example, there is no foreign key constaint referencing the lineitem relation. By contrast, the attribute suppkey in the supplier relation is referenced as a foreign key of the l_suppkey attribute in the lineitem relation. This attribute is a member of a composite clustered index on the lineitem relation, but this index is not used by the optimizer (l_suppkey is not the prefix of the composite index). Consequently, the lineitem relation is scanned for each tuple deleted from the supplier relation.

Adding a nonclustered index on the l_suppkey attribute in the lineitem relation (after the deletion from lineitem is performed) considerably speeds up the deletion from the supplier relation.

6.11 Partitioning Woes: The Hazards of Meaningful Keys

For performance and administrative reasons, a large bank partitions its clients by branch. It then commits the cardinal mistake of including the branch identifier inside the client key. The client key is used on all client accounts, the client debit card, and so on. Now, the client moves to a new branch. All accounts have to be closed and reopened, debit cards have to be destroyed and reissued. What a waste!

If the client had been given a meaningless key (say, a sequence number unique across the whole bank), then changing branches would require at most an update to a few nonkey fields in the client reference information. It is true that accesses to the client account at the client branch might have to go through a level of indirection. But that is a simple operation to make fast.

Please keep your keys meaningless.

6.12 The Problem of Time

Financial applications treat time in many ways, but the following four queries illustrate the main variants:

- Compute the value of my portfolio of bonds based on future payments and projected interest rates.
- Find correlations or other statistics among different stock prices over time or between option premiums and stock prices in order to help make better trading decisions.
- Find all events that are likely to precede a rise in the price of a security, regardless of precise timing constraints.
- Determine the data available at time *X* to trader *T* when he made the decision to buy *Y* for his own account.

Let us discuss each variant and its generalizations in turn.

6.12.1 Present Value

Contrary to what you read in the popular press, bonds are far more important than stocks for most investment banks. Companies, municipalities, and nations issue debt, and the markets buy them. Each debt note has an associated payment stream occurring at regular intervals, for example, each half-year. These debt notes are then swapped for cash (I'll borrow money and give some bonds I'm holding as collateral), tranched (sliced into payment streams of different risks), and otherwise manipulated. The basic computational issue is to value these payment streams.
But how should you value ten yearly $1 million payments? The basic principle is to normalize all payments to their *present value*. The present value of a payment of x dollars in i years assuming an interest rate of r is $y = (x/((1 + r)^i))$ dollars. (This definition is justified by the fact that putting y dollars in the bank at interest r would yield the same amount of money at the maturity of the bond as taking each of the payments and putting them in the bank at interest r.) So, the basic problem is to compute expressions of the form $(1 + r)^i$ for many different values of i. Since the value of r is never known precisely, many different r's are tried in order to calculate the statistics of the value of a bond.
Notice that each such calculation involves computing the log of $1 + r$, then multiplying by i, and then taking the inverse log of the result. Since many expressions have the same value of r but differ on i, we'd like to avoid doing the log more than once for a given r.

Relational systems on Wall Street typically store each payment in the form (amount, date, interest). The present value of each row is then computed independently of all others through either a trigger or an object-relational extension. This strategy misses the significant log optimization described in the last paragraph, and the resulting performance is punishingly slow. For this reason, many applications take the data out of the relational database and perform this calculation in some other way.

We have illustrated the potential benefits of user-defined functions executed on the database server in the previous chapter. Avoiding this exodus from the database requires the object-relational system to share computational work even among external functions. That is an undecidable task in general since the external functions are written in a Turing-complete language. Another possibility is to call external functions on bigger entities (e.g., arrays of payments and dates) so the function itself can do the necessary computational sharing.

6.12.2 Regular Time Series and Statistics

Trends are a popular abstraction for many average stock investors. Knowing that "the market is going up" is a powerful motivation to buy. Unfortunately, history and countless investment books suggest that this does not yield good results for two reasons First, it is not clear when a trend has started. Second, trend trading ignores the different properties of different companies. A far better strategy is to make use of correlations between different companies having similar properties. Take two similar banks B_1 and B_2. In the absence of differentiating external news (e.g., a default of a creditee of B_1 alone), the prices of the two banks should track each other closely. So, if B_1 starts going up while B_2 does not, a "pairs trader" will sell B_1 and buy B_2.

In the FinTime benchmark,[2] Kaippallimali J. Jacob of Morgan Stanley and Shasha have tried to capture the typical queries correlation and regression analysis require.

The model consists of a few relational tables that typically contain infrequently changing information about stocks and a number of time series tables about prices and stock splits.

Here are three typical queries.

1. Get the closing price of a set of ten stocks for a ten-year period and group into weekly, monthly, and yearly aggregates. For each aggregate period determine the low, high, and average closing price value.
2. Find the 21-day and 5-day moving average price for a specified list of 1000 stocks during a 6-month period. (Use split-adjusted prices.)
3. Find the pairwise coefficients of correlation in a set of ten securities for a two-year period. Sort the securities by the coefficient of correlation, indicating the pair of securities corresponding to that row.

Here are some features that relational systems require to answer such queries.

1. The special treatment of the semantics of time (e.g., the discovery of weekly boundaries and the ability to aggregate over such boundaries)
2. The ability to treat events in a time-ordered sequence (e.g., to perform moving averages
3. The ability to perform time-based statistical operations on multiple sequences

6.12.3 Irregular Time Series and Frequency Counting

Hurricanes, oil spills, earthquakes, and other natural and man-made disasters can influence the stock market in many ways. These events are not regular (we hope), and so moving averages and complex signal processing techniques are inapplicable. On the other hand, the fusion of disaster events with price data may make the following queries meaningful and useful:

- (Specific) Which disaster event type affects the price of insurance company X?
- (General) Which event sequences affect which companies within five days?

We have used the example of disasters, but similar queries apply to large purchases prior to merger announcements and other suspected insider activity.[3] One way to conceptualize such problems is to discover patterns in timed event sequences. Relational systems would model this by attaching a time attribute to events, but the discovery of sequence of events would require one or more joins. Most real-life systems would take some data from the database and do the real work outside.

6.12.4 Bitemporality

It is often necessary to know what someone knew at a certain date. For example, did trader Jeff believe that a merger was to take place between companies X and Y as of time Z when he bought stock in X? This might be grounds for legal concern. What was the best public guess as to the third-quarter profits of company X at time Z? This might explain why trader Carol took such a large position in the company at that time.

Such queries are called bitemporal because they involve two times: the time of the event in question (the merger or the third-quarter profits in our examples) and the time of the query (time Z in both cases). The point is that our knowledge of the event in question may be better now, but what we are interested in is what was known at some time in the past based on the transactions that had committed up to that point.

Whereas most queries having to do with compliance or personnel review occur rarely enough to be irrelevant for performance tuning, the same does not hold for other queries involving time. For example, consider a portfolio accounting application with queries like the following: As of last June, what was the anticipated revenue stream of this portfolio over the next ten years? Since a portfolio consists of many securities, this involves combining the streams from many different securities after determining which securities were in the database last June.
As Snodgrass and Jensen show in their book, such queries can be implemented in relational databases, but may require pages of code and may, in fact, be very slow. Indeed, the bank

officer who described the portfolio accounting application put the problem this way: "The functionality excites the users but makes the [standard relational] server glow a dull cherry red…."

At least two solutions are possible to speed up such actions.

- Implement the bitemporal model inside the database system. The risk is that this may create overly specialized implementations.
- Treat the data as ordered and see where that leads.

6.12.5 What to Do with Time

Our recommendations follow from a simple observation: *Sequences are more powerful than sets.*

Membership, intersection, union, and difference can be easily applied to sequences. On the other hand, order, *n*th best, moving averages, and other statistics require a tortured and inefficient encoding on sets, but are linear time on sequences. So, the best way to extend relational systems to support time is to allow query languages to take advantage of order—a scary thought for those of us brought up with the relational model.

Now let us consider two ways in which sequences might extend the relational model:

- Tables might be maintained in a specific order, as *arrables* (array tables). Selections, joins, and projections can all carry their multiset theoretic meaning to arrables (SQL is a multiset language when all is said and done), but new operations can be defined that exploit order. These new operations may include *n*th best operations, moving averages on certain columns, correlations between columns, and so on. Ordering tables into arrables has certain disadvantages, however. For example, an arrable should probably not be clustered using a hash index. Ordering does not, however, preclude a secondary hash index. The extra overhead for ordering is therefore one more access in most cases. This extra access may be significant in disk-resident data.
- Another approach to order is to break a relational commandment. In a normal SQL group by statement, for example,
- select id, avg(price)
- from trade
- group by id

each group corresponds to one stock id, and there are one or more scalar aggregates associated with that group (here, the average price in the group). Suppose instead that the following were possible:

```
select id, price, date
from trade
group by id
assume trade order by date
```

The net result here would be to cause a *vector* of prices and dates to be associated with each stock. Now that we have vectors, we can apply vector style operations that may be offered by a system or that an application program may write. For example, we may sort the prices in each group by date and then construct moving averages. Regrettably, allowing vector elements in rows violates first normal form. Maybe this is all right.

These two ways of incorporating order are separable. If introduced together, the stock arrable might be ordered by date and then the query becomes

```
select id, price
from trade
```

```
group by id
assume trade order by date
```

Prices will then be ordered by date within each date, provided the grouping preserves the sort order of the trade arrable. Even better, we can construct the five-day moving average in one shot, given a function that computes the moving average of a sequence, say, "movavg" and takes a time parameter

```
select id, movavg[5,price]
from trade
group by id
assume trade order by date
```

Several systems treat sequences as first-class objects, including Fame, SAS, S-Plus, and KDB. We discuss these in some detail in Appendix C.

Looking back on the time series queries we face in finance, let us see what order does for us.

- Computing the present value and similar operations are natural on sequences. If the external functions to compute these functions are vector aware, then optimizations, such as performing the logarithm once instead of many times, can easily be incorporated.
- Correlations, moving averages, grouping by time periods such as weeks and months, and special-purpose interpolations require an awareness of date as a special data type and the ability to add arbitrary functions in an object-relational style. Event data mining requires a subset of this functionality.
- Bitemporality remains a challenge, but less of one. Because bitemporal queries involve unchanging historical data, you can afford to store redundant data (e.g., the membership history of each portfolio). So, the query that makes a current server glow a cherry red can perhaps become red hot.

Bibliography

FAME (the temporal database system). http://www.fame.com.

FinTime (a financial time series benchmark). http://cs.nyu.edu/cs/faculty/shasha/fintime.html. FinTime is a benchmark having typical Wall Street style.

Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. *The dangers of replication and a solution.* In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Quebec, Canada, June 4–6, 173–182. ACM Press, 1996. This discusses the semantic pitfalls of replication. Work to overcome those pitfalls is continuing, and you can see pointers to that work in the citeseer database (http://citesser.nj.nec.com/).

Christian S. Jensen and Richard T. Snodgrass. *Semantics of time-varying information.* In *Information Systems*, 21(4):311–352, 1996.

KDB product. *The array database systems KDB and KSQL can be downloaded on a trial basis from* http://www.kx.com.

B. D. Ripley and W. N. Venables. *Modern Applied Statistics with S-Plus.* Springer-Verlag, 1999. http://www.stats.ox.ac.uk/ripley/ has a lot of useful functions.

John F. Roddick and Myra Spiliopoulou. *A bibliography of temporal, spatial and spatio-temporal data mining research. SIGKDD Explorations*, 1(1):34–38, 1999. Roddick and Spiliopoulou have maintained their excellent bibliography on temporal data mining as a service to the community.

SAS (the statistical package). http://www.sas.com.

Richard T. Snodgrass (editor). *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995.

Richard T. Snodgrass. *Developing Time-Oriented Database Application in SQL.* Morgan Kaufman, 2000. Jensen, Snodgrass, and many others have worked for several years on bitemporal logic. The two are excellent writers, and we recommend their book, their paper, and even the TSQL2 manual.

Exercises

EXERCISE 1

You are managing the human resources database of a large manufacturing corporation. The employee table has the fields (id, name, salary, department). Ninety percent of the employees are factory workers who have four salary levels. The remainder are office employees who have many different salaries.

The factory safety table has the fields (employeeid, whentrained). Consider the query

```
select whentrained
from employee, factorysafety
where employee.id = factorysafety.employeeid
and employee.salary = <input>
```

Suppose you now have an index on employee.id and employee.salary. Which index would you want the query to use?
Action. Most systems will use only one index and then generate a temporary table that will have no further indexes. If the query uses the index on employee.salary, it will likely not be very selective, and it will preclude the use of the index on employee.id.

EXERCISE 2

You have two independently developed databases A and B. You want to send information from A to B after it is processed at A. You want to be sure that each item sent is processed exactly once at B. What would be your design?
Action. A replication server would be the easiest choice, but you must carefully examine what happens upon a failure at site A. In particular, could it happen that a transaction may commit at A, but the transaction's updates are not transferred to B? A buffer table would work because it would keep transactional breadcrumbs associated with the various steps: data committed at A, data transferred to B, data processed at B. Provided the recovery subsystem of each database does its job, you can guarantee "exactly once" semantics: each change to A will be processed at B exactly once.

[2]FinTime is available at the following URL: http://cs.nyu.edu/cs/faculty/shasha/fintime.html.

[3]Of course, such event sequence queries appear elsewhere (e.g., in Web data mining). A typical problem there is to infer the probability that a person will buy an item given that he has visited some sequence of pages.

Chapter 7: Troubleshooting

—Alberto Lerner, *DBA, Doctoral Candidate*

7.1 Introduction

Monitoring your DBMS's performance indicators is one of the best ways to diagnose how well or badly the DBMS is performing. These indicators can be thought of as a DBMS's performance "vital signs" and can be calculated using counters, gauges, or details of the DBMs's internal activities. Monitoring performance indicators can ultimately help you determine the areas to which your tuning efforts should be directed.

To give you a better idea of which indicators and which internal activities we are talking about, consider the path a query follows inside a DBMS from its submission to the point when a result set or a return code is produced. Immediately after a query is entered into the system, it is transformed into an access plan by the optimizer. The access plan is one of the performance indicators that tell you how well the optimizer is contributing to the overall performance of your system. This is the first place to look whenever a query presents performance problems, but the query's story does not stop here.

An access plan causes other internal subsystems to run on its behalf, which will answer the query the plan represents. A query requires the cooperation of all components: the query execution subsystem may be computing a part of the plan (e.g., a join) while the disk subsystem may be fetching the next pages necessary for that operation; the cache manager may be trying to make room for those pages; the locking subsystem may be confirming that no one is changing the joined pages while the query is running; the recovery subsystem may be guaranteeing that if the application rolls back, the changes performed so far will be undone correctly; and so on. Each of these subsystems offers possibilities of performance monitoring.

Unfortunately, the number of these indicators can be overwhelming. We need a methodical approach to applying and interpreting them.

7.1.1 A Consumption Chain Approach

If you view DBMS subsystems as resources, the queries and commands submitted to the DBMS are consumers. Subsystems in turn consume low-level resources such as CPU, memory, and disks managed by the operating system. Understanding this query–DB subsystem–raw resource *consumption chain* is the first step toward the systematic approach that we suggest for performance monitoring. Such a hierarchy of producer-consumers in a database system is shown in Figure 7.1.

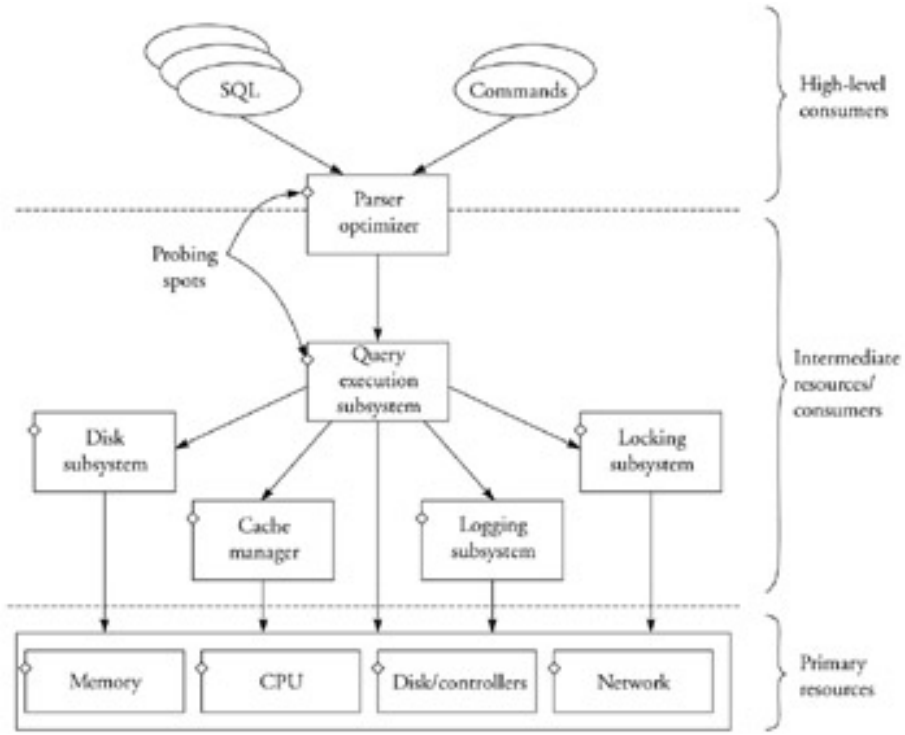


Figure 7.1: A producer-consumer hierarchy of DBMS resources. Performance probing points exist at all points of the hierarchy. Producers are also known as resources.

The producer-consumer hierarchy can be depicted as follows:

- *High-level consumers:* Every process or user attached to the SQL or the external command interface of the database system. Examples of consumers in this class are application and online users issuing queries in SQL, online or batch processes loading or extracting data, maintenance routines backing up data, or reorganizing disk partitions.
- *Intermediate consumers/resources:* Database subsystems that interact with one another to answer the queries and commands issued by the upper level. Examples of elements of this class are the locking, logging, cache management, and query execution subsystems. Each higher-level consumer concurrently uses several of the subsystems in this class, and its elements consume primary resources.
- *Primary resources:* Raw resources of the machine along with the operating system services that manage them. There are four primary resources: disks and controllers, CPUs, shared memory, and network. The levels above this one will all depend directly or indirectly on these resources.

Performance problems may involve several consumers and resources in a few recurrent cause-effect patterns. Here are the most common ones. A high-level consumer can monopolize an intermediary resource, indirectly slowing down other consumers that used that resource, as shown in Figure 7.2(a). For instance, a query that locks an enormous number of rows for its exclusive access prevents the locking subsystem from servicing other lock requests over those rows. Or a poorly configured subsystem can exhaust a raw resource and spread negative effects to high-level consumers, as depicted by Figure 7.2(b). For example, a disk subsystem that stores everything concerning an application (data, indexes, temporary area, logs, etc.) on a single disk will affect queries that belong to that application or that use that disk. Furthermore, an overloaded primary resource can slow down an entire system, as represented in Figure 7.2(c). That is the case when the CPU is overly busy because nondatabase processes are using it at the same time.

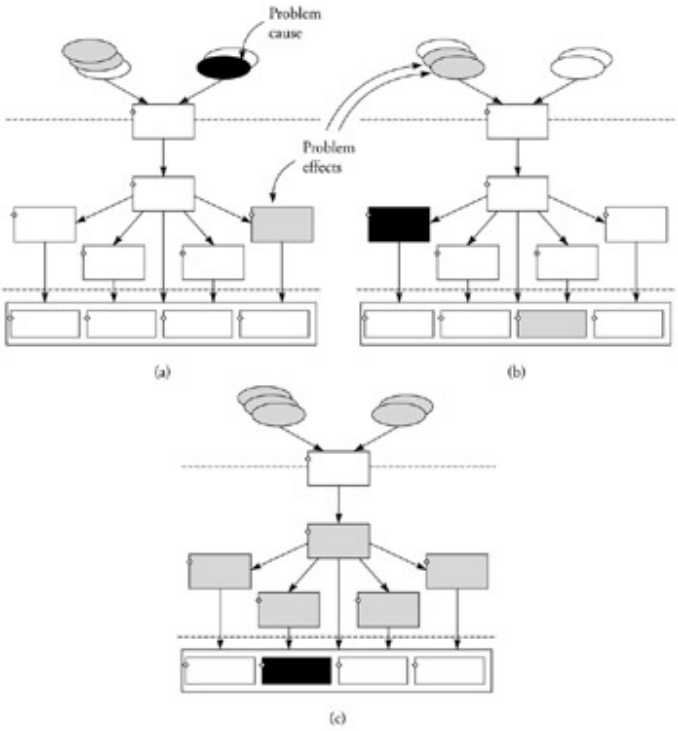


Figure 7.2: Cause-effect patterns in the consumption chain.

The chain concept helps you understand the relationship between a problem cause and its effects. This should help you identify causes.

7.1.2 The Three Questions

If you were able to picture the DBMS as this three-level machine and understood that all its internal parts affect each other, then take a look at these three questions.

- High-level consumer question (Q1): Are applications and, more specifically, their critical queries being served in the most effective manner?
- Intermediate consumer/resource question (Q2): Are DBMS subsystems making optimal use of resources so as to reach the desired performance levels?
- Primary resources question (Q3): Are there enough primary resources available for DBMS consumption and are they configured adequately, given the current and the expected operation workload (number of users, types and frequency of queries, etc.)?

A systematic procedure to answering the questions can be described as follows. First, investigate important queries—we will say how to find those—by answering Q1 over them. If you find any problem, solve it. The rationale behind starting with Q1 is that having found a positive answer for Q1 first, if any trouble appears in the other levels or questions you can discard high-level overconsumption as a cause and focus on investigating system resources. Figure 7.3 shows this *critical query monitoring* part of the procedure. Then, adopt a monitoring methodology that investigates the consumption chain from the bottom up. The idea here is that if the critical consumers are tuned, your DBMS's subsystems and your machine raw resources should be parameterized to handle their workload. Figure 7.4 depicts such a *routine monitoring* part of the procedure.

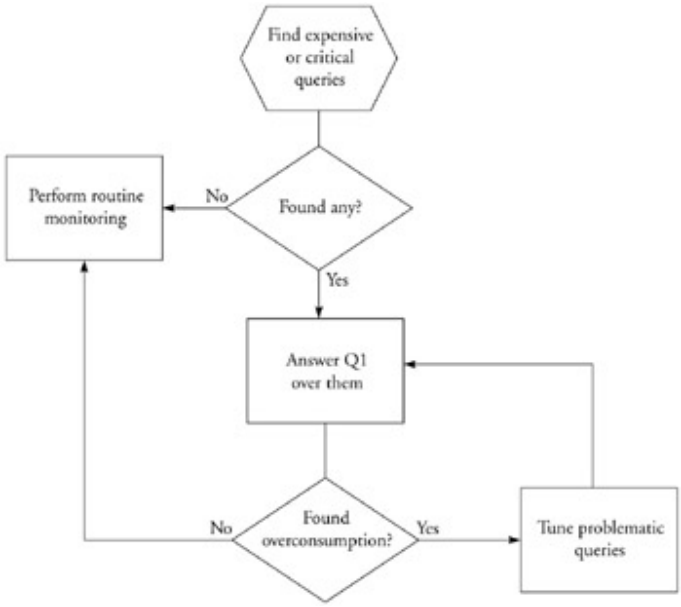


Figure 7.3: Critical query monitoring.

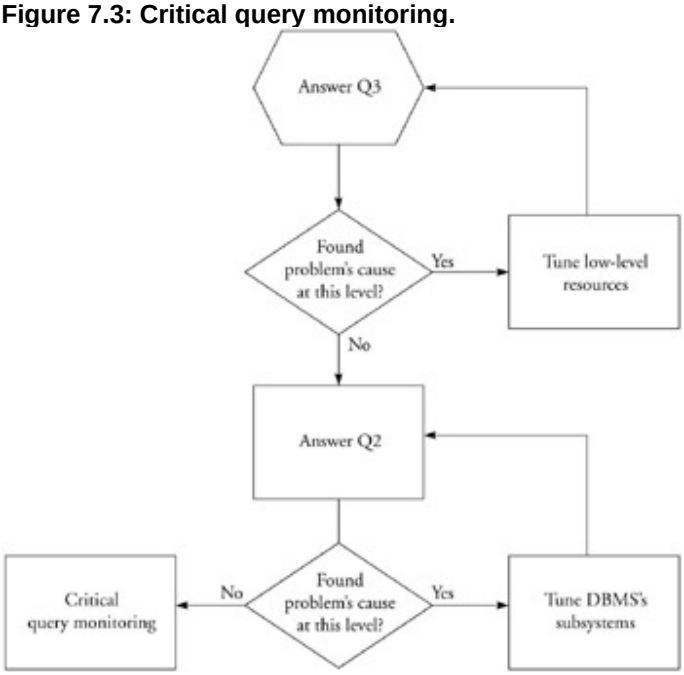


Figure 7.4: Routine monitoring.

7.2 How to Gather Information: The Tools

As indicators vary from simple gauges to complex data, so do the tools to extract and present them. Of particular interest to performance monitoring are *query plan explainers, performance monitors*, and *event monitors*.

7.2.1 Query Plan Explainers

Query plan explainers are the application developer's best friends. Whereas declarative languages like SQL free the developer from the burden of specifying how to retrieve data, SQL optimizers do not always find the best way to perform the retrieval. A query plan explainer can show you the exact access plan generated for a query, in case you need to verify its efficiency.

A query plan is usually represented as a tree whose nodes correspond to operations such as scans, and joins and arcs represent tuple (i.e., record) flow. Every operation has one or more data inputs and exactly one data output.[1] Data inputs can come from tables or from other operations.

Each DBMS has a vocabulary for labeling each operator, and you should get acquainted with it. The leaves of the tree are made of operators that read data from single tables or from indexes. You will see operators called table scan, index scan, index range scan, and so on. In the internal nodes of the tree, you will find operators identified as nested-loop join, sort-merge join, hash join, filter row, aggregation, duplicate elimination, and sort. The root of the tree is the operator that makes that last data consolidation before showing the results to the user. In some systems, it has a specific name like "result" or the name of the SQL command. Besides naming the operators, a good explainer will annotate statistical information such as expected cardinality and expected CPU consumption for each operator.
Figure 7.5 shows an example of a visual query plan. The query appears in the upper half of the screen while its access plan is displayed sideways in the bottom half. In the example, we can see that the algorithm used to solve the join expressed by the query is the Nested Loops Join. The outer table is connected to the join operator via the upper arrow; it is the table *employee* and it is being scanned, as the Table Scan operator suggests. The inner table is the *jobs* table and it is being accessed through its *PK_jobs* clustered index. Since there is nothing further to process than the join, its results are sent to the final operator, here called SELECT. Note that the explainer also estimates the percentage of the total time spent with each operator. Here, the time to join the tables is negligible (0%) compared to the time to scan the outer table (79%) and to seek the inner (21%). Other examples of explainers appear in Table 7.1.

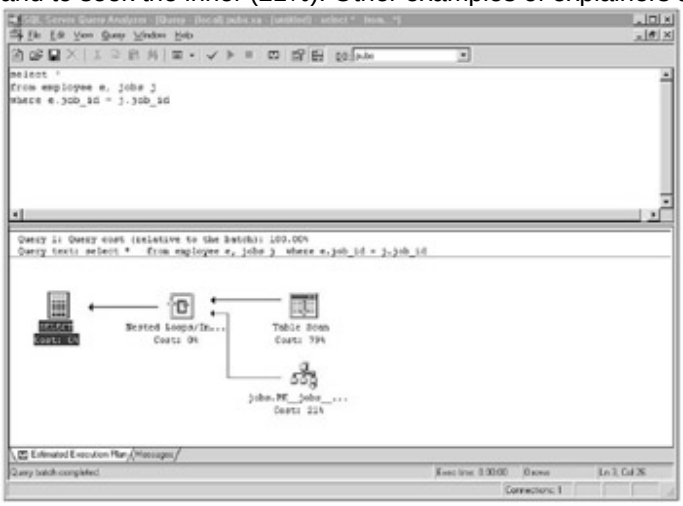


Figure 7.5: SQL Server's Query Analyzer.

Table 7.1: Example of query access plan explainers in some DBMS products

	VISUAL PRESENTATION	TEXTUAL PRESENTATION	STATISTICS PRESENTATION
DB2 UDB			
Visual Explain tool	▪		▪
explain command		▪	
SQL Server			

Table 7.1: Example of query access plan explainers in some DBMS products

	VISUAL PRESENTATION	TEXTUAL PRESENTATION	STATISTICS PRESENTATION
Query Analyzer tool	▪		▪
set showplan command		▪	
Oracle			
SQL Analyze tool	▪		▪
explain command		▪	

A gallery of example queries with their query plans and commentary is presented in Appendix D.

7.2.2 Performance Monitors

Performance monitors are the generic name given to the tools that access the DBMS's (and OS's) internal counters and gauges, and compute performance indicators.

DBMS and OS performance monitors have several flavors. For instance, to check "CPU usage" on Windows 2000, you can use the OS's performance monitor itself or the Task Manager tool; in a Unix environment, your choices range from a simple vmstat or top command to a more complete, sophisticated tool as sar, not to mention the graphical performance monitors. Similarly, a DBMS "cache-hit ratio" (cache efficiency indicator) can be monitored either using a supplied performance monitor—sometimes the OS's performance monitor itself—or with simpler utilities such as a select on the sysperfinfo system table on SQL Server, or through a query on the V$SYSSTAT view in Oracle, or with a get snapshot for all bufferpools command on DB2, to mention a few.

The following points differentiate performance monitors and ultimately should influence your decision in choosing one:

- The scope of the indicators that they access. A general performance tool can access every relevant indicator from both the operating system and the database system. More specific tools access a smaller group of correlated indicators. You should pick the general one when you are interested in how several distinct indicators behave together; otherwise, you could use the simpler tools.
- The frequency of data gathering. There are mainly three choices: *snapshotting, regular frequency monitoring*, and *threshold monitoring*. Snapshotting performance indicators allow you to freeze a situation at a specific moment in time. Choose a tool able to snapshot indicators if you know when the snapshot should be taken and where the situation happens, for example, to analyze a disk subsystem that presents peak utilization at specific moments[2] or to capture data regarding a query that has irregular response times during known periods of the day. Regular frequency monitoring means snapshotting at regular intervals, and it allows you to follow the variations of a single indicator or several different indicators over time. Tools that implement this method are useful both when confronted with a specific problem and when doing routine monitoring. Threshold monitoring allows you to enter thresholds beyond which the system is acting abnormally. Choose a tool capable of threshold monitoring whenever you need to automatically verify that critical indicators stay within acceptable thresholds.

- The presentation of the data. Data can be presented either graphically or textually. You should opt for a graphical presentation tool whenever you want to see the data trends, and the text presentation tool whenever you want just the precise indicator's values.
- The storage of the data. You should pick a tool able to save data whenever you want to defer the analysis to a later time or if you want to collect large amounts of performance analysis data in just one session. (Sometimes a ten-minute data-gathering session can yield enough information to analyze for a day.)

An example of a graphical, general performance monitor is given in Figure 7.6. The top half of the screen allows you to choose which indicators you want to track. By selecting Average Physical Read Time and clicking the Graph button, you can start monitoring this indicator, as the bottom half of the screen shows. While the last, average, minimum, and maximum values of the indicator are shown on the left side, the chart on the right represents the real-time evolution of this indicator. Note that the chart is qualitative—there are no scales on the *y*-axis—and it is divided into five stripes. Whenever the value of an indicator is considered normal, it is plotted on the middle stripe. The stripes immediately above and below the normal one correspond to alarm or warning stripes. The stripes on the top and bottom of the chart indicate emergency situations. The threshold values that determine when a situation is an alarm or an emergency are on the right side of the chart. Other performance monitor tools appear in Table 7.2.

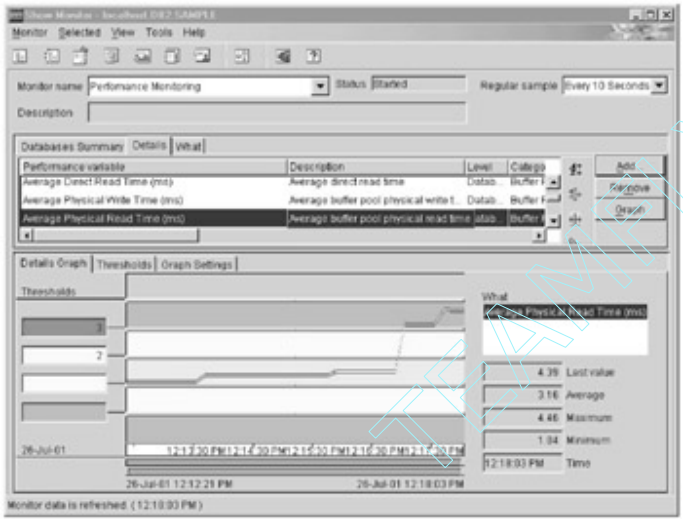


Figure 7.6: DB2 UDB's Shows Monitor tools.

Table 7.2: Example of performance monitors in some DBMS products

	SCOPE: GENERIC SPECIFIC	FREQUENCY: SNAPSHOT REGULAR THRESHOLD	GRAPHICAL PRESENTATION	SAVES COLLECTED DATA
SQL Server				
OS Performance Monitor	G	RT	▪	▪
sp_*procedures	S	S		
Oracle				
Performance	G	R	▪	▪

Table 7.2: Example of performance monitors in some DBMS products

	SCOPE: GENERIC SPECIFIC	FREQUENCY: SNAPSHOT REGULAR THRESHOLD	GRAPHICAL PRESENTATION	SAVES COLLECTED DATA
Manager				
V$* views	S	S		
DB2 UDB				
Show Monitor	G	RT	▪	
get snapshot command	S	S		

7.2.3 Event Monitors

Event monitors record DBMS's performance measurements, but only when a system event happens. A system event can be anything from a new user connection to the beginning or the end of the execution of an SQL statement. An event monitor will log the event's time, duration, and associated performance indicator values. For example, the end-of-statement execution event carries information about the text of the statement, the CPU time consumed in that statement execution, how many reads and writes were required, and so on.
Logging events with a monitor can extract fine-grained performance information with little system overhead.[3] The indicators extracted by an event monitor can be saved and further consolidated into specific reports. Figure 7.7 shows one example of such a report. Several CPU consumption indicators were collected along with their SQL execution associated events. This report permits the analyst to identify heavy consumers of CPU time spent in diverse internal query activities. Table 7.3 identifies tools capable of event monitoring in some of the current commercial DBMS systems.

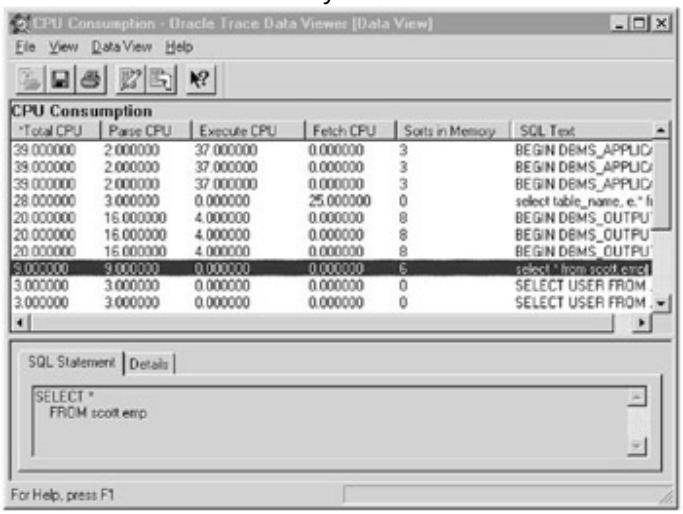


Figure 7.7: Oracle's Trace Manager (Diagnostics Pack).

Table 7.3: Example of event monitors in some DBMS products

	FILTERS EVENTS	SAVES COLLECTED DATA	CONSOLIDATES COLLECTED DATA
Oracle			
Trace Manager and Trace Data Viewer	▪	▪	▪
DB2 UDB			
Event Monitor and Event Analyzer	▪	▪	▪
SQL Server			
Server Profile	▪	▪	▪

Event monitors are particularly appropriate for capturing extraordinary conditions. For instance, you can instruct your event monitor to capture deadlock events, which you hope are rare. The monitoring report will carry information about which locks led to the conflict situation and which transactions were involved in it—a scenario you would not easily capture with snapshotting. Nevertheless, an event monitor does not replace a performance monitor because performance indicators may need to be monitored at moments when no particular event is happening.

7.2.4 Now What?

You now know the tools and you know when to use them. Roll up your sleeves and hit the indicators!
Which indicators to monitor will depend on whether you are tracking down the causes of an undesired effect or you are just doing routine monitoring. If the latteris true, proceed to a bottom-up search of the chain; that is, answer first the primary resource question (Section 7.5) and then the following two (Sections 7.4 and 7.3). If the former is true—and that is the typical case—you should begin wherever the problem is. For example, if you identified a slow query, start your analysis with the query's plan, answering the high-level consumer's question over it (Section 7.3), and then go down the chain.

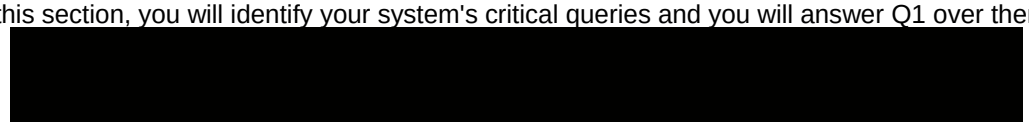
[1]Some parallel operators may have several outputs.

[2]Sometimes all it takes to find it is to look at the disk's LEDs.

[3]Little overhead does not mean zero overhead, however. Use common sense. Try not to log data that you know in advance you will not be interested in.

7.3 Queries from Hell

In this section, you will identify your system's critical queries and you will answer Q1 over them:



High-level consumer question (Q1): Are applications and, more specifically, their critical queries being served in the most effective manner?

Poorly written queries can hurt performance greatly. You had better discover them before they cause problems (or, better put, before a user brings one of them to your attention). Suspicious queries are those that (a) consume a great amount of resources (read a lot of data, perform heavy aggregation or sorting, lock an entire table, etc); (b) are executed frequently; or (c) have to be answered in a short time (e.g., queries the customer care department people run while they are talking to customers).

7.3.1 Finding "Suspicious" Queries

Although to know about (c) you need to be aware of the application's response time requirements, queries with properties (a) and (b) can be discovered by performance monitoring.

Which indicators to measure

Your DBMS event monitor logs server events, along with their relevant performance indicator values. In this particular case, you will be interested in the *end-of-statement* or in the *end-of-transaction* event, depending on your system. These events carry information such as the text (SQL) of the statement and the duration of its execution as well as CPU, IO, cache, and lock consumption statistics. This means that by capturing these events you can record exactly the resources each executed query consumed.

Considerations and how to evaluate them

By sorting the collected data over a given resource consumption statistic, you will be able to find the most expensive queries by that criterion. For instance, if you want to identify long-running queries, you can sort by the duration information; to get heavy readers, you sort by the read pages or read rows information; and so on. Frequently executed queries can be found by sorting by the SQL command text information and counting how many times a particular SQL command text was executed during the data collection period.

Keep in mind that thresholds beyond which queries are considered to be critical vary according to a number of parameters. For instance, a "batch" update that takes 30 minutes to complete may be considered normal, whereas no OLTP query should require 30 seconds to run.

What to do in case of problems

Well, if you find expensive queries, you should make sure they are tuned in the best possible way. The techniques described in Chapter 4, Sections 7.3.2 and 7.3.3, and in Appendix D will be able to help you here. Your analysis is just starting, and you might have to drill down through the consumption chain to search for the real cause of the problem.

Which tool to use and how frequently

Event Monitor tools are the best choice here. By turning on the logging of the end-of-statement and end-of-transaction events, you can obtain the query consumption data you will need in your analysis. Alternatively, there are specific tools that find top consumers in real time, like Oracle's TopSessions. Checking that expensive consumers are well tuned should be a top priority in your performance monitoring efforts.

7.3.2 Analyzing a Query's Access Plan

As shown before, an access plan is the strategy generated by the DBMS to execute the query, and without proper information, or lacking better alternatives, the DBMS can generate plans with poor performance. To assess whether the plan that the DBMS chose was a good one, compare it to your own choice, that is, a plan you would have picked had you been directing the optimizer.

Which indicators to measure

Do not even bother analyzing a plan that was generated with out-of-date statistics! If you have outdated stats, update them and regenerate the plan. Table 7.4 gives examples of such utilities in some DBMS products.

Table 7.4: Example of statistics updating utilities in some DBMS products

DBMS	TOOL
DB2 UDB	runstats command
SQL Server	update statistics command
Oracle	analyze command

Here is what the access plan tells you:

- *Access methods choices*. For every table involved on the query, determine how it is being accessed. For instance, are all its pages being scanned? (Probably bad.[4]) Or is an index used to find an entry point in the table and then a sequential scan performed? (Better.) Or is the table accessed only through an index? (Best.)
  Large scans in queries that do not aggregate data tend to produce large result sets. This sometimes suggests that the query writer might not have taken the size of the table into consideration.
- *Sorts*. Check to see if the plan includes a sorting operation. This can be used to process an ORDER BY clause and also in more subtle situations, such as to process GROUP BY clauses and duplicate elimination in DISTINCT clauses. Check also if the existence of an index would eliminate the need for the sort or if the DISTINCT could be eliminated.
- *Intermediate results*. Complex queries sometimes cause materialization of inter mediary results in a temporary area. They may result, for example, from correlated subqueries. As pointed out in Chapter 4, rewriting such queries into multiple queries can improve performance.
- *Order of operations*. For plans consisting of several operations, pay attention to which order the optimizer chose to apply them. Of particular interest is the location of the joins, sorts, aggregations, and filtering expressions. The order chosen for a query's operators aims at an early reduction in the number of tuples to process inside the plan. This is why optimizers will place selection operations in the plan's early stages. Moreover, the order of the joins should follow the same rationale. If you notice the order does not favor such a reduction (and statistics are up to date), you might have to force an ordering.

- *Algorithms used in operations*. For operations such as joins and sorts, be sure to notice which algorithm the DBMS chose to use. These operations are usually named after the algorithm chosen to process them.

Considerations and how to evaluate them

If you had envisaged a plausible access plan before starting the analysis, you can now assess the optimizer's choice of plan and compare how it differs from yours. If the optimizer's plan seems to be more complex than yours, and the query is performing badly, you have probably caught a major source of the problem.

7.3.3 Profiling a Query Execution

Expensive queries that already happen to have efficient access plans should have their execution profile carefully analyzed. A query execution profile consists of detailed information about the duration and resource consumption of its execution.

Which indicators to measure

The duration information involves three indicators: the *elapsed time* for the query, which is the time it took to process it as perceived by a user; the *CPU time*, which is the time that the CPU was actually used to process the query; and the *wait time*, which is the time the query was not processing and waiting for a resource to become available (e.g., a data page to be read, a lock to be acquired).

Resource consumption information includes

- I/O. *Physical and logical reads* and *physical and logical writes*, where a logical read/write consists of the operation to access or change a page, and a physical read/write consists of a logical read that requires a disk access.
- Locking. The *maximum number of locks held* and the *number of lock escalations* can provide an idea of how greedy a transaction or an SQL statement is in terms of data locking. The *number of deadlocks and timeouts* and *total time spent waiting for locks* will quantify the overhead involved in the locking activity.
- SQL activity. *Number of sorts* and *temporary area usage* provide a measure of time spent in expensive overhead activity.

Considerations and how to evaluate them

Two scenarios are common. In the first, the elapsed query time is close to the CPU time. The wait time is negligible if compared. The consumption measured seems fair for what the query does: only a reasonable number of data pages are being accessed to answer the query, and most of them are logical accesses; the number of locks is low or nonexistent (it is possible, depending on the isolation level used), and there were no deadlocks or lock escalations; if sorts were performed, they did not seem to augment the number of physical disk reads and writes, indicating that they were performed in memory. Well, in this case, it seems that the access plan is being executed in the best way possible—no problem is identified and if the rest of the chain is balanced, this probably represents the best performance your system can deliver for this query.
In the second scenario, we find a noticeable discrepancy between the elapsed time and the CPU time. The wait time seems to just about fill the gap between them. In this case, you can be sure to find some problem in resource consumption, either a contention problem (see Figure 7.2a) or a poorly performing resource (see Figure 7.2b). With a logical contention problem, such as a concurrently accessed table, the query was probably waiting for locks.

Physical resource contention requires operation-by-operation analysis. Suppose the area that the DBMS allocated for sorting was small. You are going to get additional I/O activity whenever a query performs a sort—which at first glance may be difficult to associate with the sort area problem.

One way to distinguish locking contention from physical resource problems is to run the query in isolation. Badly parameterized subsystems leading to poor use of physical resources will produce consistently slow response times no matter whether a query runs alone or not. Concurrency problems manifest themselves in highly concurrent environments.[5]

Which tool to use and how frequently

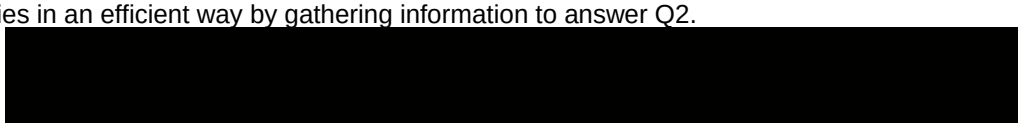
Obtaining all indicators mentioned here in full detail is sometimes called "accounting." In systems where such a concept exists, you will find appropriate tools for the task (a notable example is DB2 PM for OS/390). Other tools that might provide partial information are the db2batch (benchmarking tool) for DB2 UDB, the trace facility, TKPROF, and SQL Analyze for Oracle, and the SET STATISTICS TIME and SET STATISTICS I/O for SQL Server.

[4]As a rule of thumb, this is true. However, some examples exist in which table scans are a good choice (see Appendix D).

[5]Tautologies have the advantage of being true.

7.4 Are DBMS Subsystems Working Satisfactorily?

In this section, we are going to check whether the main DBMS subsystems are performing their duties in an efficient way by gathering information to answer Q2.



Intermediate consumer/resource question (Q2): Are DBMS subsystems making optimal use of resources so as to reach the desired performance levels?

Poorly configured subsystems may be the source of several performance problems that manifest themselves as a noticeable reduction on the workload a machine could have handled had the DBMS subsystems been correctly configured.

7.4.1 Disk Subsystem

As discussed in Chapter 2, a DBMS disk subsystem will perform well if the database's tables are well distributed. A table whose rows are uniformly distributed among several disks where they occupy contiguous pages will support sequential prefetching and parallel I/O. A table in which good-sized chunks (bigger than an average row) of free space are easily located will foster fast insertions or row-expanding updates. A table whose rowIDs are accurate (rows were not displaced) and clustered in the sequence in which they are usually accessed will probably have optimal cost selections. Finally, a balanced position of data files across the available disks will avoid contention by making the best possible use of such resources.

Monitoring the disk subsystem will help you check that data contiguity, row placement, free space organization, and file positioning are providing the necessary support for fast performance.

Which indicators to measure

Data contiguity can be evaluated by counting the *number of fragments* each table or tablespace is divided into. *Percentage of row displacement* is the ratio between the number of rows that do not reside where their rowID indicates and the total number of rows. *Free space fragmentation* is a measure that will tell you how large free space chunks are.
File positioning can be evaluated by breaking down the measurement made in Section 7.5.2 for disk transfer per second or by getting the details on *pages read/written* by tablespaces (or even by table in some systems).

Considerations and how to evaluate them

Ideally, the space occupied by a table on a physical disk should be contiguous (i.e., its number of extents[6] should be 1). Having five or fewer extents will not noticeably harm performance, but having, say, 50 will surely cause nonnegligible additional disk arm activity—let alone missing prefetching opportunities—if the whole table is to be retrieved.

The number of displaced rowIDs should be kept at lower than 5%. Here, too, values greater than that will force the database to visit noticeably more pages than necessary to retrieve a displaced row.

Measuring free space fragmentation can be done as follows. The percentage of pages that do not have contiguous space for at least one average row but are listed as having free space should be zero.

Pages read/written values should be grouped by disks and present balanced values.

What to do in case of problems

Most of the time bad indicators reflect inadequately chosen storage parameters. A few exceptions could come from poorly designed transactions that, for instance, insert a row with a null value into a column and then update that value right away, possibly causing a row to be displaced near birth!
In order to choose appropriate storage parameters, you should understand the pace at which data grows and with what mix of inserts/updates/deletes that growth rate happens (data volatility). In addition, you should identify occasional but predictable fluctuations in the database activity due to the nature of the data or processes being supported by the applications (data seasonality). With such knowledge, you could then refer to Chapter 2, where physical disk and disk subsystem tuning parameters are discussed.

Which tool to use and how frequently

In most systems, contiguity information is part of the catalog. In Oracle, the number of extents can be obtained through the DBA_SEGMENTS view. In DB2, the catalog table SYSTABLEPART stores space utilization statistics, including the number of secondary allocations used. Systems that do not store such information in the catalog usually provide specific diagnostic tools. For instance, in SQL Server, DBCC showcontig provides the necessary information.

Row placement and free space fragmentation diagnostic utilities have different names in the different systems: in DB2, the tool is REORGCHK; in Oracle, it is part of the information generated by the STATSPACK; and in SQL Server, it is DBCC showcontig.

For the monitoring of the disk activity indicators the DBMS's performance monitor is usually all you need. For instance, Oracle's STATSPACK can periodically extract V$FILESTAT information; DB2 Performance Monitor can extract disk performance indicators on a tablespace or on a table level. You should monitor disk activity regularly to avoid halting the database from a lack of space—one of the few things that can make a whole system stop.

Keep in mind also that data file reorganization is a periodical necessity. Unfortunately, not every system offers you a reorganization utility. For those that don't, reorganizing a table should be done by carefully unloading, dropping, re-creating—with new storage parameters—and reloading it.

7.4.2 Buffer (Cache) Manager

Buffer management is a central issue in a DBMS's performance because caching reduces the necessity for disk operations. Whenever a request for a page is filled through the database buffer (also known as the database cache), a physical I/O is saved. The buffer manager tries to balance the opposing needs of having often accessible pages in memory while keeping free slots for other pages.

A well-tuned buffer manager can dramatically improve the performance of the whole system and, as such, must be the focus of thorough and constant monitoring.

Which indicators to measure

The two main performance indicators you should monitor are the *cache-hit ratio*, which reflects the percentage of times that a requested page was already in the buffer, and the *number of free pages*, which tells how much space is available in the buffer at a given moment. In some DBMSs, specific buffers could exist for different purposes: data buffers (table and index pages), dictionary buffers (metadata information), procedure buffers (code pages), and so on. If that is the case, check both of the indicators for each of the specialized buffers.

Considerations and how to evaluate them

The cache-hit ratio should ideally be greater that 90%. A lower cache-hit percentage is one of the most obvious signs that the system either is not well tuned or does not have enough memory. It can be the case, however, that some queries do not achieve a good cache-hit percentage—especially the ones that scan very large tables.

The DBMS clean-up threads should also provide a steady stream of free pages. Failure to obtain this might indicate that the system is not maintaining free page slots in the buffer, or the disk subsystem is not able to provide the level of service needed.

What to do in case of problems

If your indicators have low values, your buffer manager is not saving as many disk accesses as it should. If this occurs on occasion, then one or a small number of queries may be responsible. Otherwise, you may have a slow disk subsystem. It is also possible that you have insufficient memory.
Tuning parameters to raise the cache-hit ratio and to provide the necessary level of free pages were discussed in Chapter 2, and you should refer to that chapter for complete information. As you may recall, increasing the size of the buffer without increasing the size of physical main memory may lead to paging—a particularly inefficient form of input-output because it involves a lot of random access.

Which tool to use and how frequently

Both cache hits and free pages can be obtained via your DBMS performance monitor tools. Usually, you would like to have their values monitored regularly as well as be notified whenever they fall under thresholds you consider normal (such as 90% for the hit ratio).

7.4.3 Logging Subsystem

The logging subsystem supports failure recovery. The "log" is used indirectly by every transaction that alters, inserts, or deletes data because these changes are recorded and log records are always saved before their corresponding data pages (using, for example, the write-ahead log protocol discussed in Chapter 2).

A well-tuned logging subsystem is one that can write log records at a pace that does not slow active transactions.

Which indicators to measure

To confirm that the disks storing the log files are keeping pace with the transaction workload, you should check the *number of log waits*. It is also important to know if there were any *log expansions* or *log archives* due to lack of space.
To assess whether the size of the log buffer is adequate, you should check the *log cache-hit ratio*. Like its data buffer counterpart, this is the ratio between logical reads and total reads.

Considerations and how to evaluate them

Recording log waits greater than zero means that transactions are being held longer than necessary to write those log records at commit time. Recording dynamic log expansions indicates a configuration problem. Under an ideal configuration, log expansions can be avoided by coordinating log archival, or alternatively log truncation, with the backup procedures. Here is how. When data pages are made redundant by the backup utility, the log records referring to changes to those pages no longer need be kept. At that point, the log can be truncated so as not to include them.

Which tool to use and how frequently

Your DBMS performance monitor is the best way to access the log subsystem indicators. Keep routine monitoring here since this is one of the ways to identify overall fluctuations of database activity.

7.4.4 Locking Subsystem

Locks are the basic mechanism allowing transactions to run concurrently without harming one another. When transactions are not well designed or when the lock sub subsystem is not well parameterized, contention problems like long lock waits, eventually leading to timeouts and deadlocks, may appear.

Which indicators to measure

Some parameters can be used to check the overall efficiency of locking in your system. The *average lock wait time* and the *number of locks on wait* will tell you how much time transactions spend waiting for lock requests to be confirmed and how many locks there are in such a situation.
Deadlocks can be found by checking the overall *number of deadlocks or timeouts* for the system.

Considerations and how to evaluate them

Ideally, the lock wait time for a transaction should be a small fraction of its *total transaction time*, and the number of locks waiting should be low if compared to the *total number of locks held* by transactions, certainly under 20%.

Deadlocks and timeouts are particularly harmful to the system since they cause transactions not only to hold locks longer, but also eventually to roll back whatever changes they had made before the problem was detected.

What to do in case of problems

The lock subsystem is often a reflection of how well your applications' transactions were designed, and seldom a source of a problem itself. If you are experiencing high values of wait times and locks requests on hold, you can move up the consumption chain without even finishing the evaluation of other DBMS subsystems. Section 7.3.1 explains how to find expensive consumer queries, including data locking ones. Deadlock-involved queries can be found with the tools described next.

Which tool to use and how frequently

The best-suited tool to monitor for deadlocks is an Event Monitor. Usually deadlock is a traceable event and whenever recorded carries with it information about offending locks and transactions involved.

DBMS performance monitor tools are ideal for checking or monitoring locks. Breaking down the number of locks held is usually possible using specific lock monitoring tools. SQL Server has an sp_lock procedure, Oracle has a V$LOCK view, and DB2 has a get snapshot for locks command.

[6]The term *extent* may mean different things in different systems. Here (and this is the common meaning) an extent is an arbitrary number of pages that are allocated contiguously.

7.5 Is the DBMS Getting All It Needs?

In this section, we check the DBMS resource consumption from an OS point of view by answering the primary resources question.

Primary resources question (Q3): Are there enough primary resources available for DBMS consumption and are they configured adequately, given the current and the expected operation workload (number of users, types and frequency of queries, etc.)?

Finally, we ensure that CPU, disks/controllers, memory, and network resources are adequately configured for the DBMS.

7.5.1 Checking on CPU

Monitoring the CPU allows you to determine how much of its total capacity is consumed in general and how much is used by the DBMS in particular.

Which indicators to measure

The main indicator to measure CPU activity is the *percentage of utilization*. This indicator is computed by subtracting the percentage of time the CPU was running the idle process from 100%. A busy CPU can be computing either on a user's behalf (e.g., running user programs) or on the OS's behalf (e.g., performing OS functions such as context switching and servicing interrupts). As their names suggest, the *percentage of user time* and *percentage of system time* represent the fractions nonidle time spent in each mode. If you have multiple CPUs, these measures still apply, but you will need the overall consolidated values as well.

Considerations and how to evaluate them

Sustained CPU utilization over 70% should get your attention. If you are getting it outside a heavy batch processing period of the day, you have a problem.

System utilization greater than 40% means that the operating system is spending too much time on internal work.

In a nondedicated DBMS server, you should also pay attention to the CPU time-share that the DBMS is using. This can be done by verifying that the DBMS processes are getting the biggest share of the CPU.

Ideally, each processor in a parallel machine should have the same utilization. If not, then look carefully at the overloaded processors. Verify that DBMS processes are responsible by breaking down CPU usage for each individual process. If so, then determine which queries are responsible.

Which tool to use and how frequently

Operating systems offer specific CPU monitoring tools to break down the CPU utilization in real time (e.g., top in Unix and Task Manager in Windows 2000). It is a good idea to monitor the CPU constantly to learn when and why variations happen.

7.5.2 Checking on Disks and Controllers

Disk operations are several orders of magnitude slower than RAM operations, so long disk queues can hurt response time substantially.

Which indicators to measure

In order to check disk usage, you should monitor, for each physical disk in the system, the *average size of the waiting queue*, which tells how many disk requests were waiting to be served on the average while this disk was busy serving another request, and *average time taken to service* a request, which expresses how long, again on average, these requests waited in the queue.
Along with those indicators, you should also check the number of *bytes transferred per second* from each of the disks. This parameter sums up everything read or written to a disk per unit of time.

Considerations and how to evaluate them

Both the average size of the waiting queue and the average time taken to service a request should be close to zero. Long queues and long service times indicate that new requests are waiting. If you see long queues or service times, check the bytes transferred per second indicator to confirm that the disk is truly busy. Idle disks with pending requests imply that you have controller problems. Some controllers allow a number of devices to be attached, but only one data transfer at a time. If you attach two frequently used disks to the same controller, they might end up competing for the sole data transfer channel.

Another analysis that can be done with these values is to compare the waits from each disk. If you have a group of uniform disks serving the same purpose (e.g., a disk for storing tables and indexes), they should support approximately the same volume of data transfer and have equal-length queues. Otherwise, you are not dividing the disk access load evenly.

What to do in case of problems

Solutions to disk problems are discussed in Chapters 2 and 3, and mainly involve moving data among disks. However, here again, you should reorganize data only if you are sure the I/O problems are not coming from high-level consumer inefficiency. For example, a user data disk may be overloaded because a large table stored on it is erroneously being scanned. Moving the table will only transfer the overload, not solve the problem. Overload on system data disks (temporary, sort, or log areas) should direct your attention to the corresponding subsystem.

Which tool to use and how frequently

You can measure disk utilization using operating system utilities such as the Unix monitor called iostat.

7.5.3 Checking on Memory

At the operating system level, the most significant memory problem that can affect a DBMS is paging. Page faults in database processes mean their images cannot fit in the real memory if pages are not swapped out.

Which indicators to measure

Paging can be detected by analyzing the indicators' *number of page faults/time*, which tells you whether paging is happening at all, and the *percentage of paging file in use*, which reflects how much more real memory would be necessary to eliminate paging.

Considerations and how to evaluate them

Ideally, in a dedicated DBMS machine, there should be no paging. This means that the memory allocated to database buffers would not page at all. Well, ideally. In reality, paging happens quite often and there will be no problem with it as long as (1) data (cache) buffers are not paged—obviously the database cache should be configured to fit comfortably in main memory along with the database processes themselves—and (2) paging does not escalate to thrashing.

Thrashing occurs whenever the sum of the memory working set of all processes of the systems significantly outgrows the available real memory. Detecting thrashing is quite straightforward.[7] An extremely high disk I/O activity and a noticeable slow performance characterize it.

What to do in case of problems

Assuming you have properly configured your database buffer size, paging should come from non-DBMS sources. You should check those.

Which tool to use and how frequently

Again, operating system tools are helpful, such as the vmstat tool in Unix systems. Checking memory at the operating system level should be done whenever you change the memory allocation parameters of the DBMS or there are significant changes in the number of users.

7.5.4 Checking the Network

Network transmission time should represent a negligible fraction of query response time. The goal of assessing network performance is to verify that its speed and delay are negligible.

Which parameter to measure

To rule out network overload from a problem scenario, measure the *number of collisions* and the *average network interface queue size*.
Collisions happen in a broadcast technology network when two or more devices try to transmit at the same time. This might represent an important source of network delay. In addition to these indicators, you can measure the *round-trip time* between client stations and the database server.

[7]If you go near the machine whenever it thrashes, you can literally hear the uninterrupted disk activity.

7.6 Conclusion

A well-tuned DBMS is one in which query plans are efficient, subsystems make optimal use of resources, and these resources are available in sufficient quantity. Checking whether this is true entails first discarding the possibility that excessive workloads come from untuned queries. Then comes assuring that there is capacity enough on the DBMS machine to handle the resulting workload and checking that the DBMS's subsystems are adapted to this workload's nuances.

The three questions approach is a framework that makes testing these hypotheses a systematic procedure. It introduces a consumption chain paradigm that helps distinguish the causes of performance problems from its symptoms. It points out the most relevant performance indicators of a DBMS, which with the available tools can be easily measured. Finally, it suggests how to analyze the measured values so as to determine whether they represent normal situations or not.

Bibliography

Donald K. Burleson. *Oracle High Performance Tuning with STATSPACK.* Oracle Press, 2001. This is a comprehensive guide to regular performance monitoring and tuning using the STATSPACK tool.

Ramez A. Elmasri and Shamkant B. Navathe *Fundamentals of Database Systems.* Addison-Wesley, 1999. Provides a thorough introduction to query processing.

Nigel Griffiths, James Chandler, Joao Marcos Costa de Souza, Gerhard Muller, and Diana Gfroerer. *Database Performance on AIX in DB2 UDB and Oracle Environments.* IBM, 1999. DB2 tuners (as well as Oracle's) can benefit from this “Redbook” on performance tuning. It brings an interesting classic list of tuning mistakes and a compilation of which database parameters will make a large difference in performance.

IBM. *Administration Guide V7.1, Volume 3: Performance (SC09-2945-00).* This comprehensive performance tuning guide covers DB2's architecture, monitoring tools, and other tuning aspects.

Oracle. *Oracle 9i, Database Performance Guide and Reference (A87503).* This manual covers the use of Oracle's main monitoring tools, including basic usage of the STATSPACK tool.

Oracle. *Oracle Enterprise Manager, Getting Started with the Oracle Diagnostics Pack (A88748-02).* Covers Oracle's more advanced monitoring and performance diagnostics tools.

Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts.* McGraw-Hill Higher Education, 2001. This well-written popular book provides a functional description of query plans, as do many of the better introductory database textbooks.

Edward Whalen, Marcilian Garcia, Steve Adrien DeLuca, and Dean Thompson. *Microsoft SQL Server 2000 Performance Tuning Technical Reference.* Microsoft Press, 2001. General performance tuning guide where SQL Server tuners will find how to use the product's monitoring tools.

Chapter 8: Tuning E-Commerce Applications

8.1 Goal

The goal of this chapter is to describe the fundamental architecture underlying every e-commerce system and to discuss the tuning considerations that apply to that architecture. Even if your system has added bells and whistles on top of these services, you would do well to start tuning these fundamentals. We conclude the chapter by discussing capacity planning.

8.2 E-commerce Architecture

E-commerce applications often have a three-tiered architecture consisting of Web servers, application servers, and database servers (Figure 8.1).

1. Web servers are responsible for presentation. They call functions from the underlying application servers via server extensions such as servlets or dynamic HTML interpreters such as ASP. They deliver HTML or XML pages back to the client browsers.
2. The application servers are responsible for the business logic. They implement the functions exposed to the clients. Typically, these functions include search, update shopping cart, pay, or create account. Each function uses data from a local cache or from the underlying database server and outputs HTML or XML pages.
3. The database servers perform data access.

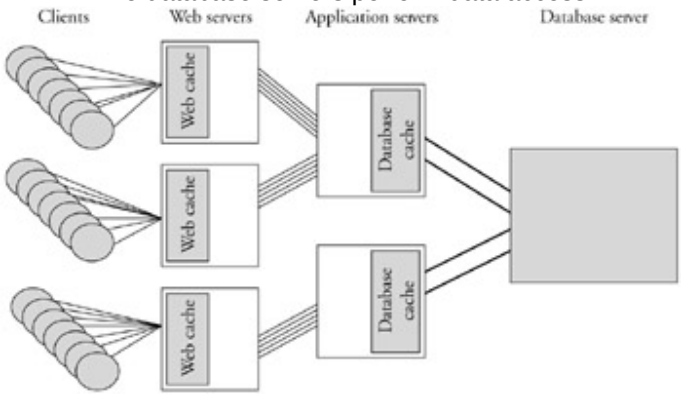


Figure 8.1: E-commerce three-tiered architecture. The three-tiered architecture comprises Web servers, application servers, and database servers. Web servers might cache Web pages (Web cache). Application servers might cache database relations (database cache).

The question for the database tuner is to fix bottlenecks at the interface between the application servers and the database servers or within the database servers. This is critical for these applications because sites that display pages in nine seconds lose 10% more customers than sites that display pages in eight seconds according to anecdotal reports.

To achieve low response time, you need to understand the functions provided by the application servers and the queries that they submit to the database servers. We have attempted to characterize the queries in a way that is relevant to the interface designer as well as to the tuner.

1. Touristic searching—access the top few pages but don't go beyond them. Pages may be personalized to promote special offers based on the history of products purchased by the customer. These queries can be very frequent. Their content can largely be cached (even "Akamized," i.e., cached at the edges of the Internet) except perhaps for the personalization piece. There is no need for transactional guarantees with respect to available stock. But timeliness (e.g., reflecting a close approximation to the stock available that day) is important. Thus, the cache may be refreshed every hour or even every day, for example.

2. Category searching—down some hierarchy, for example, men's clothing to hunting boots (frequent, partly cached, and need for timeliness guarantees). Again, absolute transactional guarantees are not required.
3. Keyword searching (frequent, uncached, and need for timeliness guarantees). The index on which the keyword searching is based need not be transactionally up to date because the search usually pertains to descriptive information.
4. Shopping cart interactions (rare, but transactional). Such interactions include marketing functionality such as displaying a list of related products after a product has been purchased (infrequent, cached, and need for timeliness).
5. Electronic purchasing (rare, but transactional).

In addition to the query breakdown, the database tuner must consider a few design issues common to e-commerce sites:

- Need to keep historic information. Pierre-Yves Gibello from Expershop notes that a noticeable fraction of the electronic payment acknowledgments get lost. An e-commerce site must keep historic information about all the orders that are passed in order to compare them with the reports from the bank. In addition, this historic information can be mined in order to establish links between related products.
- Preparation for variable load. Accesses to e-commerce Web sites (or to portals in general) seem to follow regular patterns within a day and within a week. Tom Barclay from Microsoft research reports his experience with the TerraServer Web site: the Web site is busiest in the mornings (morning for the people who access the site, i.e., 11 P.M. to 3 A.M. PST for European users, and 5 A.M. to 3 P.M. PST for American users). Similar experience with intranet portals suggests that Mondays are the busiest day in the week. As a result, there are distinct peaks in the number of accesses. The system has to be sized for those.
- Possibility of disconnections. State information can be transmitted to the client (e.g, using cookies) so that customers can keep their shopping cart across connections to the e-commerce site.
- Special considerations for low bandwidth. Not everyone has a 10 Mbyte/sec connection to the Internet. Many home customers as well as companies that operate intranet portals over leased lines can afford only limited bandwidth. On such lines, the HTML traffic might become the bottleneck (e.g., font information may be contained in each column of a table). Pictures may be particularly painful. So you should design your Web site with options for low-bandwidth users.
- Schema evolution. The conceptual schema for representing items will evolve in order to represent new products. Creating new tables to reflect this evolution requires the definition of new queries and possibly the modification of existing ones and is thus expensive. A simpler solution is to add attributes to existing relations. This leads to the definition of a few tables with a large number of attributes. This design presents two main problems: (1) the number of attributes in a table is limited (to about 1000) and (2) tables with a large number of attributes have NULL values in most fields. In order to deal with schema evolution, a few systems, such as IBM WebSphere Commerce Server, propose to represent e-commerce data as attribute-value pairs, that is, tables with a vertical three-ary schema: objectid, attribute, value. The objectid is used to recompose horizontal records. This representation, however, complicates the formulation of queries. Agrawal et al. have shown that an intermediate layer between the application and the database system can hide this complexity and yield good performance.[1]

[1]R. Agrawal, A. Somani, and Y. Xu. "Storage and Querying of E-commerce Data," *Proceedings of the 27th VLDB Conference*, Rome, 2001.

8.3 Tuning the E-commerce Architecture

There are three main aspects to consider when tuning the database supporting an e-commerce site: caching, connection pooling, and indexing.

8.3.1 Caching

Intuitively, you should cache hard-to-generate data that is read often and written infrequently. Formally, we break this down into four properties of a data item: its frequency of access, the space it takes, the space available, and how often it might be invalidated. Let us review these parameters for the query types identified earlier.

Intuitively, the pages that the tourist sees should be visually appealing (lots of pictures) and very inviting. Most people (about 70%) see only these. They need not be changed as a function of what was sold that day. For example, even if the store has no more of some hot item, the touristic Web pages need not reveal that fact. This is not to say that those pages are entirely static, but they change rarely enough that caching of these HTML/JPEG/Shockwave pages on the Web server is a big win.

Category searching is the practice of going down the hierarchies or hetarchies (intersecting trees) of an e-commerce site. Hot categories are good candidates for caching on the application server. The timeliness constraints amount to saying that invalidation need not happen immediately but maybe only periodically. Thus, if an item goes out of stock, a Web page may still indicate availability. The buyer will find out the item is out of stock (and may be given a later delivery date) upon purchase. The net result is that searches through the catalog depend either entirely or largely on start-of-day data and so can be cached.

Keyword search relies on full-text indexing capabilities provided by database servers, or by specific indexing engines with database interfaces. As a result, the keyword search itself does not benefit from caching on the application server. Frequently accessed Web pages, however, might be cached based on a least recently used replacement policy or something similar.

Shopping cart interactions require the maintenance of a table relating shopper to item, quantity, and price. This table changes frequently: items are added and removed, and shoppers are removed when their purchase goes through. These update transactions are directly performed on the database server. The shopping cart table is, nevertheless, a good candidate for caching because it is accessed frequently. Even large shopping cart tables should fit in memory on the application server site. (Counting 100 bytes per record and a million entries in the shopping cart table amounts to 100 Mb.) The data describing the user profile is not updated frequently and may be cached as well.

The architecture of e-commerce applications allows different technologies for caching.

- *Web cache:* Static Web pages or fragments of dynamically created Web pages can be stored as HTML files in the portion of the file system accessed by the Web server or even outside the firewall in an Akamized fashion. Research has shown that caching fragments of dynamically created Web pages substantially reduces server-side latency. Indeed, for dynamic Web pages, the generation of HTML or XML pages dominates response time.[2]
- *Database cache:* Front-Tier from TimesTen provides supports for caching database records in an application server. This product uses *materialized views* to represent cached data. A view is either a restriction on a database server table or the result of a complex query. A materialized view is stored in the cache. Most read-only SQL queries submitted by the application server are evaluated using these views without contacting the database server. Modifications are submitted directly to the database server. The result of these insertions, updates, or deletions is then propagated to the

cached views in order to keep the cached data consistent.[3] A similar result can be achieved with Oracle 9i using remote aggregate view maintenance. An Oracle 9i server is colocated with the application server and acts as the database cache. On this server, materialized views are defined based on tables from the remote database server. The application server submits the queries to the local Oracle 9i server and the updates to the remote Oracle 9i server. The built-in replication mechanism ensures that remote materialized views are maintained consistent with the tables on the database server. A mechanism for rewriting queries using views allows queries to make optimal use of the data cached in the materialized views.

SYSTEM FEATURES FOR FAST CACHES

[redacted]

Because (1) items can be returned up to hundreds of times faster from the cache than by being built and (2) some items such as shopping carts are updated frequently, it is useful to support an updatable cache. In one architecture, Front-Tier from TimesTen allows the following behavior:[4] As a shopper adds products to a shopping cart, those updates are recorded in Front-Tier's cache (an in-memory database cache), but are not propagated to the backend database server until the transaction commits. At commit time, Front-Tier pushes the group of changes to the backend database for server-side processing. When the backend database server commits, Front-Tier reports to the user that the purchase has completed.

If there are many caches, they may have to be coordinated (unless we can force a given shopper always to return to the same cache). Front-Tier allows peer-to-peer replication.

Finally, high-performance cache servers like Front-Tier run coresident with application servers. This avoids interprocess communication and permits the application server code to have direct access to the Front-Tier data structures.

[redacted]

8.3.2 Connection Pooling

To avoid thrashing, database servers are limited to a number N_{max} of concurrent connections, which is usually lower than the number of concurrent connections accepted by a Web server or an application server. Connection pooling is a general client-server technique for multiplexing Web server connections over database connections (see Chapter 5).

Connection pooling consists of reusing a fixed set of connections between the client and the server. First, a pool of N_{min} connections is established. The first N_{min} concurrent clients use these already opened connections; if additional clients request connections to the server, then additional connections are opened until the pool reaches its maximal size N_{max}. Additional clients that request connections must wait until already connected clients terminate their transaction and give their connection back to the common pool. This general technique was developed for transaction monitors in the 1970s and has been rediscovered in the context of e-commerce applications (Figure 8.2).

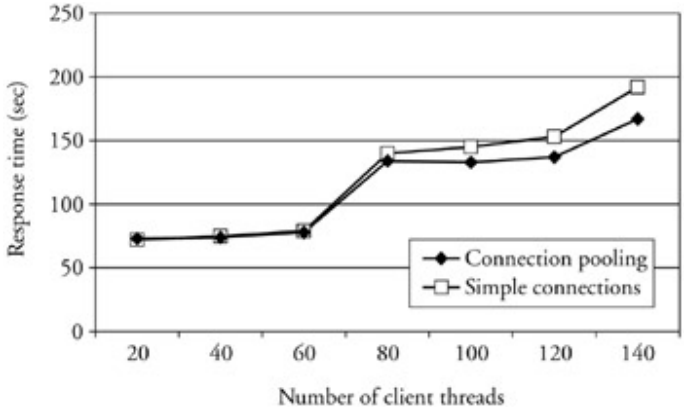


Figure 8.2: Connection pooling. For this experiment, we compare the response time using simple connections and connection pooling on Oracle 8i on Windows 2000. (Connection pooling as we have described it in this section is called connection concentration by Oracle.) We vary the number of client threads; each thread establishes a connection and runs five insert transactions. If a connection cannot be established, the thread waits 15 seconds before it establishes the connection again. The number of connections is limited to 60 on the database server. Using connection pooling, there is no problem establishing a connection; the connection requests that cannot be granted are queued and serviced whenever a connection becomes available. Using simple connections by contrast, connection requests cannot be granted when the number of client threads is greater than the maximum number of connections (60). When rejected, client threads have to wait before they try again to establish a connection.

8.3.3 Indexing

Shopping cart interactions require the maintenance of a table relating shopper to item, quantity, and price. This table changes frequently: items are added and removed, and shoppers are removed when their purchase goes through. So the queries are add row after checks on inventory, change row, remove row, bring up current shopping cart.
A clustered index on shopper id helps in two ways. First, the index speeds up shopping cart lookup. More important, a clustered index allows the system to exploit row level locking. In the absence of a clustered index, table locking is necessary to guarantee a serializable isolation level. The clustered index together with key range locking gives high concurrency among queries, insertions, deletions, and updates while preserving serializability. The experiment reported in Figure 8.3 illustrates this point. The overhead of maintaining the clustered index when inserting new entries in the shopping cart is negligible.

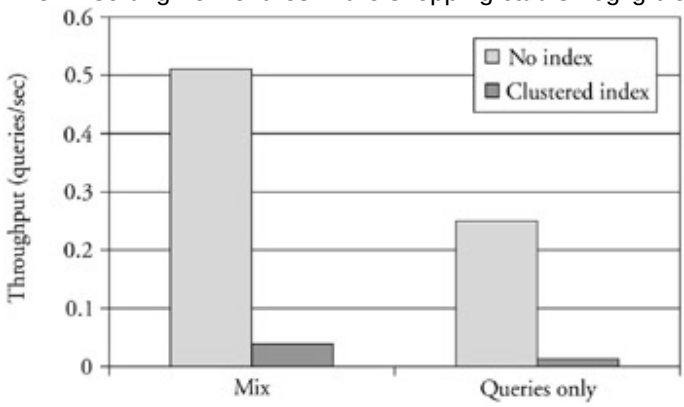


Figure 8.3: Clustering index. This graph illustrates the two benefits of a clustered index on shopper id. First, when only queries are submitted, the index speeds up response time compared to a scan (the table contains 500,000 entries). When updates, deletes, and insertions are submitted together with queries (1 delete, 1 insertion, and 10 insertions for each query), the mean

response time approximately doubles. Throughput is an order of magnitude lower in the absence of a clustered index.
Keyword search on long strings or documents requires the use of full text indexing. Most database system provide this capability. They construct inverted indexes that are used to find the documents that reference a given keyword. The database tuner should ensure that inverted indexes are located in the database server memory. Experiments in Chapter 3 illustrate the potential benefits of inverted indexes. Note that search engines with database interfaces, such as MondoSearch (which can be integrated into Microsoft Commerce Server) or Verity, construct inverted indexes outside the database server.

[2]A. Datta et al. "A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration," *Proceedings of the 27th VLDB Conference*, Rome, 2001.

[3]We surmise that future releases of this product will automatically decompose queries into a subquery using materialized views and a subquery submitted to the database server, thus taking maximum advantage of the materialized views and reducing the load on the database server. Today this decomposition is encoded by the application programmer.

[4]Tim Shetler provided this information.

8.4 Case Study: Shop Comparison Portal

Kelkoo is a shop comparison portal that provides access to online shops from eight European countries. Like most shop comparison portals, the Kelkoo Web site allows users to navigate in a hierarchy of product categories and to compare the prices for which a particular product can be obtained from various online shops as well as to explore various discounts and special offers. In the second quarter of 2001, the Kelkoo portal hosted 1.4 million visits (a visit corresponds to a user session with a duration of 30 minutes), which generated 1.8 million hits on the referenced online shops.

Kelkoo relies on Internet logs to monitor user activities. There are three purposes:

1. *How many hits are directed to which online shops?* This allows Kelkoo to negotiate with online shops.
2. *What are the most visited pages? Where do users come from?* Like every portal, Kelkoo buys ad banners displayed on the Internet and also hosts ad banners for other companies. This information is used to negotiate the price of ad banners and to improve the organization of the portal.
3. *What are users looking for?* This is used to determine discounts and special offers.

The Kelkoo portal follows a three-tiered architecture, where Web servers, application servers, and database servers are located on separate production hosts (there are several hosts for each tier).[5] Both the Web servers and the application servers generate logs. The Web server traces which pages are visited and which sites users come from, while the application servers trace user requests and hits generated for online shops. Each server generates a log file stored on the local host. There are about 100 megabytes of logs generated each day. They are compressed, then transferred from the production hosts to a separate data warehouse.

There are two performance objectives.

- The warehouse should be refreshed nightly between 2 A.M. (the time logs are closed for a given day) and 9 A.M. (the time at which the Kelkoo team starts querying the warehouse).
- Members of the Kelkoo team should have fast access to the visual representation of the log warehouse; otherwise, experience shows that data in the warehouse is not used.

The Internet log warehouse is composed of 26 tables. The main tables concern user visits, pages visited, hits on ads, and hits on online stores. Other tables are materialized views created for aggregate maintenance (see Chapter 4). Raw data is kept for the current month and the three previous months. Older raw data is archived. Aggregate data is kept in the warehouse. The log warehouse occupied about 6 gigabytes as of July 2001.

Statistical manipulations of the raw data are performed using scripts written in-house or using Webtrends; that is, statistical manipulations are performed at the application level outside the database system. The output of these programs is formatted so that it can be loaded into the database or displayed using a graph visualization tool.

Here are two examples of the use of aggregate maintenance for this Internet log warehouse.

- A graph representing the number of visits per month, per user, over four months (including the current month) is generated each day. Generating this graph is part of the activity that is performed at night and must be terminated within seven hours. Running a program that computed the graph from raw data (the number of visits) took about an hour. Aggregate maintenance reduced the time needed to produce the graph to a few seconds. A table storing the visits per month was created. Each month a row is inserted in this table using the raw data. Each night the graph is produced in about ten minutes using the raw data for the current month and the aggregated data for the previous months.
- The list of top online shops is generated on demand. In this list, shops are ranked by the current number of hits; the list also includes the history of hits for these shops over the last months and a provision for the next month. Generating this list took a minute using raw data, and it now takes about five seconds using a table of the top shops. This table is generated each night. The gain in performance is obtained at the cost of a loss in accuracy, which is okay in the context of this application.

It is more efficient to run a script and bulk insert the rows that are generated instead of inserting rows whenever they are generated. Performing a bulk insert has the obvious advantage of reducing the length of the insert transaction: 2 minutes instead of 30 minutes. Note finally that the whole infrastructure has been audited to ensure that the contents of the warehouse corresponds to the data obtained from the logs. For instance, it is important to ensure that the loading of the data warehouse does not cause hits to online shops to be counted twice or requests coming from other sites to be dropped. We discuss how to achieve such "exactly once" behavior in Chapter 6. The basic idea is to make the transfer of data part of a database transaction.

[5]Note that 75% of the requests generated by the application servers are directed to the database servers on which price information is loaded every night, whereas 25% of the traffic is directed to remote online auctioning or flight reservation servers.

8.5 Capacity Planning in a Nutshell

"I am of the opinion that the boldest measures are the safest."

—British Admiral Horatio Nelson, upon urging a direct assault on a fortified harbor and fleet in 1801. He prevailed.

Capacity planning is hard. The math is easy (a little probability and algebra as we will see). The measurement part is time consuming but straightforward (the many experiments we offer on our Web page should give you a head start). Getting the demand assumptions right are what makes it hard. It helps to be bold.

To give you one example, when the French national phone company introduced Minitel (a kind of proto-Web) in the 1980s, their engineers assumed a certain traffic. To encourage use, they offered added services such as e-mail sex lines. The concept was that users would engage in arousing conversations over the tiny monocolor Minitel screens with squarish fonts. Now who would find that appealing? But we're talking about France, the land of romance and poetry. The sex e-mail was so popular that the heavy traffic caused the communication hubs to crash!

Unfortunately, no book can tell you how to estimate demand when the product is entirely new, as e-mail was to most of France or Napster was to the Internet. When your product is less novel (say, a new kind of insurance), then market it heavily.[6] After that, organize focus groups, perform telephone surveys, and hire statisticians to do the relatively straightforward extrapolations.[7] This tends to work a lot better than asking the system architects to estimate demand.

8.5.1 Capacity Planning Essentials

For purposes of this section, we're going to assume that someone gave you the demand somehow. This might come in the form, "We expect to get *x* hits to our Web site per minute, with a peak load of *y*. Fraction *z* of those hits will perform searches, and so on." We express that using our first mathematical tool: probabilistic Markov models. Figure 8.4 shows one example in which the entry point has *service time* S_1 (the time to handle an entry). Of those users who enter, 0.4 go to the search node. Each search takes time S_2, repeats with probability 0.5, and otherwise enters checkout with probability 0.1.

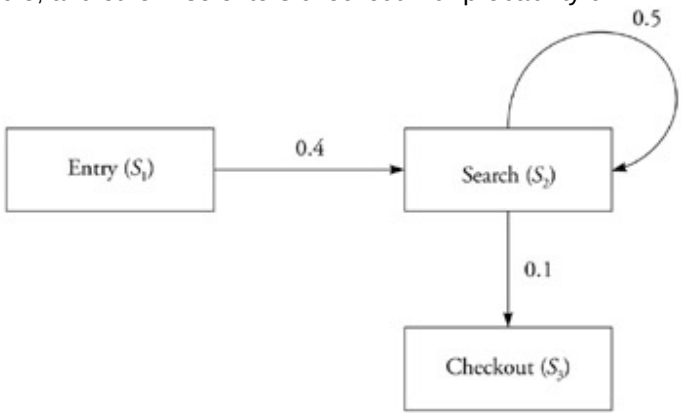


Figure 8.4: A probabilistic state transition diagram of a simple e-commerce application: 0.4 of the clicks on the entry proceed to S_2, 0.5 of the entries to S_2 visit S_2 again, and 0.1 enter S_3.

The *arrival rate* is the number of requests that arrive in a given time interval. The arrival rate into entry is given by marketing, and we will call it A_1. A_2 is the computed arrival rate (both from entry and from feedback) into S_2, and A_3 is the computed arrival rate into checkout. The fact that the fractions leaving each node total to less than 1 implies that there is *leakage*—customers who leave the site.

The equations for the two unknowns, A_2 and A_3, are simple.

$$A_2 = (0.4A_1) + (0.5A_2)$$

$$A_3 = 0.1A_2$$

Solving in terms of A_1, we have $A_2 = 0.8\ A_1$, and $A_3 = 0.08\ A_1$.

Now that we have the arrival rates, we must measure the service times, S_1, S_2, and S_3. This involves several steps.

1. Estimate the service time for a single server. Please be as realistic about size and disk resources as possible. (Feel free to modify the experiments from our Web page.)
2. Take parallelism into account. Now, the service time for a service depends partly on the number of servers *N*. If the servers are identical, the network bandwidth is high, and they share no physical resources, then you can obtain the service time at a resource

by dividing the time at a server by *N* (or, if you want to be conservative, a number slightly less than *N* such as 0.8 *N* to take into account failures and unforeseen interactions).

3. Decide on your server model. The single most important qualitative observation of queueing theory is that *your system's customers will enjoy a much more uniform response time if you have a single queue for multiple servers than if you have one queue for each server.*[8] Implementing a single queue requires some kind of middleware and high-bandwidth connections to that middleware, so sometimes this is not feasible. If infeasible, then the next best design is to assign jobs to servers randomly or perhaps based on the hash of a meaningless key. The reason we stress the desirability of single queues is that many systems must meet stringent response time constraints in order to satisfy the many customers who have the attention span of fruit flies.

So, now you have the service times S_1, S_2, and S_3 and the arrival rates A_1, A_2, and A_3. Suppose we want to determine the average response time at each node *X*. This will be a function of the arrival rate *A* and the service time *S* at *X*. Recall that *A* tells how many jobs arrive in a certain time period, say, 100,000 per second. *S* is the time to perform each request, say, 2 microseconds (the requests are simple). Therefore, $A \times S$ is the fraction of time the servers for *X* are busy. That is called the *average utilization* or *U*. (*U* is 0.2 in this case.) If this number is over 1, you need more servers.

Whereas a utilization near 1 could be achieved, if the traffic arrived in a perfectly regular manner, as it does, say, when you load data, external traffic is seldom regular. Even assuming you have designed for the peak arrival rates as expressed, say, as events per second, there are small variations in actual arrivals. These are best characterized by a *Poisson model.*[9] In that case, the utilization should be considerably less than 1 to handle variant traffic. The rule of thumb is 60% or less. The mathematics underlying that rule of thumb is the following equation from queueing theory giving the response time *R* (and assuming Poisson arrivals):

$$R = U/(A(1-U)) = (AS)/A(1-U) = S/(1-U)$$

Thus, if the utilization is 50%, the response time *R* is twice the service time *S*. As the utilization approaches 1, the response time gets worse very fast.

8.5.2 What to Purchase

If you have bought your equipment, then the preceding discussion provides you all the basic tools you need. If you haven't, then try to borrow time from a manufacturer and then do your measurements on those. If you can't do that, then you're reduced to guesswork based on analogous applications. Since your measurements will probably miss some important transactions, you may want to purchase some multiple of the capacity the measurements and your demand model suggest. Most capacity planners use a multiple of 2 at least. Be bold.

Bibliography

Rakesh Agrawal, Amit Somani, and Yirong Xu. *Storage and querying of e-commerce data.* In *Proceedings of the 27th VLDB Conference*, 2001. This paper discusses an architectural suggestion that builders of e-commerce systems would do well to study.

Chris Carrigan and Terry Dickey, editors. *Using Business Statistics: A Guide for Beginners*. Crisp Publications, San Francisco, 1995. Dickey and Carrigan outline the basics of marketing statistics.

Mark Cuban and Jon Spoelstra. *Marketing Outrageously*. Bard Press, Austin, Texas, 2001. The authors discuss the art of making a staid project look glamorous.

A. Datta, K. K. Dutta, H. Thomas, D. Vandermeer, K. Ramamrithan, and D. Fishman. *A comparative study of alternative middle tier caching solutions to support dynamic web content*

acceleration. In *Proceedings of the 27th VLDB Conference*, 2001. This paper shows the limited advantage of using the database cache for generating dynamic Web pages. Indeed the overhead of HTML and XML generation dominates response time. The authors propose to cache fragments of dynamic Web pages.

Leonard Kleinrock and Richard Gail. *Queuing Systems: Problems and Solutions*. John Wiley and Sons, New York, March 1996. Kleinrock and Gail offer a lucid explanation of queueing theory in practice.

Daniel A. Menasce and Virgilio A. F. Almeida. *Scaling for E-business: Technologies, Models, Performance, and Capacity Planning.* Prentice Hall, Upper Saddle River, N.J., 2000. This book explains capacity planning methodologies for e-commerce, including some worked examples.

Linda Mui and Patrick Killelea. *Web Performance Tuning: Speeding Up the Web*. O'Reilly Nutshell, Sebastopol, Calif., 1998. Most e-commerce texts are quite vague about performance, but this book gives some practical tips.

Oracle9i Application Server Database Cache. Available online at http://technet.oracle.com/products/ias/db_cache/db_cache_twppdf.

TimesTen Front-Tier. Available online at http://www.timesten.com.

Exercises

EXERCISE 1

Evaluate the relative benefits of caching for a site that

- Delivers current stock quotes.
- sells medieval illuminated manuscripts.

Discussion. Caching helps when the item cached is large and difficult to construct and changes rarely. Both properties hold for illuminated manuscripts. Neither holds for stock quotes.

EXERCISE 2

As of this writing, Akamai and its competitors sell a service whereby data is downloaded from an e-commerce site S to many servers widely dispersed on the net. When a user connects to site S, the data actually comes from an Akamai server in a transparent fashion. Evaluate the following assertion: if it's worth caching the data, then it's worth "Akamizing" it.
Discussion. Generally, this assertion holds. Akamizing exaggerates both the costs and benefits of caching.

- Sending the data to all Akamai caches takes longer than materializing data in site S's cache.
- Downloading the data from an Akamai cache takes less time than downloading it from s's cache for most user devices.

So, data that changes every few minutes might not be worth Akamizing, which is also true of data that is accessed by hundreds rather than millions of users.

EXERCISE 3

You are designing an e-commerce site having an average peak demand of 8000 initial requests per second, each of which requires 10 milliseconds of service time on a single initial

request server. Of those 8000 requests, 20% turn into secondary requests requiring 100 milliseconds of service time on a single secondary request server. Each request requires a response time of 500 milliseconds. You may assume that the application is nearly perfectly parallelizable across servers. How many servers of each type will you need?
Action.

$$A_1 = 8000/\text{sec}$$

$$A_2 = 0.2A_1 = 1600/\text{sec}$$

$$S_1 = 0.01 \text{ sec}$$

$$S_2 = 0.1 \text{ sec}$$

The utilization is ($A_1 \times S_1$) = 80 for the first service and ($A_2 \times S_2$) = 160 for the second. These are both over 1. So, we must reduce the utilization through parallelism.
Recall that, under the Poisson model, the response time R per request is given by the equation

$$R = S/(1 - U) = S/(1 - AS) \quad \text{or} \quad S = R(1 - AS) = R - RAS$$

or

$$S(1 + RA) = R \quad \text{or} \quad S = R/(1 + RA)$$

For the initial request server, this implies S = (0.5)/(1 + (8000 × 0.5)) = 1/8002 seconds. Since the service time on a single server is 0.01 second, perfect parallelism would say that we would need 0.01 / (1/8002) = 80.02 servers. But assuming that near perfect parallelism means that N servers will reduce service time by only 0.8 N, we have to multiply 80.2 by 1/0.8, yielding about 100 servers.
Repeating this calculation for the secondary request server, S = (0.5)/(1 + (1600× 0.5)) = 1/1602 seconds. Since the service time on a single server is 0.1 second, perfect parallelism would say that we would need 0.1/(1/1602) = 160.2 servers. Using the 0.8 discount gives us about 200 servers. It's good that hardware is cheap.

EXERCISE 4

Calculate the net arrival rates if 50% of the secondary request jobs return to the first server and 30% return to a new request at the secondary request server.
Action.

$$A_1 = 8000 + 0.5A_2$$

$$A_2 = 0.2A_1 + 0.3A_2$$

The algebra is particularly simple here: simply substitute the expression for A_1 into the equation for A_2:

$$A_2 = 0.2(8000 + 0.5A_2) + 0.3$$

$$A_2 = 1600 + 0.4A_2$$

So, 0.6 A_2 = 1600 and A_2 = 1600/0.6. This raises the arrival rate at 1 to

$$A_1 = 8000 + 800/0.6 = 9333.33$$

$$A_2 = 2666.67$$

MINIPROJECT
There are a group of e-commerce-related performance experiments for you to do at the book's Web site at http://www.mkp.com/dbtune/. There you will find both code snippets to modify and example runs. Some of these experiments require external components. The projects concern the following issues:

1. Where is the time spent when generating dynamic Web pages? What is the time spent fetching data from the database server, and what is the time spent generating HTML or XML code?

2. Is it worth using the database cache for the data used in dynamic Web pages? Is the behavior similar for text and images? Is it worth using a clustered index for shopping cart data?
3. Given your server configuration, how many processes run on the server once you have reached the maximum number of connections?
4. Compare an external search engine combined with a database system to the internal full text indexing capabilities of the database server. What is the space utilization? What is the response time?

MINIPROJECT
A miniproject on the book's Web site at http://www.mkp.com/dbtune/ consists in comparing estimated response time with actual response time. A client application (simulating the application server) issues queries to the database server. The goal of the mini-project is first to predict utilization and response time for a given workload (i.e., a given configuration of the client application). To do so, it is necessary to obtain the service time parameter from the server. The second step consists in measuring the actual response time.

[6]See, for example, Jon Spoelstra and Mark Cuban. *Marketing Outrageously*, Bard Press, Austin, Texas, July 2001.

[7]Terry Dickey, Chris Carrigan (editors). *Using Business Statistics: A Guide for Beginners (A Fifty-Minute Series Book)*, Crisp Publications, San Francisco, May 1995.

[8]Banks have figured this out. Supermarkets have not—don't you love those shoppers who bring in newspaper coupons?

[9]In the Poisson model, the time between arrival *i* and arrival *i* + 1 is independent of the time between any other successive arrivals and is characterized by a function of time and the arrival rate.

Chapter 9: Celko on Data Warehouses—Techniques, Successes, and Mistakes

—Joe Celko, *Columnist, Author, Consultant*

9.1 Early History

Data warehousing began in the medical community with epidemiology studies. Historical data was kept in standardized formats that allowed researchers to look for patterns in the data and take appropriate actions. Cost was not much of a problem since the alternative to epidemiology is epidemics. People have always understood "your money or your life" as a sales pitch.

Over the years, the cost of storage and computing power decreased drastically. Before cheap storage, any research done by commercial data processing users either was a special project that collected its own data or involved digging thru archival storage. Archival storage meant "magnetic tapes" and a lot of manual labor to extract data from them. Once you got your data out of archival storage, it would not be uniformly formatted. Changes in the way that a company did business, new packages, and patches to old applications all made historical research difficult.

Another factor was that archival storage was destroyed. In the American legal system, old data can be used in court against you, but you are not obligated to retain records past a certain lifetime. One of the jobs of a records manager was to destroy out-of-date records. For the most part, this meant paper shredding, but it also meant erasing magnetic tapes.

The technology has, of course, changed. Commercial users can keep large amounts of data in active storage. Suddenly, you were actually using the "high-end" metric prefixes like tera- and peta- in real conversations.

9.2 Forget What the Elders Taught You

The relational database management systems are well established, and SQL is the de jure and de facto language. But the mental model of a database was still the transactional systems. Everything you know about transactional databases is the opposite in a data warehouse. Get ready to unlearn.

A transactional database holds current data. A data warehouse holds historical data. A transactional database should be as small as possible because a small database is usually faster. It can handle more transactions per minute.

A data warehouse should be as large as possible. The more historical data and the more detailed that data is, the more complex and long range are the patterns that can be discovered.

A transactional database is designed for many users who engage in short sessions. The session is almost always a fixed application. The classic example is an ATM machine.

A data warehouse is designed for a few trained users who engage in long, complicated sessions. These sessions are ad hoc queries or statistical analyses.

The transactional database session is very specific—move money from my saving account to my checking account. The data warehouse session might produce a chart of the movement of money from all saving accounts to checking accounts by time of day.
A good transactional database is normalized. It has to be normalized because the data is changing (INSERT, UPDATE, and DELETE statements in SQL), and an unnormalized database will produce data anomalies (see Chapter 4).

A data warehouse should not be normalized because the data never changes while it is in use. Yes, new data comes into the warehouse, but it is bulk loaded as a separate operation, not while the database is being queried.

This is where the star and snowflake schema designs come into play. They feature a large, denormalized "fact table," which represents a long statement about the enterprise. As a simplified example, the fact table might model all the information we have on all the cash register slips at all of our stores.

In one row, we would find the items bought, the price, coupons used, method of payment, and so on. This makes asking questions like "Do people who use credit card X buy more per shopping trip than people who use credit card Y?" because it is available in one read. The rows are also ordered chronologically, so temporal queries are very easy for query systems that can take advantage of physical ordering to do cumulative aggregates.

The other tables are dimensions—the units that each of the attributes in the fact tables uses. For example, the credit card column in the fact table might reference a table of credit cards we accept. This table would have the customers' names, when we added them, what our limitations were at that time, and other information about the cards.

Some dimensions are hierarchical in nature. The most common example is the ZIP+4 code, which is a geographical hierarchy. Using the ZIP+4 code, you could query the data warehouse at a regional level, then "zoom in" to state level, to city level, and finally to postal route and street level.

It is also possible to have more than one hierarchy on the same data. My favorite example was a shoe company that viewed their product with a manufacturing classification scheme that was constant. They also had a marketing classification scheme that changed after every market research study. In the manufacturing classification scheme, steel-toed work boots were one item. In the marketing classification scheme, small steel-toed work boots were a totally different product from large steel-toed work boots. Why? Construction workers bought the larger sizes, and teenage girls bought the smaller sizes—two different sales channels.Classification brings up the topic of how to represent hierarchies. The answer is surprising.

REPRESENTING HIERARCHIES

The usual example of a tree structure in SQL books is called an adjacency list model, and it looks like this:

```
CREATE TABLE Personnel

(emp CHAR(10) NOT NULL PRIMARY KEY,

boss CHAR(10) DEFAULT NULL REFERENCES Personnel(emp),

salary DECIMAL(6,2) NOT NULL DEFAULT 100.00);



Personnel

emp        boss       salary

---------------------------------
```

```
'Albert'    'NULL'      1000.00

'Bert'      'Albert'    900.00

'Chuck'     'Albert'    900.00

'Donna'     'Chuck'     800.00

'Eddie'     'Chuck'     700.00

'Fred'      'Chuck'     600.00
```

Another way of representing trees is to show them as nested sets. Since SQL is a set-oriented language, this is a better model than the usual adjacency list approach you see in most textbooks. Let us define a simple personnel table like this, ignoring the left (lft) and right (rgt) columns for now.

```
CREATE TABLE Personnel

(emp CHAR(10) NOT NULL PRIMARY KEY,

lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),

rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 1),

CONSTRAINT order_okay CHECK (lft < rgt) );
```

```
Personnel

emp         lft       rgt

---------------------------

'Albert'    1         12

'Bert'      2         3

'Chuck'     4         11

'Donna'     5         6

'Eddie'     7         8

'Fred'      9         10
```

Figure 9.1 represents the organizational chart as a directed graph.

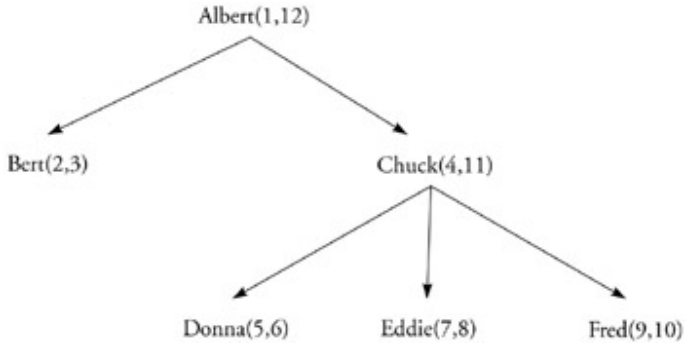


Figure 9.1: Organizational chart as a directed graph.

A problem with the adjacency list model is that the boss and employee columns are the same kind of thing (i.e., names of personnel), and therefore should be shown in only one column in a normalized table. To prove that this embodies redundancy, assume that Chuck changes his name to Charles; you have to change his name in both columns and several places. The defining characteristic of a non-redundant table is that you have one fact, one place, one time.

Further, the adjacency list model does not model subordination. Authority flows downhill in a hierarchy, but if I fire Chuck, I disconnect all his subordinates from Albert. There are situations (i.e., water pipes) where this is true, but that is not the expected situation in this case.

That brings us to lft and rgt. Do a traversal on the outside of the tree. Associate with each leaf a number and its successor, and associate with each interior node the number as you traverse downward and the number as you traverse upward.

If that mental model does not work, then imagine a little worm crawling counterclockwise along the tree. Every time he gets to the left or right side of a node, he numbers it. The worm stops when he gets all the way around the tree and back to the top.

This has some predictable results that we can use for building queries. The root is always (left = 1, right = 2 * (SELECT COUNT(*) FROM TreeTable)); leaf nodes always have (left + 1 = right); subtrees are defined by the BETWEEN predicate; and so forth. Here are two common queries that can be used to build others.

1. An employee and all of their supervisors, no matter how deep the tree.

```
2.  SELECT P2.*
3.    FROM Personnel AS P1, Personnel AS P2
4.   WHERE P1.lft BETWEEN P2.lft AND P2.rgt
5.     AND P1.emp = :myemployee;
```

6. The employee and all subordinates. There is a nice symmetry here.

```
7.  SELECT P2.*
8.    FROM Personnel AS P1, Personnel AS P2
9.   WHERE P1.lft BETWEEN P2.lft AND P2.rgt
10.    AND P2.emp = :myemployee;
```

Add a GROUP BY and aggregate functions to these basic queries and you have hierarchical reports; for example, the total salaries that each employee controls:

```
SELECT P2.emp, SUM(S1.salary)

  FROM Personnel AS P1, Personnel AS P2,
```

```
        Salaries AS S1
    WHERE P1.lft BETWEEN P2.lft AND P2.rgt
      AND P1.emp = S1.emp
    GROUP BY P2.emp;
```

To find the level of each node so you can print the tree as an indented listing, use the following:

```
    DECLARE Out_Tree CURSOR FOR
    SELECT P1.lft, COUNT(P2.emp) AS indentation, P1.emp
      FROM Personnel AS P1, Personnel AS P2
     WHERE P1.lft BETWEEN P2.lft AND P2.rgt
     GROUP BY P1.emp
     ORDER BY P1.lft;
```

This approach will be two to three orders of magnitude faster than the adjacency list model for subtree and aggregate operations.
For details, see the chapter in my book *Joe Celko's SQL for Smarties* (Morgan Kaufmann, 1999, second edition).

Time is important at this level since dimensions can change over time. For example, when did you start selling soft drinks in liters instead of fluid ounces? How do you compute dollar values that are adjusted for inflation?

The snowflake schema is like a star schema, but the dimension tables have associated smaller tables that explain their attributes. Think of a fractal design or a snowflake. The goal is to help with changes in dimensions.

Imagine a central fact table that holds sales data. Some of the dimensions of a sale are the product, the price, and the store where the sale was made. The store dimension is linked by an identifier to a store table. But a store has geography and demographic subdimensions associated with it. If store number 5 has a change in demographics, say, a swinging singles condo opens next to the old folks home in the neighborhood, the store dimension table is "relinked" to a new demographic code and some temporal data about when the switch in the neighborhood occurred.

Going back to the differences between transactional databases and data warehouses, we want to use the minimal number of tables in a transactional query, and we can be pretty clever about the code.

Writing an SQL query for a snowflake or star schema is not difficult, but it is bulky because the dimensions are joined to the fact table in the majority of the queries. But you have a large number of tables in those joins compared to a transactional system. The solution has been to build frontend tools with a graphic interface that constructs the SQL under the covers and perhaps brings the answers back in a graphic display.

One implication of all this is that it is a bad idea to put a data warehouse on the same hardware as an OLTP system. The transactions will be unpredictable when all the disk space is eaten by the warehouse, and the CPU time suddenly goes off to perform some massive statistical calculations. Want to tell people that the payroll will be a day late because someone in marketing decided to see if there is a relationship between 25 independent variables?

9.3 Building a Warehouse Is Hard

Data warehouses were originally done at the corporate level and took a long time to build. A major reason for the high cost and time was that they went across departmental lines and had to pull data from many different systems. These systems never worked together before.

Building a uniform data model at that level is hard. Each data element has to be sweated over and rigorously defined. One of the classic examples was a project at a major airline that discovered they had just over 60 different definitions of a "passenger" in their application systems. Is an infant who does not have a ticket a passenger? Is an animal in the cargo hold with a ticket a passenger? Are you the same passenger on each leg of your flight? Is an empty seat a passenger when it was paid for on a nonrefundable ticket? Are overbooked passengers accounted for in some logical fashion? Standby passengers? Do airline employees on a free ticket count?

Because the data warehouse was so expensive, we then came up with the less ambitious "data mart" concept. These databases would be something like a small data warehouse, but they would focus on one topic, which might be at a departmental level or might go across departments. For example, I might wish to look at payment patterns over time and I would need only the accounting department records. But if I wanted to look at customer relations, I would track orders, payments, returns, service requests, and other things that are not in one department.
The misconception was that by building a lot of data marts, you could merge them into a data warehouse. Remember how hard it was to get a uniform data model across multiple application databases? *It is just as hard or harder to get a uniform data model across multiple data marts.*

The data warehouse queries deal with aggregates rather than with single transactions. Computing the same aggregations over and over is a waste of time. But storage is cheap, so it is possible to store the aggregations in the data warehouse. You would never do this in a transactional database because you would then have to maintain the redundant aggregate information.

This leads to analytical processing and a host of unpronounceable abbreviations. The first one, OLAP, On-Line Analytical Processing, was defined by E. F. Codd, the inventor of the relational model. If you use a relational database, it is called ROLAP; however, a relational database is not good at storing a hierarchy of aggregations. This lead to Multiple Dimensional Databases (MDDBs), which have storage structures for doing aggregations.

Hybrid OLAP (HOLAP) allows you to construct aggregations in a relational system and store them in an MDDB. This attempts to give you the best of both worlds, but you will pay for it in storage.

9.4 The Effect on the Bottom Line

Now, the bottom line: does it work? There are a lot of failed data warehouse projects. I know that people will argue over why they failed, but my guess is that most of the failed projects were never really data warehouses. A data warehouse is not a large pile of historical data in one black box, it is not a collection of data marts, and it is not a magic lamp that grants wishes. I think that companies started such projects as corporate fads and did not want to take the time to unify the schemas, could not overcome the departmental desire to hoard data, or did not learn enough statistics to do the data mining.

Are there successes? Oh yes! The first good consequence is that setting up the data warehouse made companies do data audits and clean up their production databases. The amount of dirty data in the production systems was always an unpleasant surprise.

The second effect of a data warehouse project is that people look at the entire enterprise as an integrated process, instead of separate parts only vaguely related to each other.

The third effect of a data warehouse project is that you get useful information out of the warehouse. One drugstore chain had been removing regular candy from the shelves to make room for Valentine's Day candy in most of their stores. Regular candy has a high markup while Valentine's Day candy has a low markup. The logic is simple: the candy companies know you have to have Valentine's Day candy, but one candy bar is much like any of the hundreds of other brands and thus has to compete for display shelf space.

The question was, how does leaving the regular candy on the shelves during the Valentine's Day season affect sales? Based on their data, they saw a halo effect— Valentine's Day candy increased all candy sales. The data warehouse project paid for itself over the Valentine's Day season.

Some information that comes out is just weird and might be just random correlations. Did you know that people who wear tweed eat twice as much chocolate as the average person; and that having a sale on beer at a Quicky Mart will increase the sale of disposable diapers?

People can invent explanations after the fact. The traditional rationale for the beer and diapers was that working-class fathers on their way home would see the beer sale, buy a case, see the diapers and remember to bring them home so they would not catch hell about the beer. A less traditional explanation was that working-class mothers would go to the Quicky Mart to get diapers and see the beer sale. She's been home with the kids all day, so she needs the beer even more than the father does. But does the cause matter if you have the fact and can turn it into a course of action?

9.4.1 Wal-Mart

The classic case study for a successful data warehouse is Wal-Mart. It is so much so that there is even a book about their techniques.[1] The input of every scanner in every Wal-Mart is fed into Bentonville, Arkansas, to the Retail Link, a proprietary system. Wal-Mart averages about 100 million customers a week.

My personal view of Wal-Mart was formed when my wife and I took a cruise to Alaska. I do not like the standard stops that cruises plan for you—they never include a brew pub or a strip joint. Before Wal-Mart arrived in one small town in Alaska, the price of milk was just over $6.00 a carton at their local mom-and-pop stores. Wal-Mart sold milk for just over $3.00 a carton, and gave their employees stock plans and benefits they had never had before.

They have been able to come into small towns and different countries to offer local customers huge savings in large part due to their volume buying, but also because of their reduced overhead. That reduced overhead is due in part to their data warehouse.

Their data warehouse is measured in terabytes and might be in petabytes by the time you are reading this chapter. The mental image that IT people form when hearing about such a system for such a large company is that it is great for expected cycles, but not for unexpected changes. But that is not the case for Wal-Mart.

Let me make that clearer with a deliberately silly example. Imagine you are the buyer for women's swimwear. Given long-range weather forecasting, a purchasing history by each store,

and other demographics, you can predict with some certainty how many and what kinds of swimwear to put in each store.

In Florida, you ship large-sized, one-piece maillots with Godzilla tummy control panels for retirement villages. The only thongs they wear are on their feet. The string bikinis with hip, cool name-brand labels head for Fort Lauderdale just before spring break. You adjust your shipping dates by the expected weather forecast for warm weather and you are done.

But let's imagine that the weather suddenly changes and we get a prolonged snow-storm instead of a hurricane in Florida this year. What happens at the individual stores; what does the individual employee do? The immediate thought is that a huge system cannot adjust to sudden radical change. It is like a dinosaur that is so large it needs two brains; by the time the front brain tells the back brain that we are heading for the La Brea tar pits, the tail is swishing the asphalt. The bikinis will arrive, and parkas will not, according to schedule and last year's planning. The individual manager will try to adjust, but the system as a whole will keeping heading for the tar pits.

In a traditional IT environment, the disjoint systems would have been like the dinosaur's brains. By contrast, the centralized warehouse becomes the "mammal brain" for the enterprise.

My bikinis-in-a-snowstorm scenario was meant to be funny. Some scenarios are less so. On September 11, 2001, the World Trade Center in New York City was destroyed, and the Pentagon was damaged. I see no humor in these things, but we had to get back to normalcy and restore commerce. Let's look at what happened to Wal-Mart.

Nationwide, Wal-Mart store sales dropped 10%, but the stores in northern Virginia and those around New York dropped as much as 40% from the year earlier. When people did start coming back to shop, the purchases changed.

On the Tuesday of the attack, people began buying American flags (116,000 versus 6400 the year before), canned food, bottled water, other emergency supplies, guns, and ammo—but in the Midwest, not on the East Coast where the attacks occurred.

Wednesday saw what Wal-Mart called "the CNN effect"—early morning shopping for candles, more flags (200,500 of them), and T-shirts.

By Thursday, shopping patterns were starting to return to normal, but at a reduced individual purchase level. The expectation was the weekend after the attack would be a tidal wave of pent-up buying needs.

A Nebraska store manager named MacNeil made a decision based on this data and predictions. He increased his weekend staff by 20.

Moral to the story: a good warehouse can respond faster to catastrophe than a traditional set of disjoint systems. It does so by giving global feedback to the individual units.

9.4.2 Supervalu

Perhaps the most enlightening case study is the one that deals with a company that starts from scratch and hits all the walls that you can hit while trying to get started. A person should learn from experience—but not his or her own. Let some other guy get the arrows in his back.

Supervalu is one of the top ten grocery chains in the United States with $23.5 billion in sales (Price-Waterhouse) and in the top ten in retail sales at $9 billion. Their retail outlets include Cubb Foods, Shop 'n Save, Metro, Bigg's, Save-a-lot, Farm Fresh, Scot's, and Hornbacher's for a total of 11,658 stores, serving 9 million customers a week.

Rick Collison, director of retail merchandising, and his team began a data ware-house project with a set of clear goals. They were to start small but still model the entire enterprise and support the category managers with at least one year of history.

They started with a few reasonable hypotheses about their data.

1. Captured sales data is accurate.
2. We sell stuff we purchased.
3. The item hierarchy is stable.

Of course, this was all wrong. Let's start with premise number one.

The stores had different POS (point of sale) systems (IBM, ACR, ICL S-18, and some NCR ACS units, if you wanted the details). Each of the systems works a little differently. The machines summarize data differently. Some of the systems would use the number of units per week and the price at the end of the week for their summary report. Let's say we are selling Coca-Cola for $1.00 a six-pack on a special on Wednesday and Thursday only. The normal price is $3.00, and the price returns to that on Friday when the POS report is issued. Some POS units returned the units sold but not the sales.

Then the stores would report duplicate files two weeks in a row, report truncated files, or not report at all.

There is also the subtle problem of the grocery items like meat and bulk items that do not come in neat packages. The systems that price the meat when it arrives at the store, the scales that weigh it when it is cut, and the POS unit that puts it on the check-out slip are all different.

Then the accounting department and the category managers had a different hierarchy of categories, so they did not really talk to each other in the same language.

Premise two, we sell what we buy, was also wrong. Many of the stores sold things they did not buy through the company. The franchise stores are not obligated to sell only what they get from the company.

Things bought by weight are sold by "the each" and vice versa. The warehouse works on the item codes rather than the UPC codes. Mixed pallets and mixed shippers for the same item mean different prices for the same UPC.

Another cute "gotcha" is that the UPC code on the item is not the same as the UPC code in the vendor's book. Which code did the store managers report to you? Both, of course.

If you are not familiar with UPC bar codes, you might wonder how the two could be different. The UPC codes really identify the vendor (the first five digits you see in the bar code) and package (the last five digits), not the product. Thus, a brand of toothpaste that comes in five sizes will have a UPC code for each of those sizes. But the vendor is relatively free to reuse a code when a product is discontinued or to reassign the codes to other products.

Add bad UPC codes and other data quality problems, and you can get an idea of what life in the real world is like.

Premise three, a stable product hierarchy, was endangered by another team that was using the same data to develop a new gross profit accounting system. The solution was to combine the project so that they did not get "two versions of the truth" in the company.

So much for the premises that we started with. The next kicker was that they decided to use Teradata for the warehouse and they had no experience with that database. Oh, did I mention that the data warehouse was to be ready for prime time in 90 days? It wasn't.

Data scrubbing was a major problem. The team soon recognized that they were not scrubbing data—they were really starting to make up data to get it to fit their model. But when it was all over, the data "smelled bad" to an experienced user.

The real solution was not to scrub the data, but to make sure that the equipment at the stores were all reporting by the same units of measure and playing by the same rules.

Once you have your data, you can do simple reasonableness checks with simple cross-tabulations, looking at minimum and maximum values (ask yourself, does anyone really buy 5000 bananas?).

The moral to the story, according to Mr. Collison, is "Whatever work you think this will be, it will be more."

9.4.3 Harrah's

Harrah's operates 21 casinos in the United States, is listed on the New York Stock Exchange, and has served about 18 million people. Their concern was customer relations management (CRM), and their method was a customer rewards program under the name Total Rewards. This is not a radical new idea; everyone from supermarkets to airline frequent flyer programs has a plastic card for their regulars.

The Harrah player puts his Rewards card into a slot machine, and the system looks for his data locally. If he is not a local at this casino, the system goes to the central data warehouse and downloads his profile. I think of this as a reverse ATM since you are going to put money in and not take it out.

Before the warehouse project, each casino kept its own database and did not communicate with the other properties in the company. The local OLTP databases still hold more customer information than the central warehouse.

The goal of this warehouse is the analysis of the customers and their buying patterns as related to Harrah's offerings. It is not a general enterprise-level warehouse, and that is a good thing. This lets the IT people develop their own customer profiles and focus on one job.

When you check into a property, the system knows who and where you are and can make a customized offer to you. If the player is from out of town, discounts at a local hotel would be important. If the player is local, then perhaps theater tickets would be a better offer. If the show is not selling out, the ticket offer might be made to all the customers.

The marketing can go down to individuals. Look at the history of a family and see that they visit Las Vegas about the same time every year. Harrah's can mail them a list of events and things to do before they make their annual visit. The rewards can be the deciding factor in their vacation plans.

Does it work? Harrah's has gotten awards for their CRM system, but that is the trade honoring its own. The best measures of success are on the bottom line of the company. New customer visits increased by as much as 15% per month, and repeat visits increased from 1.2 to 1.9 per month. The Tunica, Mississippi, casino doubled the profitability per customer. The overall profit improvement was $50 million in 1999 from a 72% growth in business.

There is a moral here: a data warehouse does not have to be huge to produce results. It has to be smart.

[1]Paul Westerman. it *Data Warehousing: Using the Wal-Mart Model:* Morgan Kaufmann, 2001.

Chapter 10: Data Warehouse Tuning

10.1 What's Different About Data Warehouses

Data warehouses differ from transactional databases in three main ways.

- They are bigger—terabytes instead of megabytes or gigabytes.
- They change less often, often daily or at most hourly. If online changes are allowed, they are normally appends.
- Queries use aggregation or complex WHERE clauses.

The implications of these three points are surprising.

- Scanning all the data is too slow. Redundant information is needed in the form of special indexes (such as bitmaps or R-trees) or in the form of structures that hold aggregate information. In Chapter 4, we discussed holding aggregate information about total store sales in one table and total sales per vendor in another table. Data warehouses raise such tricks to high art—or try to.
- There is time to build data structures because the data changes slowly or large parts of the data (e.g., all old data) change slowly.
- Queries that perform aggregates benefit from structures that hold aggregate information. Queries having complex WHERE clauses benefit from query processing engines that can exploit multiple indexes for a single table.
- Because of the large variety of data warehouse applications (aggregate rich, complex WHERE clause, massive joins), many technologies have survived and have found niches.

In this chapter, we discuss the use and tuning of a wide variety of indexing techniques (e.g., bitmaps and star schemas), table caching techniques (e.g., tables and matrices that store aggregate data), sampling (e.g., extrapolate from portions of tables), and query processing methods (e.g., optimized foreign key joins).
In the spirit of Celko (but who could hope to imitate his style?), we start by illustrating several applications of data warehouses. In many of these applications, aggregates (e.g., find hot-selling items) suggest decisions but the effect (e.g., organize a promotion) is directed at individual data items or perhaps individual customers. That is, aggregates flow up from a wide selection of data, and then targeted decisions flow down. We call this process *aggregate targeting*. Please notice how often that applies in the real world.

10.1.1 Uses of Data Warehouses

At an abstract level, the principal goals of data warehouses are to increase the efficiency of the supply chain and to improve customer relationship management (CRM). What do these buzzwords mean in a concrete sense?[1]

1. Companies can negotiate better buying terms from their suppliers and enforce high-quality deliveries.
2. Large-volume suppliers can get information on the sales of their products so they can ship the product and avoid out-of-stock items. Some retailers take advantage of this by paying their suppliers on consignment (i.e., only when the retailer is paid).
3. Companies can identify and reward loyal customers by giving them frequent flier miles, for example, or by giving special discounts.
4. Data warehouses form the nucleus for data mining techniques using statistics, neural nets, decision trees, or association rules. Data mining can help discover fraud, identify good candidates for mortgage loans, or identify prospects for advanced telephone services. Market basket (what people buy at one time) analysis can help suggest promotions. In a notorious example cited by Celko, Wal-Mart, for example, discovered that

people who buy Pampers often buy beer, so they moved Pampers and beer close together. The result was that sales of both increased (*Computer Business Review*, October 1996).

Wal-Mart is the world's most successful retailer. Some indications: a 200,000-square-foot Wal-Mart opens every few days; a Barbie doll is sold at a Wal-Mart every two seconds. The company credits much of its success to the control of its data. Many of Wal-Mart's large-volume suppliers, such as Procter & Gamble, have direct access to the Wal-Mart data warehouse, so they deliver goods to specific stores as needed. Wal-Mart pays such companies for their product only when it is sold. Procter & Gamble ships most of its items this way, eliminating paperwork and sales calls on both sides, a clear benefit of aggregate targeting. The Wal-Mart data warehouse is also the source for corporatewide decisions. The company can effect price changes globally and instantaneously at its more than 3000 locations.

Data warehouses enable many companies to identify hot-selling items and avoid out-of-stock conditions (aggregate targeting again).

1. An analyst discovers that bags of multicolored confetti sell out in certain cities of the southwest United States. A store manager in Texas informs him that the confetti was being used in Latino communities to fill Easter eggs. The buyer authorized stores in other parts of the country to ship their confetti to these stores.
2. A food store discovers that a best-selling item in college towns is albacore tuna.
3. Greeting card analysts can ask questions like "Do people buy valentines at the last minute?" "Do they buy photo albums more on Father's Day or for graduations?"
4. Casino Supermarche, a superstore in France, recouped several million dollars when its analysts noticed that Coca-Cola was often out of stock in many of their stores.

Warehouses help retailers practice trend merchandising by "riding the waves of clothing fads" that begin on either U.S. coast and work their way inland. There is also weather-induced clothes-buying fashion (e.g., spring clothes sell earlier in Houston than in Minneapolis).

Warehouses can identify the effectiveness of advertisements. At Buttrey Food and Drug, a report can be generated that includes units sold, average cost, retail price, gross margin percentage, and dollars realized during an advertisement campaign. Buttrey also uses their warehouse to index competitors' prices against their own—competitive analysis meets data warehousing.

Warehouses can support business-critical ad hoc queries. A clothing store started a line for petite women. Many catalogs were requested, but few purchases. It turns out that many women wanted to be petite so requested that catalog but were no longer petite. The store did a directed ad campaign to women who had bought small sizes (dresses under size 7, shoes under size 5, etc.). They got a 12% response rate, a fantastically high rate.

Data warehouses can also be used to keep customers happy by supporting strategic preventive maintenance. For example, in the mid-1990s, Whirlpool engineers detected a faulty hose clamp on hundreds of washing machines that had already been sold. The affected customers were identified through the data warehouse. Mechanics replaced the clamps before flooding could occur.

Warehouses can improve productivity in a variety of industries. A railway company uses their warehouse to analyze the work habits of all crew members and thus reduce absenteeism. They also do data mining on their warehouse to try to identify the underlying causes of derailments. An airplane manufacturer uses its data warehouse to track the quality records of its suppliers.

An airline uses its warehouse for customer information. Which special routes were frequent flier customers flying? Which agencies fail to promote that airline? How many no-shows were there on a particular route? Flights with many no-shows might be good candidates for overbooking.

The data warehouse is a good place to put bought data. Continental buys rental car customer data. It tries to attract people who live on one coast yet rent cars on the other.

An airline tracks delays that affect its frequent customers. If it determines that a frequent customer was delayed several times in the space of a few weeks, it sends a letter explaining how rare such delays are along with a free ticket. An angered customer turns into a loyal one (aggregate targeting, again).

In telecommunications, a company may use its data warehouse to identify operational problems. Any item can be drilled down. For example, given installation times by region, a manager can drill down to the installation times by switching area. If one switching area is bad, he or she can drill down by problem type. If one problem type is bad, the reason may be that a switch doesn't support a sold service.

Data warehouses can support new marketing campaigns. Sprint once ran a promotion called "The Most" in which Sprint halved the bill for whomever a customer called the most.

Coming back to the frequent theme of aggregate targeting, please notice that the performance-critical step normally has to do with computing the aggregates. Finding the customers (or other targets) to apply the decisions to is often just a selection. Constructing the rule involves either human judgment or data mining.

10.1.2 Technology for Data Warehousing

Whereas online transaction systems (OLTP systems) are optimized for finding a record associated with a specific key (e.g., finding the information about employee 123124), data warehousing applications require finding information about sets of records very quickly.
As we've seen, some of these queries are aggregates in nature, for example, find all sales by region and quarter, or find stores that sell the greatest volume of sportswear per month, or select the top five stores for each product category for the last year. Query conditions are often over ranges of values (e.g., a range of sales) or on categorical attributes (e.g., sportswear).
The challenge is to process these queries without doing a linear scan of all or most of the database. Because these queries aggregate data, (1) data structures or materialized views that store the aggregates are a promising approach; (2) compression techniques and sampling are possible because slight inaccuracies matter little. We call applications requiring this class of queries *broad*.
Other queries, however, require precise individualized information, for example, which women actually bought petite clothes or which frequent fliers suffered several delays in a short time. We call applications requiring this class of queries *deep*.

Aggregate targeting applications are always broad and often deep.
An additional issue is whether the application requires information to be up to date and if so whether the data is updated frequently. If both are required, then the data structures to be used must be incrementally updatable. Otherwise, they can be reconstructed from time to time. In the first case, we say the application is *dynamic*. In the second case, we say the application is *static*.

Five main approaches have been used to address this problem: multidimensional arrays, special indexes, table caching, main memory database techniques, and approximation by sampling.

Multidimensional arrays (matrices)

Ideal application parameters: broad, static, moderately sized database

In this approach, each array dimension corresponds to an attribute of the data. For example, the line order table can be viewed as a matrix having coordinates store location, product type, customer id, and so on. A particular line order can be identified by specifying all values for these attributes, and queries are expressed by specifying values for a subset of the attributes. Corresponding to many of these subsets, the data structure stores aggregate values. This is the basic strategy used by Hyperion's Essbase[2] (distributed with IBM DB2 OLAP) and Oracle Express. It works best for relatively small databases (under one gigabyte as of this writing) or very dense ones. By dense, we mean that nearly all members of the cartesian product of possible values should be meaningful (i.e., every customer is likely to buy every product from every store). Because density doesn't usually hold for all combinations of values, this scheme must be modified to deal with sparse matrices. Hyperion does this by defining a notion of sparse attributes and dense ones. For example, it might be that every store carries every product (a dense relationship that can be stored in a matrix), but only some of these combinations are valid for any given customer. In that case, a conventional index would be used for customer sales, but a dense one for storewide or productwide sales. The dense attributes of such structures are optimized for range queries.

Using our categorization at the beginning of this section, this approach works best for broad, static applications. The precomputation can take a long time as well as multiply the database size sevenfold or more.

Bitmaps

Ideal application parameters: deep and broad, static, many query attributes

Bitmap indexes are index structures tailored to data warehouses. These indexes have already been used in some commercial products to speed up query processing: the main early example was Model 204, a prerelational DBMS from Computer Corporation of America. Many DBMS vendors, including Oracle, Informix, and Sybase, have introduced bitmap indexes into their products.

In its simplest form, a bitmap index on an index consists of one vector of bits (i.e., bitmap) per attribute value, where the size of each bitmap is equal to the number of records in the indexed relation. For example, if the attribute is day of the week, then there would be seven bitmap vectors for that attribute, one for each day. The bitmap vector corresponding to Monday would have a 1 at position i if record i contains Monday in the day-of-week attribute. This approach is called a value-list index.

Other techniques associate bitmap vectors with ranges of values, so there could, for a salary attribute, be a vector for the range 0 to 20,000 euros, 20,000.01 to 35,000 euros, and so on. Still others associate each bitmap vector with a bit value (a 1 or a 0) in a given position. So, if the attribute holds 32 bit numbers, then there would be 64 bitmaps (position 1, bit value 1; position 1, bit value 0; position 2, bit value 1;…; position 32, bit value 1; position 32, bit value 0). Such bitmaps may be useful particularly if the attribute is a mask where each bit position represents a binary alternative (male/female; healthy/sick; etc.). Figure 10.1 illustrates such bitmaps for the l_linestatus and l_returnflag attributes of the TPC-H line item relation that we are using for our experiments.

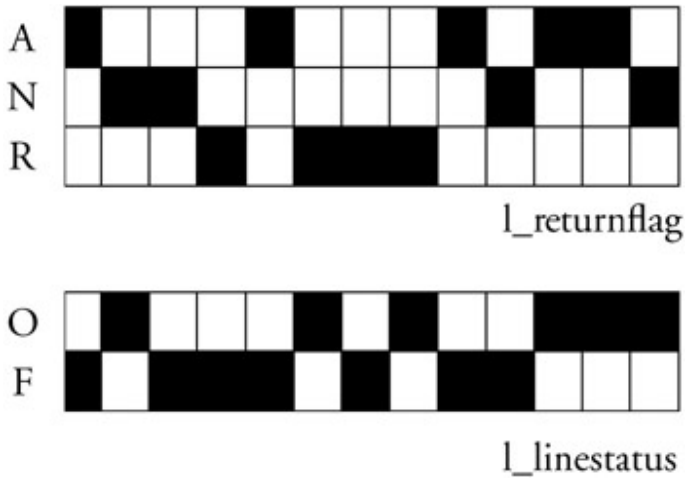


Figure 10.1: Bitmap. The l_returnflag attribute takes three possible values (A, N, R), while l_linestatus takes two values (O, F). The bitmap associates a value to each record in the lineitem relation. In the figure, records are represented vertically, and the associations are marked with black rectangles: the first record (from the left) has value A in the l_returnflag attribute and value F in the l_linestatus attribute.

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 (Figure 10.2). 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 age bitmap representing the ages between 30 and 40, and so on. One of the first users of bitmaps was the CIA.

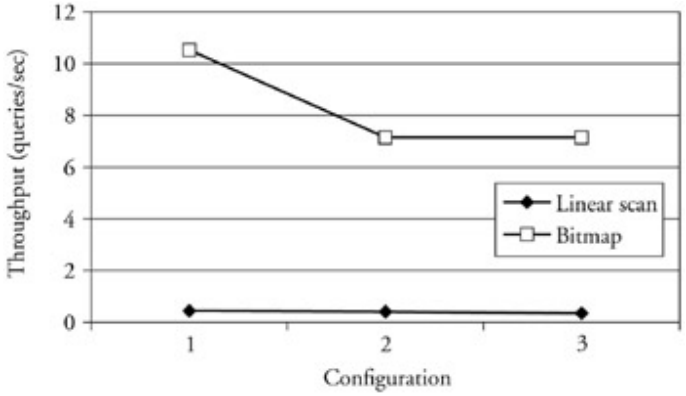


Figure 10.2: Bitmaps. We use the lineitem relation from the TPC-H benchmark for this experiment. We run a summation query, and we vary the WHERE clause so that it involves 1, 2, or 3 attributes. The attributes involved are l_returnflag, l_linenumber, and l_linestatus. The entire lineitem relation contains 600,000 records; the conditions on each attribute have approximately a 50% selectivity. The query on one attribute selects 300,000 records, the query on two attributes selects 100,000 records, and the query on three attributes selects 2000 records. To avoid crossing the application interface too often, each query is an aggregate that returns one record. We compare the performance of these aggregation queries using linear scan and bitmap indexes constructed on single attributes. This graph shows that bitmaps yield a significant performance improvement: the throughput is increased by an order of magnitude compared to a linear scan.

Note that the experiment is conducted with a warm buffer, so CPU cost dominates. The graph shows that there is a slight overhead when combining several bitmaps. This experiment was conducted on Oracle 8i EE on Windows 2000.

Comparing bitmaps with matrix approaches, we see that bitmaps work best if queries apply to many attributes, each having few values, and the result is a set of rows or an aggregate (though the advantage is less if the result is an aggregate). Multidimensional array approaches work better if there are few attributes in a query (especially if they are the dense ones) and the query seeks an aggregate.

If you'll excuse the caricature, the marketing department might prefer aggregates to detect trends, whereas the customer relationship management group might want information about classes of individuals (e.g., those whose flights have been frequently delayed).

Multidimensional indexes

Ideal application parameters: deep and broad, static, few query attributes with range queries on at least two of them

Multidimensional indexes are an alternative way to access multiple attributes. Multidimensional indexes such as quadtrees, R-trees, and their successors are conceptually grids on a multidimensional space. In the simplest case, you can imagine a big checkerboard. However, the grid cells are of different sizes, the population of points differs on different places in a hyperspace. For intuition, consider a map of equipopulation rectangles of France. The rectangles would be far more dense in Paris and in Nice than in the Alps. Indexes like this work well for spatial data queries (where they are used to find the points contained in latitude-longitude quadrants). The geographical example illustrates an important advantage of multidimensional indexes over composite indexes. A multidimensional index over latitude-longitude allows a range specification over both latitude and longitude, whereas a composite index on the attributes (latitude, longitude) works best for equality on latitude and a range query over longitude.
One popular structure is the R-tree, which consists of hierarchically nested, possibly overlapping boxes (Figure 10.3). The tree is height balanced, making it good for large databases. Variants of the quadtree also work well for such applications.

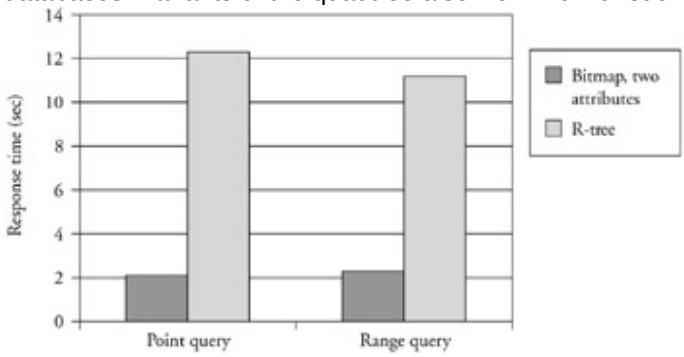


Figure 10.3: R-tree. We use a synthesized relation on top of the spatial extension to Oracle 8i to compare point and range queries on two-dimensional data using an R-tree and bitmaps. We define two-dimensional points as a spatial data type, and we use spatial functions in the queries. Using bitmaps, the *X* and *Y* coordinates of each point are simply represented as integers. In both cases, the relation contains eight other attributes; there are few distinct points stored in this relation. The graph shows the response time of a point query and a range query using bitmaps and the R-tree. The R-tree index loses. This is largely due to the over-head of using spatial functions to encode simple point or range queries. This result suggests that an R-tree (or at least this implementation of an R-tree) should be used only to support far more complex spatial

operations, such as overlap or nearest neighbor search, on spatial objects such as polygons. This graph comes from data from Oracle 8i running on Windows 2000.
Multidimensional indexes are little used outside geographical applications, however, because they do not scale well with increasing dimensionality, and commercial systems typically have far more than three dimensions (as in the CIA-style query). The scalability problem comes from the fact that a multidimensional index on attributes A_1, A_2,…,A_k alternates its search criterion among the different attributes at different levels of the tree. For example, consider the case when k=3. The top level may select based on A_3, then the next level based on A_1, then the next based on A_2, then on A_3 again, and so on. So, a search that is unselective for attribute A_1 will have to search many subtrees below the A_1 level.

Materialized views

Ideal application parameters: broad, static, also dynamic at a cost

If you don't have the luxury to design new indexes on top of a database system, you can precompute a large number of anticipated aggregate queries. For example, if a large retailer frequently asks queries that sum the total sales across multiple stores or multiple products, you may store such information in special tables. The main cost of such a strategy is maintaining these tables in the face of updates. (Disk space is no longer a significant cost.) In the example, every sale of item I at store S would have to update the total product sales for I and the total store sales for S. So, this strategy is worthwhile if there are few updates between queries, and is not worthwhile if there are many. Oracle 9i, Informix, and others use this strategy, and other vendors have at least considered it (SQL Server 2000 allows indexed views, but it requires the definition of a unique clustered index on those).

Product support for materialized views is still evolving.

- They implement different strategies for maintaining the materialized views: incremental or rebuild. Oracle 9i allows users to make this choice when creating the materialized view. RedBrick, an Informix product, automatically chooses a strategy based on the type of aggregate in the materialized view, the type of modification to be propagated, and statistics about the base table and the materialized view.
- The optimizer uses these precomputed materialized views by rewriting the submitted query on the base relations (if it is cost-effective) to use the materialized views without requiring user intervention. It is worthwhile to check query plans (see Appendix E) to make sure that the optimizer is using the correct view.
In Chapter 4, we discussed aggregate maintenance using materialized views. We illustrated the trade-off between query speed-up and view maintenance cost. We showed that the cost of view maintenance goes down if it is possible to separate insertions and view maintenance (refresh on demand). This is possible in a data warehouse where loading/insertions are clearly separated from queries.

Table caching entails some management overhead. For example, the data warehouse of a small German city has 400 materialized views. Refreshment time is a problem. In such a situation, it would be good to

1. eliminate redundant views.
2. ensure that all the materialized views are in fact used at least once between updates to the data warehouses; others can be calculated only when needed.
3. find an order to calculate the views that allows one view to be computed from already materialized views. To do this, use a little graph theory.

a. Let the views and the base tables be nodes.

b. Let there be an edge from a node v_1 to v_2 if it is possible to compute the view in v_2 from v_1 (possibly with the help of some dimension tables in the base relations). Such a calculation is often possible for different views having to do with the same "dimension." For example, location has attributes like city, country, and so on. So, a view that sums the sales by country can be computed from a view that sums the sales by city along with the information relating city to country.

c. Make the cost of the edge from v_1 to v_2 be the cost to compute view v_2 from v_1.

d. Now compute the all pairs shortest path (look this up in your first-year algorithms text if you have a chance) where the start node is the set of base tables. The all pairs shortest path can be programmed to return a set of edges that are used in the paths from the start node to all other nodes.

e. The result is an acyclic graph A. Take a topological sort (again, consult your algorithms text) of this graph and let that be the order of view construction.

f. Construct a view v from v' when the resulting acyclic graph has an edge from v' to v. Figure 10.4 illustrates the result of this algorithm on an example.

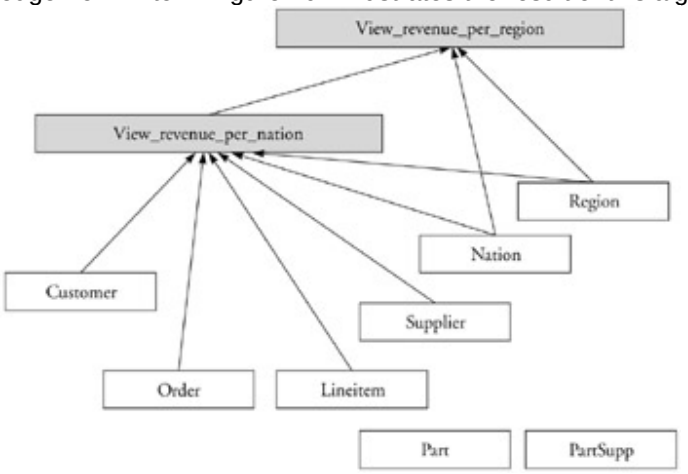


Figure 10.4: Materialized view creation graph. This graph illustrates the result of the materialized view creation algorithm applied to the TPC-H schema, and two aggregate materialized views, revenue_per_nation and revenue_per_region (in TPC-H regions actually denote continents). The revenue is computed using a six-way join (the TPC-H query Q5 is used to define the aggregate materialized view). The graph shows that the revenue_per_region view should be computed from the existing revenue_per_nation view and two base relations, instead of being computed from scratch using the base relations. The algorithm suggests the order of view construction: first revenue_per_nation, then revenue_per_region.

The algorithm as it stands fails to capture the possibility that a view might best be computed from several other views. There are two reasons we have not included this: (1) (empirical) the single dependency model captures most of the useful dependencies very well as in the nation/region example of Figure 10.4; and (2) (pragmatic) the all pairs shortest path problem is NP-complete for hypergraphs.[3]

Nevertheless, as a heuristic, the following strategy should work well in most cases. First perform the computation as above using single views to compute other views. This results in an acyclic graph A computed from the all pairs shortest path algorithm. Now consider some v that is best computed from multiple other views $v_1, v_2, \ldots, v_k$. Tentatively add edges from each v_i to v to the graph A. If the result is still acyclic, then leave the edges there. Otherwise, remove those edges. This keeps the graph acyclic. After forming the topological sort, if a view v must be computed, use multiple views if that is the best method to compute v and the multiple views are available.

Optimized foreign key joins

Ideal application parameters: deep and broad, ranges from static to dynamic, large data
Most queries in data warehouses entail joins between a fact table (e.g., line order detail in our running example) and a set of dimension tables (e.g., the location table in our example). Each dimension table consists of a collection of related attributes, say, all those having to do with product categorization or all those having to do with location.[4] Now the query "find the sales of hunting gear in the Northwest" entails a join between line order detail (which might record customer id, product ids, store id, quantity, and price) and product category as well as location. Such joins are known as foreign key joins because the product identifier in the line order table, for example, is a key of the product category table. (Recall that a key is a value that is unique in a table.) Figure 10.5 illustrates the presence of foreign keys in the TPC-H schema. The arrows point from the detailed fact tables to other tables called dimension tables. When a single fact table points to all the dimension tables, the schema is called a *star schema*. When some dimension tables point to subdimension tables, providing support for attribute hierarchies (and usually obtaining better normalization), the schema is called a *snowflake schema*. If several fact tables (tables that are sources but not destinations of arrows) share dimension tables, then the schema is called a *constellation schema*.

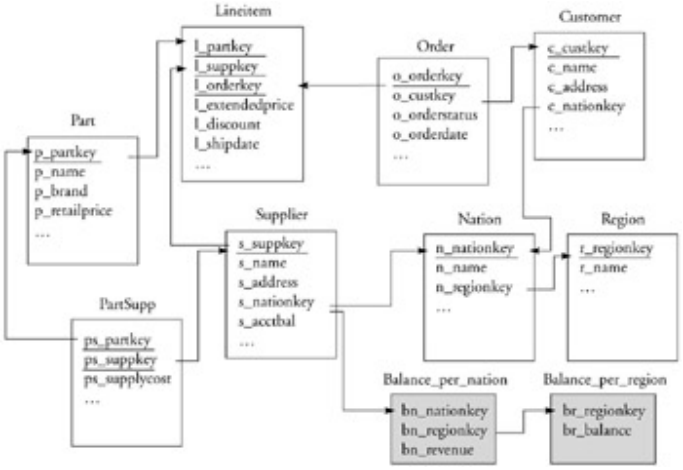


Figure 10.5: Foreign key dependencies. The base tables in the TPC-H schema form a *constellation schema*. Lineitem and PartSupp are fact tables; Supplier, Part, and Order are dimension tables. The arrows on the diagram correspond to foreign key dependencies. If we consider the two aggregate materialized views balance_per_nation and balance_per_region (sum of supplier account balances s_acctbal grouped by nation or per region), foreign key dependencies link dimension tables (e.g., Supplier) and aggregate fact tables (e.g., balance_per_nation).
One way to optimize these joins is to create a linkage between fact table records and dimension records. This can be done in several ways: (1) create an index that holds fact table record identifiers and dimension table record identifiers, an idea first explored in detail by Patrick Valduriez;[5] (2) create bidirectional pointers between fact table records and dimension table rows—this is what object-oriented databases do; or (3) replace the product record identifiers in the fact table by offsets into the dimension tables. The first choice is most robust to changes in the physical organization of the tables and therefore is best for heavily updated systems. The second choice is the least flexible to physical reorganization. The third choice entails marking deletions in dimension tables as opposed to removing the record in order to avoid changing the positions of undeleted elements. This is done, for example, in KDB (http://www.kx.com). Choice three allows the fact table to be reorganized at will, however.

Approximating the result

Ideal application parameters: broad, dynamic, large data

People who use data warehouses to get strategic aggregate information would be happy with a fast approximation as long as they know the answers are close. There has been recent excellent research in this area, for example, at Bell Labs (Gibbons and Poosala), at Berkeley (Hellerstein and Haas), at Microsoft (Surajit Chaudhuri), and at the University of North Carolina (Vitter and students). On a single table, classical statistical techniques can be used for fields with the property that no small number of records could have a large influence on results. Suppose you want the average of all salaries. Begin to compute the average on increasing sample sizes. If the average value changes little with increasing size, then it is probably accurate for all the data. Haas and Hellerstein call this idea "rippling."

For multiple tables, the ripple join work at Berkeley has shown that it is possible to achieve increasingly accurate results as a join proceeds. This requires that the ripple join idea be incorporated into the database management system you happen to be using.
The Bell Labs Aqua work by contrast is interesting even on top of existing database management systems. The goal is to estimate aggregate results in data warehouses while giving error bounds. Figures 10.6, 10.7, and 10.8 illustrate the benefits. The basic problem is that if you sample all tables and then do aggregates, that doesn't work in general. For example, if *R* and *S* have the same set of keys and you join the key of *R* with a key of *S*, then taking a 1/10 sample of each will give a size that is 1/100 of the size of the real join. So, you must be more clever. Their basic idea is to take a set of tables *R*, *S*, *T*, … that are linked by foreign key joins. Suppose, for example, that *R* is the fact table and the others are dimension tables. Take a random sample *R′* of *R* and then perform foreign key joins based on *R′* yielding *S′*, *T′*, …. Now, if a query involves *R*, *S*, *T* and includes the foreign key links among these, then the query can be done with great accuracy on *R′*, *S′*, *T′*. The error can be estimated by considering the result obtained by several subsets of *R′* and studying their variance.

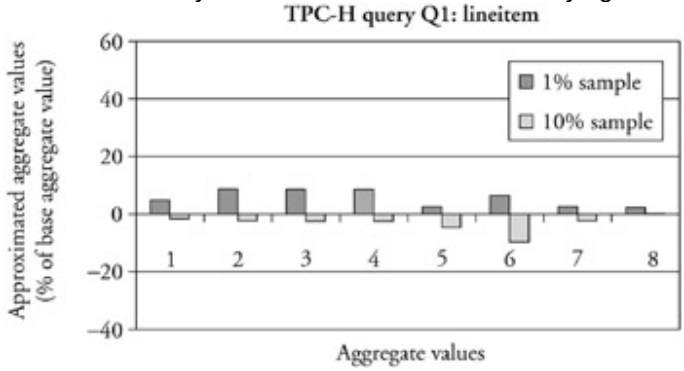


Figure 10.6: Approximation on one relation. We sample 1% and 10% of the lineitem table by selecting the top *N* records on an attribute in the fact table (here l_linenumber). That is, we are taking an approximation of a random sample. We compare the results of a query (Q1 in TPC-H) that accesses only records in the lineitem relation. The graph shows the difference between the aggregated values obtained using the base relations and our two samples. There are eight aggregate values projected out in the select clause of this query. Using the 1% sample, the difference between the aggregated value obtained using base relations and sample relations is never greater than 9%; using a 10% sample, this difference falls to around 2% in all cases but one.

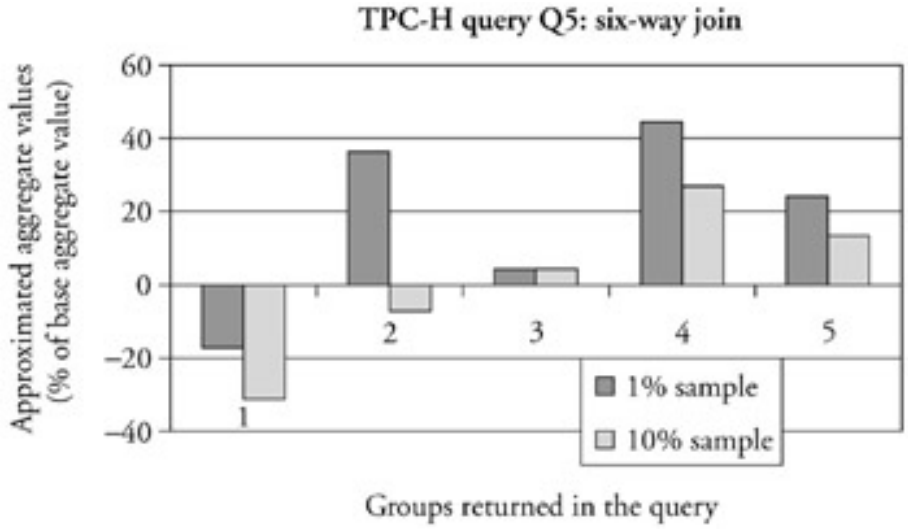


Figure 10.7: Approximation on a six-way join. As indicated in this section, we take a sample of the lineitem table and join from there on foreign keys to obtain samples for all tables in the TPC-H schema. We run query Q5, which is a six-way join. The graph shows the error for the five groups obtained with this query (only one aggregated value is projected out). For one group, using a 1% sample (of lineitem and using the foreign key dependencies to obtain samples on the other tables), we obtain an aggregated value that is 40% off the aggregated value we obtained using the base relations, and using a 10% sample, we obtain a 25% difference. As a consequence of this error, the groups are not ordered the same way using base relations and approximated relations.

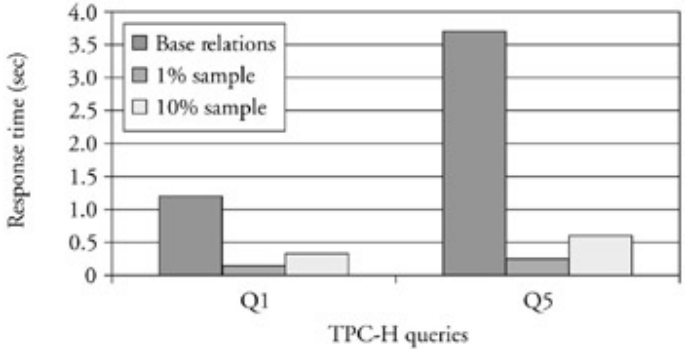


Figure 10.8: Response time benefits of approximate results. The benefits of using approximated relations much smaller than the base relations are, naturally, significant.

[1]The information in this section comes from conversations with data warehouse developers and consultants, particularly Felipe Carino and Mark Jahnke, mostly on Teradata hardware.

[2]Robert J. Earle. Arbor Software Corporation. U.S. patent # 5359724, October 1994.

[3]A view v that depends on v_1, v_2,…,v_k corresponds to a hyperedge where v_1,…,v_k are considered to be sources of the hyperedge and v is the target of the hyperedge. The proof of NP-completion is by reduction from the hitting set problem. The smallest set of views from which all others can be computed constitutes a hitting set.

[4]Often dimension tables in these *star* or *constellation* schemas are not normalized, so you might have store, city, and country in one table even though there is a functional dependency from city

to country. This is done in order to avoid an extra join and is reasonable since data warehouses are updated only occasionally.

[5]Patrick Valduriez. "Join Indices." *Transactions on Database Systems*, 12(2), 1987.

10.2 Tuning for Customer Relationship Management Systems

As suggested, customer relationship management (CRM) systems are built on top of data warehouses. Since CRM systems typically involve cross-selling (e.g., a person who buys long-distance telephone service from me may want to buy roadside repair service from me, too), they normally include four ingredients.

- A data warehouse that gives a unified view of each customer, which services he or she has purchased, and any available demographics
- A data mining toolkit to infer strategies for improving sales from the data
- The placement of "business rules" derived from the strategies into either the data warehouse, the transactional database, or some special database
- The use of those rules

Data mining toolkits normally extract a denormalized sample of data from a large transactional system or from a data warehouse and then process it. The processing occurs outside the database system and typically involves multiple passes over the data for the purpose of identifying correlations (e.g., older people who have cars are good candidates for roadside service) or building a decision tree. By the partitioning tuning principle, that processing should occur far from our production database hardware. So the main impact on the database of the analytical portion of CRM is a potentially expensive query on the data warehouse. See the box for some guidance about which data mining approach to use.

CHOOSING THE RIGHT DATA MINING ALGORITHM

The goal of data mining is to discover rules, such as who makes a good credit risk, who is likely to be a repeat customer, and which customer is likely to buy a yacht. You will apply those rules to new and existing customers.

Some data mining specialists like building decision trees. Some like neural nets. But the fact is that the same algorithm does not work well for all applications.[6]

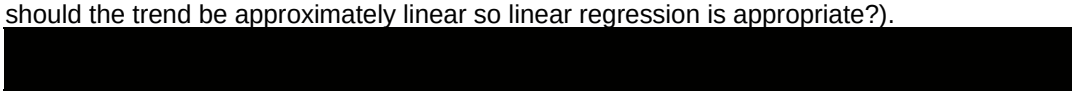
Here are the main methods.

- *Decision tree methods* include symbolic or logic learning methods such as ID3 or C4.5 that use the divide-and-conquer paradigm to generate decision trees.
- *Case-based methods* extract representative examples from the data set to approximate the knowledge hidden in an information repository. The idea is to apply to new situations the lessons learned from "similar" cases in the past.
- *Statistical methods* encompass various techniques, including linear and nonlinear regression, Bayesian methods, and clustering methods.
- *Biological methods* include neural nets and genetic algorithms. Neural nets are acyclic networks in which edges represent excitation or inhibition, source nodes correspond to input data, and end nodes represent a decision. Genetic algorithms emulate the Darwinian model of evolution, in which "chromosomes" represent candidate solutions, reproduction involves crossover, and mutation of chromosomes in which the "fittest" (so far, best) chromosomes have the highest chance to succeed.

The characteristics of an application should guide the choice of method.

- *Categorical/continuous*. Applications in which categorical data (sex, hobbies, socioeconomic group, etc.) has a large influence on a decision should not use neural nets. Applications in which continuous data alone (e.g., temperature, pressure, wavelength) may influence a decision are well suited to neural nets though the other methods can also apply.
- *Explainability*. If users may ask for the reasons behind a decision, then decision trees are the best choice. Case-based approaches come in second. There is no third.
- *Time to train*. Each mining data structure must be built before it can be used. For example, a decision tree must be built before it is used. Training may occur once and for all but usually occurs on a regular basis (e.g., every week). Decision trees and neural networks are expensive (superlinear) to train. Genetic algorithms give better results the more training time they are given. Scalable statistical methods such as clustering require time linear in the size of the data. Case-based methods require minimal training.
- *Time to apply to new cases*. Once a mining data structure is built, it is used to make decisions about input data. Decision trees, neural networks, and genetic algorithms do this fast. Statistical methods and case-based methods are slower.
- *Accuracy*. All methods will give an answer. The quality is another story and depends on a collection of parameters associated with each package.

- Decision tree methods suffer from the possibility of "overfitting" the training data (i.e., few records determine a rule). This may lead to incorrect rules, especially when the data is dirty (which, as Celko points out, is almost always).
- Case-based methods are very sensitive to the a priori notion of similarity that determines which cases are "similar" to a new input case. For example, if you have customer information and want to determine who will buy a yacht, age may be relevant (90-year-olds don't buy yachts), income is certainly relevant, but shoe size is not. One approach is to form a decision tree and then use that to determine which attributes should be most important in the similarity measure.
- In K-means-style clustering methods, where K determines the number of clusters, the parameter K as well as the similarity measure used for clustering can have a large effect on the useful ness of the clustering.
- Neural networks require a normalization step in which data is converted to a value between 0 and 1. That can introduce biases. Statistical methods may suffer from inaccuracies having to do with initial statistical assumptions (e.g., should the trend be approximately linear so linear regression is appropriate?).



The placement of business rules (e.g., if the customer has had a telephone in his name for 30 years, he's probably old enough to need roadside service) into the database is straightforward and normally doesn't interfere with other updates.

The use of business rules normally requires a join. For example, suppose you sell telephone services and I call up to ask for some. To know whether I am a good candidate for roadside services, your computer system must look me up in the business rules database. An index can make this efficient.

Some enterprises use business rules to generate mass mailings. For example, they may want to find all people who live in California, like to surf, and own a house. Bitmaps are useful for such applications.

The bottom line is that 90% of the work that goes into building the CRM system is building the data warehouse.

[6]Most of the observations of this subsection come from Hany Saleeb, a senior applications designer at Oracle.

10.3 Federated Data Warehouse Tuning

A *federated data warehouse* is a data warehouse constructed on top of a set of independent databases. That is, the data warehouse is analogous to a view table: it offers a logical picture of a collection of data that physically resides on different databases. One more server, the "federator," does the necessary final joins and may hold data itself. As Ron Yorita of IBM reports:

The DB2 optimizer attempts to find the least cost access to data by looking at various join methods, ordering of the joins, indexes, etc. When you add datajoiner/federated multiple remote tables into the equation,… the network cost of transferring data becomes a factor.… The bulk of optimization revolves around ensuring that remote sources do as much work as possible and return a minimal amount of data for additional processing at the federated server.

He suggests some more tuning optimizations:

1. When defining a remote server (create server) to a federated database, the default is to push down (pushdown = 'Y') operations to the remote source. This will allow the federated server more flexibility in optimization. If you specify 'N', the remote source must return data to the federated server, and the remote source is limited to processing/filtering. So use the default for pushdown.
2. Since DB2 and federated use cost-based optimization, the statistics about tables and indexes feed directly into the query plan selection process. It's surprising how often the statistics are not accurate, which in turn results in bad query plans. Maintaining current statistics is system dependent. For example, to gather current statistics on an Oracle server, you must run the analyze utility. Then to refresh the federated statistics, drop and re-create the nickname for the table (with DataJoiner you can just do a runstats instead of drop/create).
3. When defining a data source (create server), there is a parameter called collating_sequence (colseq). If you set the parameter to yes, you are asserting that certain character fields in the server are in the same order as in the virtual DataJoiner database. This can make operations like Min, Max, and merge join enjoy improved performance. Of course if the assertion is wrong, you get incorrect results.

The query optimizer can do strange things. Yorita reports:

I was told of one situation that involved multiple remote sources. Two of the tables were on one remote source, but the optimizer decided to decompose the query. This meant that the join of the two tables was not done on the remote source and had to be processed on the federated engine. In order to push down the join, they eventually created a view. The view did the table join on the remote system. A nickname was created on the server using the view. The query was revised to reference the nickname for the view. This then allowed the join processing to be done at the remote server.

10.4 Product Selection

It is hard for a small company with a great idea to break into the transaction processing marketplace. Customers are wary of systems that will replace their transaction processing systems because their databases are the enterprises' lifeblood. A database vendor that is here today and gone tomorrow can render this lifeblood inaccessible. Data warehouse products that derive their data from the database of record are less of a risk since the transactional data is still available even if the warehouse goes away.

So, a customer can reason as follows: "Even if data warehouse vendor X goes out of business, my data and applications are safe. I may have to buy more hardware or eliminate some queries, but I'm still very much in business."

Until a few years ago, a large player in software-based relational data warehousing was RedBrick. Their product embodied several clever ideas resulting in both expressibility and speed. Expressibility came through extensions to SQL, for example, year-to-year sales comparisons, running sums, and bucketing (such as grouping products into high sellers, medium sellers, and low sellers). They offered high speed through optimization for star schemas. Recall that a star schema is one in which a large central table called the fact table (e.g., every line order) is linked to a set of satellite tables (giving detail of the location of the store, the vendor of the product sold, etc.). They offered two special kinds of indexes called targetindexes and starindexes. Both of these are multitable indexes (meaning they related data in multiple tables) that mix bitmap and B-tree capabilities.
As of this writing, however, Microsoft SQL Server seems to be winning the TPC-H benchmarks for 100 gigabytes or less. Redbrick (still part of Informix when the benchmark results were published in the spring of 2001, now owned by IBM) comes into its own at 300 gigabytes. Note that the benchmark results are constantly evolving and that database and hardware providers take turns as the top performers (http://www.tpc.org/new_result/h-ttperf.idc). The TPC-H benchmark is a broad benchmark in that it is oriented toward statistical queries.

Teradata produces huge systems holding several terabytes and built on special parallel hardware for very large customers such as Wal-Mart, Delta Airlines, Royal Bank of Canada, large hospitals, the express mail services, and so on. The application area is aggregate targeting. Their basic strategy is to horizontally partition their data (so the rows of each table are partitioned across many different sites). This permits them to divide the work of processing queries across many sites. Their current strategy is to support what they call "active data warehouses" wherein their large databases implement business rules as well as gather statistics. For example, recall a frequent flier on an airline X who encounters two late flights in two weeks. An active data warehouse will contact that customer and offer him perks because data mining on the data warehouse reveals that customers having bad experiences in a short period of time are at risk of switching to another airline.

Bibliography

Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. *Join synopses for approximate query answering.* In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings ACM SIGMOD International Conference on Management of Data*, June 1–3, 1999, Philadelphia, Pa., 275–286. ACM Press, 1999.

Benchmark Results: http://www.tpc.org/new_result/h-ttperf.idc.

Chee Yong Chan and Yannis E. Ioannidis. *Bitmap index design and evaluation.* In Laura M. Haas and Ashutosh Tiwary, editors, *Proceedings ACM SIGMOD International Conference on Management of Data*, June 2–4, 1998, Seattle, Wash., 355–366. ACM Press, 1998.

Surajit Chaudhuri, Gautam Das, and Vivek R. Narasayya. *A robust, optimization-based approach for approximate answering of aggregate queries. Proceedings ACM SIGMOD International Conference on Management of Data*, 295–306. ACM Press, New York, 2001.

Surajit Chaudhuri and Umeshwar Dayal. *An overview of data warehousing and OLAP technology. SIGMOD Record*, 26(1):65–74, 1997.

Excelon: http://www.exceloncorp.com/.

Antonin Guttman. *R-trees: A dynamic index structure for spatial searching.* In Beatrice Yormark, editor, *SIGMOD'84 Proceedings of Annual Meeting*, Boston, Mass., June 18–21, 47–57. ACM Press, 1984.

Peter J. Haas and Joseph M. Hellerstein. *Ripple joins for online aggregation.* In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings ACM SIGMOD International Conference on Management of Data*, June 1–3, 1999, Philadelphia, Pa., 287–298. ACM Press, 1999.

Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. *Implementing data cubes efficiently.* In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Quebec, Canada, June 4–6, 205–216. ACM Press, 1996.

Hyperion: http://www.hyperion.com/.

Theodore Johnson and Dennis Shasha. *Some approaches to index design for cube forest. Data Engineering Bulletin*, 20(1):27–35, 1997.

KDB product: http://www.kx.com.

Ralph Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses.* John Wiley & Sons, 1996.

Yannis Kotidis and Nick Roussopoulos. *An alternative storage organization for ROLAP aggregate views based on cubetrees.* In Laura M. Haas and Ashutosh Tiwary, editors, *Proceedings ACM SIGMOD International Conference on Management of Data*, June 2–4, 1998, Seattle, Wash., 249–258. ACM Press, 1998.

Yannis Kotidis and Nick Roussopoulos. *Dynamat: A dynamic view management system for data warehouses.* In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings ACM SIGMOD International Conference on Management of Data*, June 1–3, 1999, Philadelphia, Pa., 371–382. ACM Press, 1999.

Oracle Corp. *Performance and scalability in DSS environment with oracle 9i.* Oracle white paper, April 2001.

RedBrick Software: http://www.ibm.com/software/data/informix/redbrick/.

Hanan Samet. *The quadtree and related hierarchical data structures. ACM Computing Surveys*, 16(2):187–260, 1984.

Sunita Sarawagi. *Indexing OLAP data. Data Engineering Bulletin*, 20(1):36–43, 1997.

Stephen J. Smith, Alex Berson, and Kurt Thearling. *Building Data Mining Applications for CRM.* McGraw-Hill Professional Publishing, 1999.

Sybase IQ Administration Guide. *Sybase IQ Indexes.* Sybase IQ Collection, 1997.

Teradata software: http://www.teradata.com/.

Erik Thomsen. *OLAP Solutions: Building Multidimensional Information Systems.* John Wiley & Sons, 1997.

TimesTen: http://www.timesten.com.

Patrick Valduriez. *Join indices. TODS*, 12(2):218–246, 1987.

Exercise

EXERCISE 1

Your customers want to identify people having general characteristics, for example, in their 30s, driving sports cars, married, five kids, and so on. None of these characteristics by itself is very selective, but taken together they might be. What would be an appropriate data structure and why?
Action. Neither an OLAP system nor materialized views will help identify individuals. A set of unselective conditions is best handled either by a set of bitmaps or, if bitmaps are unavailable, by a powerful scan engine.

MINIPROJECT
There are a group of data warehouse–related performance experiments for you to do at the book's Web site at http://www.mkp.com/dbtune/. There you will find code to modify and an example run. Some of these experiments require external components. The projects concern the following issues (for the sake of concreteness, we use the TPC-H data):

1. If you have access to an OLAP system, record the space and construction time overhead of constructing data structures with roll-ups on profit and volume and revenue per month, per year, and per nations and regions.
2. Define materialized views for profit, volume, and revenue per month, per year, per nation, and per region. Define the view computation graph.
3. Compare query execution times using this OLAP system with a relational system with and without materialized views.
4. Denormalize supplier and customer by including the geographic information about nation and region. What is the impact on query execution time and on memory usage?
5. Find a sampling method for line item that minimizes the error on Q5 when using the Aqua method.

Appendix A: Real-Time Databases

A.1 Overview

A real-time database is one where some of the transactions must meet timing constraints. For most database applications, these constraints are of a statistical nature (e.g., 90% of all transactions must complete within one second and 99% within five seconds). Such applications are known as *soft real time* and apply to applications such as telemarketing, financial analysis, and even to certain kinds of industrial control.

Other real-time applications are characterized as "hard" if failure to meet a deadline can lead to catastrophe. The prototypical example is a jet aircraft control system that must keep an "inherently unstable" jet from crashing. Database applications in such domains require main memory databases, simplified locking, and either main memory recovery mechanisms or no recovery at all.

- Allocate time-critical application data to main memory, either by storing all important data in main memory or by placing some tables in favored buffers. Oracle and DB2 UDB allow database administrators to associate a buffer with specific tables. This is very useful in cases in which all timing constraints concern accesses to some table (or a few tables). If one table is critical, for example, then dedicating a large buffer to that table and a large enough slice of main memory to that buffer ensures that all important data is in main memory.
- Try to establish predictable lock patterns to avoid deadlocks and to minimize delays resulting from lock conflicts.
  A real-time systems designer has the advantage of knowing what set of transactions is possible. He or she should use that information to chop transactions as discussed in Appendix B, as well as to order the accesses within each transaction.
  The question of order is essential to avoid deadlocks. One promising technique is to identify a total order among the lockable items (whether records or pages) and lock them in that order. For example, if transaction type 1 locks *X*, *Y*, and *Z* and transaction type 2 locks *W*, *X*, and *Z*, and transaction type 3 locks *Z* and *W*, then they should all acquire locks in some order *W*, *X*, *Y*, and *Z*. That is, the acquisition order should be *W* before *X*; *W* and/or *X* before *Y*; and *W*, *X*, and/or *Y* before *Z*. This will eliminate the possibility of deadlock.[1].
- Give different transactions different priorities, but be careful. To avoid the priority inversion problem described in Chapter 2, you must give the same priority to any two transactions that issue conflicting locks to the same data item. That is, if two transactions access some item *X* and at least one of them obtains a write (i.e., exclusive) lock on *X*, then the two transactions should run at the same priority.
  On the other hand, if this rule does not force you to give transaction T_1 and T_2 the same priority and T_1 is time critical, then you might consider giving T_1 higher priority.
- Make your data structures predictable. Usually this means some kind of balanced tree structure, such as a B-tree, though TimesTen and KDB use trees with less fanout for the in-memory part.

  Overflow chaining can cause poor response times for tasks that must traverse those chains. (This will affect deletes even more than inserts because a delete must traverse an entire chain even if the item to be deleted is not present.)
- Minimize the overhead of the recovery subsystem.

1. Use the fastest commit options that are available. This includes options that delay updates to the database and that write many transactions together onto the log (group commit). The only disadvantage to fast commit options is that they may increase recovery times from failures of random access memory. Minimize the chance of such failures by ensuring that your power supply is reliable.

2. Checkpoints and database dumps can cause unpredictable delays to real-time transactions so should be deferred to quiet periods.

[1]A subtle problem can arise in systems in which the lockable items are larger than the data items of interest. For example, suppose that *W*, *X*, *Y*, and *Z* are records and the system uses page-level locking. Using the lock-ordering *W*, *X*, *Y*, and *Z*, you believe you have eliminated the possibility of deadlock. Unfortunately, however, *W* and *Z* are on the same page P_1, and *X* and *Y* are on the same page P_2. A transaction of type 1 may lock page P_2 at the same time that a transaction of type 2 locks page P_1. Each will seek to lock the other page, resulting in a deadlock.

A.2 Replicated State Machine Approach

If your database can fit in your memory and you need both high performance and reliability, then you can use a replicated state machine approach. The idea is very simple and elegant: *if you initialize a set of sites S1,…, Sn in the same state and you apply the same operations in the same order to all the sites, then they will end up in the same state.*

There are a few caveats.

1. The transactions must be deterministic and time independent. Effectively, this means they must not contain calls to random number generators or to the clock.
2. The transactions must appear to execute serially *in the order they arrive.* This is stronger than serializability by itself, which requires only that the transactions appear to execute in some serial order. For example, imagine that one transaction increases every salary by 10% and another increases Bob's salary by $10,000. If the 10% transaction appears to precede the $10,000 transaction on site A but follows the $10,000 transaction on site B, then Bob will have a higher salary on site B. Ensuring this stronger form of serializability is not hard: you detect transactions that conflict and forbid them from executing concurrently. (This is discussed in the paper "High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++" by Arthur Whitney, Dennis Shasha, and Steve Apter, 211–217, Seventh International Workshop on High Performance Transaction Systems, September 1997, Asilomar, Calif.)
One very significant benefit of executing transactions in the order they arrive (or appearing to) is that you can log transaction commands rather than modified records or pages. This reduces the overhead of the recovery subsystem substantially. It also eliminates the need for two-phase locking. That, too, is discussed in the Whitney et al. paper and is the basis for recovery in the KDB system obtainable from http://www.kx.com.

Appendix B: Transaction Chopping

B.1 Assumptions

This appendix continues the discussion in Chapter 2 about making transactions smaller for the purpose of increasing available concurrency. It uses simple graph theoretical ideas to show how to cut up transactions in a safe way. If there are no control dependencies between the pieces that are cut up, then the pieces can be executed in parallel. Here are the assumptions that must hold for you to make use of this material.

- You can characterize all the transactions that will run in some interval. The characterization may be parametrized. For example, you may know that some transactions update account balances and branch balances, whereas others check account balances. However, you need not know exactly which accounts or branches will be updated.
- Your goal is to achieve the guarantees of serializability. (This appendix sets degree 3 isolation as its goal instead of full serializability because it makes the theory easier to understand. To apply chopping to systems desiring full serializability, the conflict graph should include semantic conflicts, for example, the conflict between a read of everyone whose last name begins with some letter between B and S and the insertion of a record about Bill Clinton.) You just don't want to pay for it.

  That is, you would like either to use degree 2 isolation (i.e., write locks are acquired in a two-phased manner, but read locks are released immediately after use), to use snapshot isolation, or to chop your transactions into smaller pieces. The guarantee should be that the resulting execution be equivalent to one in which each original transaction executes in isolation.
- If a transaction makes one or more calls to rollback, you know when these occur. Suppose that you chop up the code for a transaction T into two pieces T_1 and T_2, where the T_1 part executes first. If the T_2 part executes a rollback statement in a given execution after T_1 commits, then the modifications done by T_1 will still be reflected in the database. This is not equivalent to an execution in which T executes a rollback statement and undoes all its modifications. Thus, you should rearrange the code so rollbacks occur early. We will formalize this intuition below with the notion of rollback safety.
- Suppose a transaction T modifies x, a program variable not in the database. If T aborts because of a concurrency control conflict and then executes properly to completion, variable x will be in a consistent state. (That is, we want the transaction code to be "reentrant.")
- If a failure occurs, it is possible to determine which transactions completed before the failure and which ones did not.

Suppose there are n transactionsT_1, T_2,…,T_n that can execute within some interval. Let us assume, for now, that each such transaction results from a distinct program. Chopping a transaction will then consist of modifying the unique program that the transaction executes. Because of the form of the chopping algorithm, assuming this is possible will have no effect on the result.

A *chopping* partitions each, T_i into *pieces* c_{i1}, c_{i2},…, c_{ik}. Every database access performed by T_i is in exactly one piece.

A chopping of a transaction T is said to be *rollback safe* if either T has no rollback statements or all the rollback statements of T are in its first piece. The first piece must have the property that all its statements execute before any other statements of T. This will prevent a transaction from half-committing and then rolling back.

A chopping is said to be *rollback safe* if each of its transactions is rollback safe. Each piece will act like a transaction in the sense that each piece will acquire locks according to the two-phase locking algorithm and will release them when it ends. It will also commit its changes when it ends. Two cases are of particular interest.

- The transaction *T* is sequential and the pieces are nonoverlapping subsequences of that transaction.
  For example, suppose *T* updates an account balance and then updates a branch balance. Each update might become a separate piece, acting as a separate transaction.
- The transaction *T* operates at degree 2 isolation in which read locks are released as soon as reads complete.
  In this case, each read by itself constitutes a piece.[1] All writes together form a piece (because the locks for the writes are only released when *T* completes).

EXECUTION RULES

1. When pieces execute, they obey the order given by the transaction. For example, if the transaction updates account *X* first and branch balance *B* second, then the piece that updates account *X* should complete before the piece that updates branch balance *B* begins. By contrast, if the transaction performs the two steps in parallel, then the pieces can execute in parallel.
2. If a piece is aborted because of a lock conflict, then it will be resubmitted repeatedly until it commits.
3. If a piece is aborted because of a rollback, then no other pieces for that transaction will execute.

[1]Technically, this needs some qualification. Each read that doesn't follow a write on the same data item constitutes a piece. The reason for the qualification is that if a write(*x*) precedes a read(*x*), then the transaction will continue to hold the lock on *x* after the write completes.

B.2 Correct Choppings

We will characterize the correctness of a chopping with the aid of an undirected graph having two kinds of edges.

1. *C edges*. C stands for *conflict*. Two pieces *p* and *p′* from different original transactions conflict if there is some data item *x* that both access and at least one modifies.[2] In this case, draw an edge between *p* and *p′* and label the edge C.
2. *S edges*. S stands for *sibling*. Two pieces *p* and *p′* are siblings if they come from the same transaction *T*. In this case, draw an edge between *p* and *p′* and label the edge S.

We call the resulting graph the *chopping graph*. (Note that no edge can have both an S and a C label.)
We say that a chopping graph has an *SC-cycle* if it contains a simple cycle that includes at least one S edge and at least one C edge.[3]

We say that a chopping of $T_1, T_2, \ldots, T_n$ is *correct* if any execution of the chopping that obeys the execution rules is equivalent to some serial execution of the original transactions. "Equivalent" is in the sense of the textbook.[4] That is, every read (respectively, write) from every transaction returns (respectively, writes) the same value in the two executions and the same transactions roll back. Now, we can prove the following theorem.

Theorem 1 A chopping is correct if it is rollback safe and its chopping graph contains no SC-cycle.
Proof: The proof requires the properties of a serialization graph. Formally, a serialization graph is a directed graph whose nodes are transactions and whose directed edges represent ordered conflicts. That is, $T \rightarrow T'$ if *T* and *T′* both access some data item *x*, one of them modifies *x*, and *T* accessed *x* first. Following the Bernstein, Hadzilacos, and Goodman textbook, if the serialization graph resulting from an execution is acyclic, then the execution is equivalent to a

serial one. The book also shows that if all transactions use two-phase locking, then all those who commit produce an acyclic serialization graph.
Call any execution of a chopping for which the chopping graph contains no SC-cycles a *chopped execution*. We must show that

1. any chopped execution yields an acyclic serialization graph on the given transactions T_1, T_2,…,T_n and hence is equivalent to a serial execution of those transactions.
2. the transactions that roll back in the chopped execution would also roll back if properly placed in the equivalent serial execution.

For point 1, we proceed by contradiction. Suppose there was a cycle in T_1, T_2, …, T_n. That is $T \rightarrow T' \rightarrow \ldots \rightarrow T$. Consider three consecutive transactions T_i, T_{i+1}, and T_{i+2}. There must be pieces p_1, p_2, p_3, and p_4 such that p_1 comes from T_i, conflicts with p_2 from T_{i+1}, which is a sibling of or identical to p_3 from T_{i+1} and that conflicts with p_4 of T_{i+2}. Thus the transaction conflicts in the cycle correspond to a path P of directed conflict edges and possibly some sibling edges. Observe that there must be at least one sibling edge in P because otherwise there would be a pure conflict path of the form $p \rightarrow p' \rightarrow \ldots \rightarrow p$. Since the pieces are executed according to two-phase locking and since two-phase locking never allows cyclic serialization graphs, this cannot happen. (A piece may abort because of a concurrency control conflict, but then it will reexecute again and again until it commits.) Therefore, P corresponds to an SC-cycle, so there must be an SC-cycle in the chopping graph. By assumption, no such cycle exists.
For point 2, notice that any transaction T whose first piece p rolls back in the chopped execution will have no effect on the database, since the chopping is rollback safe. We want to show that T would also roll back if properly placed in the equivalent serial execution. Suppose that p conflicts with and follows pieces from the set of transactions W_1, …, W_k. Then place T immediately after the last of those transactions in the equivalent serial execution. In that case, the first reads of T will be exactly those of the first reads of p. Because p rolls back, so will T.▪

Theorem 1 shows that the goal of any chopping of a set of transactions should be to obtain a rollback-safe chopping without an SC-cycle.

CHOPPING GRAPH EXAMPLE 1

Suppose there are three transactions that can abstractly be characterized as follows:

T1: R(x) W(x) R(y) W(y)

T2: R(x) W(x)

T3: R(y) W(y)

Breaking up T1 into

T11: R(x) W(x)

T12: R(y) W(y)

will result in a graph without an SC-cycle (Figure B.1). This verifies the rule of thumb from Chapter 2.

Figure B.1: No SC-cycle.

CHOPPING GRAPH EXAMPLE 2

With the same T2 and T3 as in the first example, breaking up T11 further into

T111: R(x)

T112: W(x)

will result in an SC-cycle (Figure B.2).

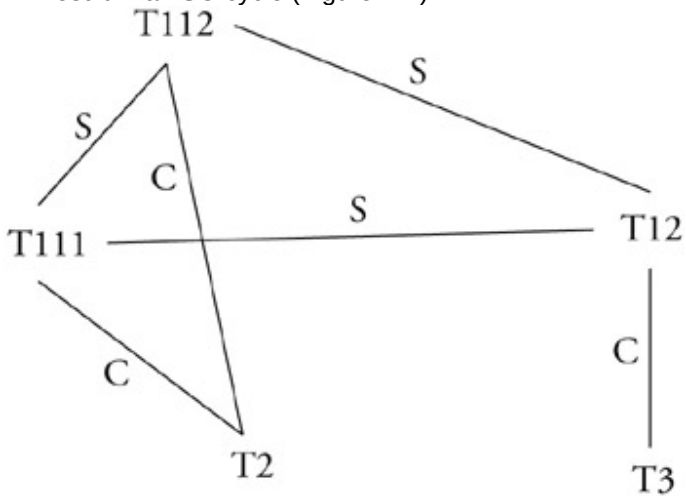


Figure B.2: SC-cycle.

CHOPPING GRAPH EXAMPLE 3

By contrast, if the three transactions were

T1: R(x) W(x) R(y) W(y)

T2: R(x)

T3: R(y) W(y)

then T1 could be broken up into

T111: R(x)

T112: W(x)

T12: R(y) W(y)

There is no SC-cycle (Figure B.3). There is an S-cycle, but that doesn't matter.

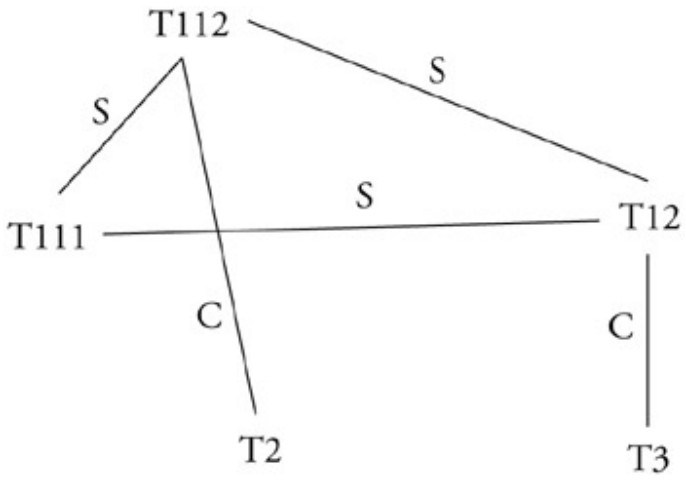


Figure B.3: No SC-cycle.

CHOPPING GRAPH EXAMPLE 4
Now, let us take the example from Chapter 2 in which there are three types of transactions.

- A transaction that updates a single depositor's account and the depositor's corresponding branch balance.
- A transaction that reads a depositor's account balance.
- A transaction that compares the sum of the depositors' account balances with the sum of the branch balances.

For the sake of concreteness, suppose that depositor accounts D11, D12, and D13 all belong to branch B1; depositor accounts D21 and D22 both belong to B2. Here are the transactions.

T1 (update depositor): RW(D11) RW(B1)

T2 (update depositor): RW(D13) RW(B1)

T3 (update depositor): RW(D21) RW(B2)

T4 (read depositor): R(D12)

T5 (read depositor): R(D21)

T6 (compare balances): R(D11) R(D12) R(D13) R(B1) R(D21)
R(D22) R(B2)

Thus, T6 is the balance comparison transaction. Let us see first whether T6 can be broken up into two transactions.

T61: R(D11) R(D12) R(D13) R(B1)

T62: R(D21) R(D22) R(B2)

The absence of an SC-cycle shows that this is possible (Figure B.4).

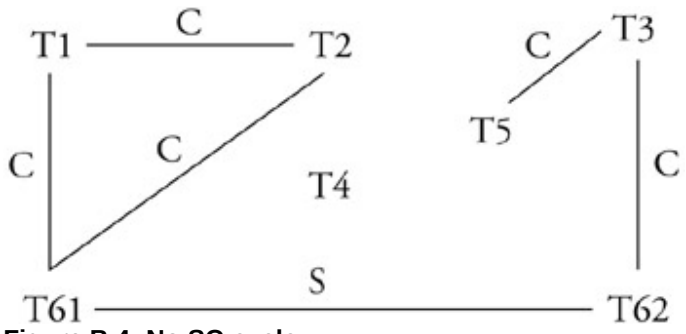


Figure B.4: No SC-cycle.

CHOPPING GRAPH EXAMPLE 5
Taking the transaction population from the previous example, let us now consider dividing T1 into two transactions, giving the following transaction population Figure B.5).

T11: RW(D11)

T12: RW(B1)

T2: RW(D13) RW(B1)

T3: RW(D21) RW(B2)

T4: R(D12)

T5: R(D21)

T6: R(D11) R(D12) R(D13) R(B1) R(D21) R(D22) R(B2)

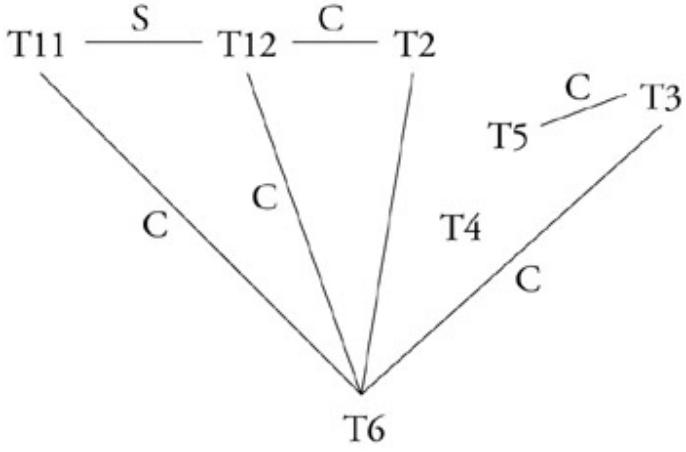


Figure B.5: SC-cycle.

This results in an SC-cycle.

[2]As has been observed repeatedly in the literature, this notion of conflict is too strong. For example, if the only data item in common between two transactions is one that is only incremented and whose exact value is insignificant, then such a conflict might be ignored. We assume the simpler read/write model only for the purposes of exposition.

[3]Recall that a simple cycle consists of (1) a sequence of nodes n_1, n_2,…, n_k such that no node is repeated and (2) a collection of associated edges: there is an edge between n_i and n_{i+1} for $1 \leq i < k$ and an edge between n_k and n_1; no edge is included twice.

[4]See the following two references:

P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Reading, Mass., Addison-Wesley, 1987. *research.microsoft.com/pubs/ccontrol/*.
Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery*. Morgan Kaufmann, May 2001.

B.3 Finding the Finest Chopping

Now, you might wonder whether there is an algorithm to obtain a correct chopping. Two questions are especially worrisome.

1. Can chopping a piece into smaller pieces break an SC-cycle?
2. Can chopping one transaction prevent one from chopping another?

Remarkably, the answer to both questions is no.

Lemma 1 If a chopping is not correct, then any further chopping of any of the transactions will not render it correct.
Proof: Let the transaction to be chopped be called *T* and let the result of the chopping be called pieces(*T*). If *T* is not in an SC-cycle, then chopping *T* will have no effect on the cycle. If *T* is in an SC-cycle, then there are three cases.

1. If there are two C edges touching *T* from the cycle, then those edges will touch one or more pieces in pieces(*T*). (The reason is that the pieces partition the database accesses of *T* so the conflicts reflected by the C edges will still be present.) Those pieces (if there are more than one) will be connected by S edges after *T* is chopped, so the cycle will not be broken.
2. If there is one C edge touching *T* and one S edge leaving *T* (because *T* is already the result of a chopping), then the C edge will be connected to one piece *p* of pieces(*T*). If the S edge from *T* is connected to a transaction *T′*, then *p* will also be connected by an S edge to *T′* because *p* and *T′* come from the same original transaction. So, the cycle will not be broken.
3. If there are two S edges touching *T*, then both of those S edges will be inherited by each piece in pieces(*T*), so again the cycle will not be broken.▪

Lemma 2 If two pieces of transaction *T* are in an SC-cycle as the result of some chopping, then they will be in a cycle even if no other transactions are chopped (Figure B.6).

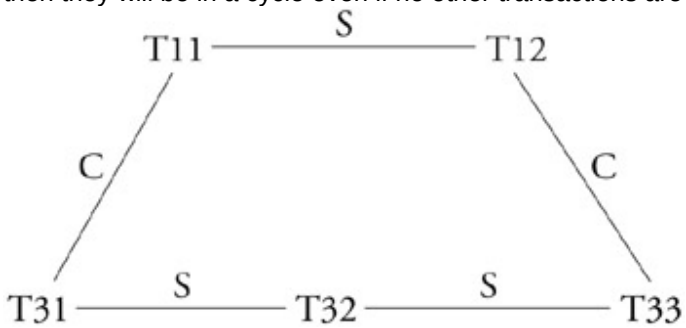


Figure B.6: Putting three pieces of T3 into one will not make chopping of T1 all right, nor will chopping T3 further.

Proof: Since two pieces, say, *p* and *p′*, of *T* are in a cycle, there is an S edge between them and a C edge leading from each of them to pieces of other transactions. If only one piece of some other transaction *T′* is in the cycle, then combining all the pieces of *T′* will not affect the length of the cycle. If several pieces of *T′* are in the cycle, then combining them will simply shorten the cycle.▪

These two lemmas lead directly to a systematic method for chopping transactions as finely as possible. Consider again the set of transactions that can run in this interval $\{T_1, T_2,\ldots, T_n\}$. We will take each transaction T_i in turn. We call $\{c_1, c_2, \ldots, c_k\}$ a *private chopping* of T_i, denoted private(T_i), if both of the following hold:

1. $\{c_1, c_2, \ldots, c_k\}$ is a rollback-safe chopping of T_i.
2. There is no SC-cycle in the graph whose nodes are $\{T_1, \ldots, T_{i-1}, c_1, c_2, \ldots, c_k, T_{i+1}, \ldots, T_n\}$. (If $i = 1$, then the set is $\{c_1, c_2, \ldots, c_k, T_2, \ldots, T_n\}$. If $i = n$, then the set is $\{T_1, \ldots, T_{n-1}, c_1, c_2, \ldots, c_k\}$, i.e., the graph of all other transactions plus the chopping of T_i.)

Theorem 2 The chopping consisting of {*private* (T_1), *private* (T_2) … ,*private* (T_n)} is rollback safe and has no SC-cycles.
Proof:

- *Rollback safe*. The "chopping" is rollback safe because all its constituents are rollback safe.
- *No SC-cycles*. If there were an SC-cycle that involved two pieces of private(T_i) then lemma 2 would imply that the cycle would still be present even if all other transactions were not chopped. But that contradicts the definition of private (T_i).▪

B.4 Optimal Chopping Algorithm

Theorem 2 implies that if we can discover a fine-granularity private (T_i) for each T_i, then we can just take their union. Formally, the *finest chopping*, of T_i, denoted FineChop (T_i) (whose existence we will prove) has two properties:

- FineChop(T_i) is a private chopping of T_i.
- If piece p is a member of FineChop(T_i), then there is no other private chopping of T_i containing p_1 and p_2 such that p_1 and p_2 partition p and neither is empty.

That is, we would have the following algorithm:

```
procedure chop(T_1,...,T_n)
 for each T_i
  Fine_i := finest chopping of T_i
 end for;
the finest chopping is
  {Fine_1, Fine_2,..., Fine_n}
```

We now give an algorithm to find the finest private chopping of T.

```
Algorithm FineChop(T)

initialization:

 if there are rollback statements then
  p_1 := all database writes
   of T that may occur
   before or concurrently with any rollback
   statement in T
 else
```

```
  p_1 := set consisting of
  the first database access;
 end
 P := {{x} | x is a database access not in p_1};
 P := P union {p_1};

 merging pieces:

  construct the connected components
  of the graph induced by C edges alone
  on all transactions besides T
  and on the pieces in P
  update P based on the following rule:

  if p_j and p_k are in the same connected
  component and j <  k, then
   add the accesses from p_k to p_j;
   delete p_k from P
  end if
 call the resulting partition FineChop(T)
```

Note on efficiency. The expensive part of the algorithm is finding the connected components of the graph induced by C on all transactions besides *T* and the pieces in P. We have assumed a naive implementation in which this is recomputed for each transaction *T* at a cost of $O(e + m)$ time in the worst case, where *e* is the number of C edges in the transaction graph and *m* is the size of P. Because there are *n* transactions, the total time is $O(n(e + m))$. Note that the only transactions relevant to FineChop(*T*) are those that are in the same connected component as *T* in the C graph. An improvement in the running time is possible if we avoid the total recomputation of the common parts of the connected components graph for the different transactions.

Note on shared code. Suppose that T_i and T_j result from the same program *P*. Since the chopping is implemented by changing *P*, T_i and T_j must be chopped in the same way. This may seem surprising at first, but in fact the preceding algorithm will give the result that FineChop(T_i) = FineChop(T_j). The reason is that the two transactions are treated symmetrically by the algorithm. When FineChop(T_i) runs, T_j is treated as unchopped and similarly for T_j. Thus, shared code does not change this result at all.

Theorem 3 FineChop(*T*) is the finest chopping of *T*.

Proof: We must prove two things: FineChop(*T*) is a private chopping of *T* and it is the finest one.

- FineChop(*T*) is a private chopping of *T*.

1. *Rollback safety*. This holds by inspection of the algorithm. The initialization step creates a rollback-safe partition. The merging step can only cause p_1 to become larger.
2. *No SC-cycles*. Any such cycle would involve a path through the conflict graph between two distinct pieces from FineChop(*T*). The merging step would have merged any two such pieces to a single one.

- No piece of FineChop(*T*) can be further chopped. Suppose *p* is a piece in FineChop(*T*). Suppose there is a private chopping *TooFine* of *T* that partitions *p* into two

nonempty subsets *q* and *r*. Because *p* contains at least two accesses, the accesses of *q* and *r* could come from two different sources.

1. Piece *p* is the first piece; that is, p_1, and *q* and *r* each contain accesses of p_1 as constructed in the initialization step. In that case, p_1 contains one or more rollback statements. So, one of *q* or *r* may commit before the other rolls back by construction of p_1. This would violate rollback safety.
2. The accesses in *q* and *r* result from the merging step. In that case, there is a path consisting of C edges through the other transactions from *q* to *r*. This implies the existence of an SC-cycle for chopping *TooFine*.▪

B.5 Application to Typical Database Systems

For us, a typical database system will be one running SQL. Our main problem is to figure out what conflicts with what. Because of the existence of bind variables, it will be unclear whether a transaction that updates the account of customer :x will access the same record as a transaction that reads the account of customer :y. So, we will have to be conservative. Still, we can achieve substantial gains.

We can use the tricks of typical predicate locking schemes, however.[5] For example, if two statements on relation Account are both conjunctive (only AND's in the qualification) and one has the predicate

```
AND name LIKE 'T%'
```

whereas the other has the predicate

```
AND name LIKE 'S%'
```

they clearly will not conflict at the logical data item level. (This is the only level that matters because that is the only level that affects the return value to the user.) Detecting the absence of conflicts between two qualifications is the province of compiler writers. We offer nothing new.

The only new idea we have to offer is that we can make use of information in addition to simple conflict information. For example, if there is an update on the Account table with a conjunctive qualification and one of the predicates is

```
AND acctnum = :x
```

then, if acctnum is a key, we know that the update will access at most one record. This will mean that a concurrent reader of the form

```
SELECT ...
FROM Account
WHERE ...
```

will conflict with the update on at most one record, a single data item. We will therefore decorate this conflict edge with the label "1." If the update had not had an equality predicate on a key, then we would decorate this conflict edge with the label "many."
How does this decoration help? Well, suppose that the read and the update are the only two transactions in the system. Then if the label on the conflict is "1," the read can run at degree 2 isolation. This corresponds to breaking up the reader into *n* pieces, where *n* is the number of data items the reader accesses. A cycle in the SC-graph would imply that two pieces of the resulting chopping conflict with the update. This is impossible, however, since the reader and update conflict on only one piece altogether.

B.5.1 Chopping and Snapshot Isolation

Snapshot isolation (SI) is a concurrency control algorithm that never delays read operations, even for read/write transactions. SI has been implemented by Oracle (with certain variations), and it provides an isolation level that avoids many of the common concurrency anomalies. SI does not guarantee serializability, however. Like most protocols that fail to guarantee serializability, SI can lead to arbitrarily serious violations of integrity constraints. This results, we suspect, in thousands of errors per day, some of which may be quite damaging. Fortunately, there are many situations where snapshot isolation does give serializability, and chopping can help identify those situations.[6]

In snapshot isolation, a transaction T1 reads data from the committed state of the database as of time start(T1) (the snapshot), and holds the results of its own writes in local memory store (so if it reads data a second time it will read its own output). Thus, writes performed by other transactions that were active after T1 started are not visible to T1. When T1 is ready to commit, it obeys the first-committer-wins rule: T1 will successfully commit if and only if no other concurrent transaction T2 has already committed writes to data that T1 intends to write. In the Oracle implementation of snapshot isolation, writes also take write locks, and a write is sometimes aborted immediately when a concurrent transaction has already written to that item, but the first-committer-wins certification check is still done at commit time.
Snapshot isolation does not guarantee serializability, however. For example, suppose X and Y are data items representing bank balances for a married couple, with the constraint that $X + Y > 0$ (the bank permits either account to overdraw as long as the sum of the account balances remains positive). Assume that initially $X_0 = 70$ and $Y_0 = 80$. Transaction T_1 reads X_0 and Y_0, then subtracts 100 from X, assuming it is safe because the two data items added up to 150. Transaction T_2 concurrently reads X_0 and Y_0, then subtracts 100 from Y_0, assuming it is safe for the same reason. Each update is safe by itself, but both are not, so no serial execution would allow them both to occur (since the second would be stopped by the constraint). Such an execution may take place on an Oracle DBMS under "set transaction isolation level serializable."

So snapshot isolation is not in general serializable, but observe that if we consider the read portion of each transaction to be a piece and the write portion to be another piece, then the execution on pieces is serializable. The reason is that the reads appear to execute when the transaction begins and the writes are serializable with respect to one another by the first-committer-wins rule. So, consider that decomposition to be the chopping. Snapshot serializability is serializable if the SC-graph on the pieces so constructed is acyclic. If a transaction program can run several times, it is sufficient to consider just two executions from the point of view of discovering whether a cycle is present.

[5]K. C. Wong and M. Edelberg. "Interval Hierarchies and Their Application to Predicate Files." *ACM Transactions on Database Systems*, Vol. 2, No. 3, 223–232, September 1977.

[6]This subsection is drawn from joint work by Alan Fekete, Pat O'Neil, Betty O'Neil, Dimitrios Liarokapis, and Shasha.

B.6 Related Work

There is a rich body of work in the literature on the subject of chopping up transactions or reducing concurrency control constraints, some of which we review here. Such work nearly always proposes a new concurrency control algorithm and often proposes a weakening of isolation guarantees. (There is other work that proposes special-purpose concurrency control algorithms on data structures.)

The algorithm presented here avoids both proposals because it is aimed at database users rather than database implementors. Database users normally cannot change the concurrency control algorithms of the underlying system, but must make do with two-phase locking and its variants. Even if users could change the concurrency control algorithms, they probably should avoid doing so since the bugs that might result could easily corrupt a system.

The literature offers many good ideas, however. Here is a brief summary of some of the major contributions.

1. Abdel Aziz Farrag and M. Tamer Ozsu, "Using Semantic Knowledge of Transactions to Increase Concurrency." *ACM Transactions on Database Systems*, Vol. 14, No. 4, 503–525, December 1989.
Farrag and Ozsu of the University of Alberta consider the possibility of chopping up transactions by using "semantic" knowledge and a new locking mechanism. For example, consider a hotel reservations system that supports a single transaction Reserve. Reserve performs the following two steps:

a. Decrement the number of available rooms or roll back if that number is already 0.
b. Find a free room and allocate it to a guest.

If reservation transactions are the only ones running, then the authors assert that each reservation can be broken up into two transactions, one for each step. Our mechanism might or might not come to the same conclusion, depending on the way the transactions are written. To see this, suppose that the variable *A* represents the number of available rooms, and *r* and *r′* represent distinct rooms. Suppose we can represent two reservation transactions by

T1: RW(A) RW(r)
T2: RW(A) RW(r')

Then the chopping graph resulting from the transactions

T11: RW(A)
T12: RW(r)
T21: RW(A)
T22: RW(r')

will have no cycles. By contrast, if T2 must read room *r* first, then the two steps cannot be made into transactions. That is, dividing

T1: RW(A) RW(r)
T2: RW(A) R(r) RW(r')

into

T11: RW(A)
T12: RW(r)
T21: RW(A)
T22: R(r) RW(r')

will create an SC-cycle as Figure B.7 shows. However, the semantics of hotel reservations tell us that it does not matter if one transaction decrements *A* first but gets room *r′*. The difficulty is applying the semantics. The authors note in conclusion that finding semantically acceptable interleavings is difficult.

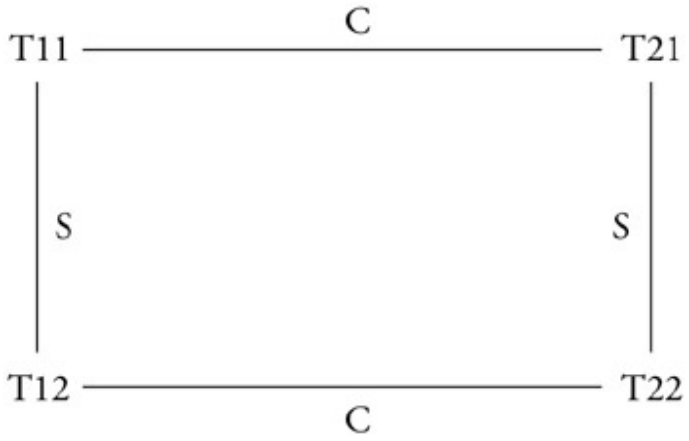


Figure B.7: SC-cycle if T2 must access room *r* before accessing room *r′*.

2. H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database." *ACM Transactions on Database Systems*, Vol. 8, No. 2, 186–213, June 1983.

Hector Garcia-Molina suggested using semantics by partitioning transactions into classes. Transactions in the same class can run concurrently, whereas transactions in different classes must synchronize. He proposes using semantic notions of consistency to allow more concurrency than serializability would allow and using counterstep transactions to undo the effect of transactions that should not have committed.

3. N. Lynch, "Multi-level Atomicity—A New Correctness Criterion for Database Concurrency Control." *ACM Transactions on Database Systems*, Vol. 8, No. 4, 484–502, December 1983.

Nancy Lynch generalized Garcia-Molina's model by making the unit of recovery different from the unit of locking (this is also possible with the check-out/check-in model offered by some object-oriented database systems). She groups transactions into nested classes with specified possible interleavings between them. Then she proposes a new scheduling mechanism that ensures that a specific order among conflict steps is maintained.

4. R. Bayer, "Consistency of Transactions and Random Batch." *ACM Transactions on Database Systems*, Vol. 11, No. 4, 397–404, December 1986.

Rudolf Bayer of the Technische Universitat of Munich showed how to change the concurrency control and recovery subsystems to allow a single batch transaction to run concurrently with many short transactions.

5. M. Hsu and A. Chan, "Partitioned Two-Phase Locking." *ACM Transactions on Database Systems*, Vol. 11, No. 4, 431–446, December 1986.

Meichun Hsu and Arvola Chan have examined special concurrency control algorithms for situations in which data is divided into raw data and derived data. The idea is that the consistency of the raw data is not so important in many applications, so updates to that data should be able to proceed without being blocked by reads of that data. For example, suppose there are three transactions with the following read/write patterns:

T1: W(x)

T2: R(x) W(y)

T3: R(x) R(y) W(z)

Consider chopping each of these transactions into single accesses. There is an SC-cycle in such a chopping: R3(x) W1(x) R2(x) W2(y) R3(y). If reads only accessed current values, this would correspond to a cycle in the serialization graph. Their algorithm by contrast would allow the R3(y) to see the old state of y.

Some commercial systems such as Oracle use this scheme as well as allowing reads to view old data. That facility would remove the necessity to use the algorithms in this paper for read-only transactions.

6. P. O'Neil, "The Escrow Transactional Mechanism." *ACM Transactions on Database Systems*, Vol. 11, No. 4, 405–430, December 1986.

Patrick O'Neil takes advantage of the commutativity of increments to release locks early even in the case of writes.

7. O. Wolfson, "The Virtues of Locking by Symbolic Names." *Journal of Algorithms*, Vol. 8, 536–556, 1987.

Ouri Wolfson presents an algorithm for releasing certain locks early without violating serializability.

8. M. Yannakakis, "A Theory of Safe Locking Policies in Database Systems." *Journal of the ACM*, Vol. 29, No. 3, 718–740, 1982.

Yannakakis assumes that the user has complete control over the acquisition and release of locks. The setting here is a special case: the user can control only how to chop up a transaction or whether to allow reads to give up their locks immediately. As mentioned earlier, we have restricted the user's control in this way for the simple pragmatic reason that systems restrict the user's control in the same way.

9. P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, "Concurrency Control in a System for Distributed Databases (SDD-1)." *ACM Transactions on Database Systems*, Vol. 5, No. 1, 18–51, March 1980.

Bernstein, Shipman, and Rothnie introduced the idea of conflict graphs in an experimental system called SDD-1 in the late 1970s. Their system divided transactions into classes such that transactions within a class executed serially, whereas transactions between classes could execute without any synchronization. Their assumed concurrency control algorithm was based on locks and timestamps.

10. Marco Casanova, *The Concurrency Control Problem for Database Systems*. Springer-Verlag Lecture Notes in Computer Science, No. 116, 1981.

Marco Casanova's thesis extended the SDD-1 work by representing each transaction by its flowchart and by generalizing the notion of conflict. A cycle in his graphs indicated the need for synchronization if it included both conflict and flow edges.

11. D. Shasha and M. Snir, "Efficient and Correct Execution of Parallel Programs that Share Memory." *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 2, 282–312, April 1988.

Shasha and Snir explored graphs that combine conflict, program flow, and atomicity constraints in a study of the correct execution of parallel shared memory programs that have critical sections. The graphs used here are a special case of the ones used in that article.

Ongoing work related to transaction chopping has been done by Bruno Agarwal, Arthur Bernstein, Alan Fekete, Wenway Hseush, Dimitrios Liarokapis, Ling Liu, Elizabeth O'Neil, Pat O'Neil, Calton Pu, Lihchyun Shu, Michal Young, and Tong Zhou.

Bibliography

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. *A critique of ANSI SQL isolation levels.* In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 22–25, 1–10. ACM Press, 1995.

A. Fekete. *Serialisability and snapshot isolation*. In *Proceedings of the Australian Database Conference*, Auckland, New Zealand, 201–210, 1999.

Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. *Transaction chopping: Algorithms and performance studies*. *TODS*, 20(3):325–363, 1995.

Appendix C: Time Series, Especially for Finance

Overview

This section addresses several subjects.

- How time series are used in business and finance.
- What systems (FAME, S-Plus, SAS, KSQL) that are specialized for array processing do to support them.
- A brief discussion of what the relational vendors are offering, focusing on Oracle 8i. (The reason we are brief is that we view these offerings as evolving.)
- An annotated bibliography in time series for computer scientists.

Since much of this work was motivated by Shasha's consulting on Wall Street, we start with a trading example to serve as motivation. An analytical group discovers the desirability of "pairs trading." The goal is to identify pairs (or, in general, groups) of stocks whose prices track one another after factoring in dividends.
You can make money (lots has been made) because, for example, you may know that the two banks Chase and Citibank track each other (their difference is a stationary process). Now if, on a certain day, Chase goes up but Citibank doesn't, then buy Citibank and sell Chase. This is simplified: the model often involves a whole collection of stocks, and externalities such as dividends and business news must be taken into account. Typical *challenge queries* from such an application include the following:

- Correlate the price histories of two stocks or in general among many stocks and options. (For most traders, returns are more interesting than prices because they have better statistics: a stock that trends up over the years has an unstationary mean, but perhaps a stationary return. So, you perform correlations over "returns." The return at day t is $ln(\text{price}(t)/\text{price}(t-1))$.
- Perform the correlation over certain time intervals to evaluate the stationarity.
- The correlation might be weighted: recent history counts more than distant history.

C.1 Setting Up a Time Series Database

The raw data comes in the form of ticks (stock, quantity, price) and can be stored in a relational database without a problem. Time series are difficult because the relational model does not take advantage of the order of rows. Whereas you can perform an order by query and manipulate the data in some other language, you cannot natively manipulate the ordered data using select, from, and where. Arguably, this is good for data independence, but it is bad for time series queries. Realizing this, the traders curse a lot and tell their programmers to cobble something together. The programmers do so and create a piece of software that is part spreadsheet and part special-purpose database, with lots of C++ code. Employment goes up.
Joe Celko shows how to bend SQL to the task of simulating order in his popular and excellent book, *The SQL Puzzle Book*, published by Morgan Kaufmann. Usually, the bending results in a loss of efficiency. It also works only for special cases. Fortunately, many relational vendors include time series as a special data type along with a set of functions to manipulate them. Our discussion at a semantic level applies to such relational extensions as well as to special-purpose array databases.
In order to explain the semantics, we return to the basic question: what are time series? In finance, marketing, and other applications, time series are a sequence of values usually recorded at or aggregated to regular intervals (yearly, monthly, weekly, …, secondly). For example, a stock market ticker tape will have the ticks of buys, sells, and offers. The ticker will mix different stocks, but each stock's stream will be in strictly increasing time order. The price for an interval of interest will be the latest price from that interval, for example, the last price of

the day. Regularity ensures that moving averages, autocorrelations, and other time-related statistics make sense.

Nonregular time series are also of interest (e.g., in finance, the history of stock splits), but mostly as adjustments to the regular time series. For example, if a stock has been split 2 for 1, then all prices preceding the split will be divided in half in order to be "split adjusted."
To be useful, time series should also exhibit *historicity*: the past is an indicator of the future. That is why autoregression can be used to predict the future of sales and why the past volatility may predict future volatility, to cite two examples.

To prepare data for time series queries, the data must be regularized to a certain frequency. Unfortunately, the data may not have values for every time instance at the stated frequency. For example, business day has the frequency of a day but has no values on holidays or weekends. Further, one country's holidays may not be the same as another's. For example, some Asian stock markets are open on Saturdays. The basic solution is to store values without gaps everywhere (i.e., every day). That brings up the question of how to fill in the gaps. Time series values are of two general types (we borrow this distinction from the FAME system).

- *Level* values stay the same from one period to the next in the absence of activity. For example, inventory is a level value because inventory stays the same if you neither buy nor sell.
- *Flow* values are zero in the absence of activity. For example, expenses go to zero if you buy nothing.

The type clearly determines the value before interpolation. Interpolation is used to smooth curves. Typically, systems use various spline techniques such as a cubic spline to interpolate missing values though more complicated methods can be used, such as the Black-Derman-Toy interpolation of the yield curve.[1]

[1]The yield curve is the annualized interest that you would get if you lent money for various time durations. For example, you might get *x*% annualized interest if you lent money for 5 days, *y*% for one month, *z*% for 100 years. Normally, these numbers increase with the time. Given such values, the Black-Derman-Toy interpolation will give imputed values for, say, 17 days. It will then go on compute option values for interest rate options.

C.2 FAME

FAME is a special-purpose system that supports time series. FAME stands for forecasting, analysis, and modeling environment and comes from FAME information systems, Ann Arbor, Michigan (http://www.fame.com). Typical FAME operations include

- cumulative sum (e.g., year-to-date sales).
- moving averages (e.g., 30-day average of stock prices).
- nth best (e.g., fifth best sales region).
- median (one in the middle).
- rank (associate ordinal to each value based on its sort order).
- discretize (e.g., rank the revenues by whether they are in the top third, the middle third, or the bottom third). This implies discovering the boundaries and then using them in an update query.
- year-to-year comparisons (e.g., balance of trade of this year versus last).
- accounting functions (e.g., average growth rate, amortization, internal rate of return).
- statistical functions (e.g., autocorrelation and correlation between two series).
- forecasting functions (e.g., autoregression).[2]

In options finance, the basic approach to forecasting is to assume that the price of an equity is based on a random walk (Brownian motion) around a basic slope. The magnitude of the

randomness is called the volatility. In a result due to Norbert Wiener (he worked it out to shoot down bombers over London), for this model, the standard deviation of the difference between the initial price and the price at a certain time *t* rises as the square root of time *t*.

For concreteness, here are the steps in a typical FAME session.

- Specify frequency; say, monthly, starting at January 1, 1996, and ending at the current time.
- Create sales and expense time series by importing these from a file or typing them in. Specify that these are flow-type time series.
- Create a new time series: formula profit = sales − expenses.
- Create a fourth time series with weekly frequency on inventory. Specify that inventory is a level-type time series.
- Convert the first three time series to a weekly frequency (by dividing the monthly values by 4.2 or by constructing a cubic spline to make the sales, expenses, and profits curve look smooth). This interpolation depends on knowing that sales and expenses are flow-type values.
- Now, use autoregression to predict future time series values.

FAME also has a relational frontend called the FAME Relational Gateway, developed by Thinkbank.

[2]Yule invented the autoregressive technique in 1927 so that he could predict the annual number of sunspots. This was a linear model, and the basic approach was to assume a linear underlying process modified by noise. That model is often used in marketing (e.g., what will my sales of wheat be next month?). There are also seasonal autoregressive models.

C.3 S-Plus

S-Plus is an interpretive environment for data analysis, not specifically oriented toward time series, but based on vectors (http://www.mathsoft.com/splus/). S-Plus is derived from the S language developed at AT&T Bell Laboratories by Becker, Chambers, and Wilkens, but development now belongs to MathSoft Inc.

S-Plus has

- standard statistical and mathematical functions, including anova, wavelets, bootstrapping to check for model overfitting, and so on. These are the mathematical tools used in futures and options trading.
- graphics capabilities for visualization (user testimonials say this is a particularly strong point).
- combinatorial data mining (e.g., inference of classification trees and regression).
- an object-oriented language allowing encapsulation and overloading. For example, objects of a certain class will have a special plot function.

The S-Plus programming model is vector oriented. Here are some typical statements.

- *sum*(*age* < *mean*(*age*)) gives the count of people who are younger than the mean.
- *agecat* ← *if else*(*age* < 16, 'Young', 'Old') assigns to the vector agecat a sequence of Young/Old values thus discretizing the data.
- *f-sum* (*age, subset* = *sex* = = *'female'*) finds the sum of the ages of all females.

So, you can create vectors, do statistics on them, and perform selection-style operations on them. The S-Plus implementation is also vector oriented. S-Plus is slow for large databases, especially for data that exceeds RAM size. Below RAM size, it is very fast. For this reason, S-Plus is often used with SAS because SAS has data management capabilities.

C.4 SAS

SAS (http://www.sas.com) is a leading vendor for statistical databases. (Originally, it stood for Statistical Analysis System, but now the acronym stands for itself.) An SAS programmer interacts with the system by parametrizing various functions, as the following example shows:

```
proc forecast data=leadprd
  ar=1          /* number of autoregressive parameters
                   to estimate                    */
  interval=month /* frequency of input time series      */
  trend=1       /* fit a constant trend model          */
  method=stepar  /* use stepwise autoregressive method   */
  out=leadout1   /* create output data set for forecasts  */
  lead=12       /* number of forecast periods          */
  outlimit
  outstd;
  var leadprod;
  id date;      /* identification variable             */
  run;
```

In addition, SAS has an integrated SQL dialect called Proc SQL. SAS has modules for data mining and data warehousing as well. To obtain support for time series data management in SAS, you purchase a library called ETS that enables you to do

- interpolation
- econometric forecasting (e.g., maximum likelihood method)
- financial analysis (analysis of fixed-rate mortgages, adjustable-rate mortgages, etc.)
- time series forecasting (exponential smoothing, ARIMA, dynamic regression)

S-Plus is more flexible for special-purpose problems and is fast for problems that fit into RAM. It also has great graphics. Combining the two works well if the application selects a subset of data and then works on it (like a loose-coupling expert system).

C.5 KDB

KDB is a database system implemented on top of the K language environment (produced by Kx Systems: http://www.kx.com), an array language. Data structures (e.g., tables) can be interchanged between the two, and functions can be called in both directions. A free trial version can be downloaded.[3]

KDB supports an SQL dialect called KSQL. KSQL is easy to learn (for anyone fluent in SQL) and carries over the speed and functionality of K to large data manipulation. KDB also supports most of standard SQL as well as some extensions.
The basic data structure in KSQL is the *arrable* (array table), which is a table whose order can be exploited. In this sense, it is similar to S-Plus. Arrables are non-first-normal-form objects: a field of a record can be an array. For example, an entire time series can be stored in a field.

Like most modern SQLs, KSQL allows the inclusion of user-defined functions inside database statements. Unlike other SQLs, KSQL allows functions to be defined over arrays as well as scalars. Like classical SQL, KSQL has aggregates, grouping, selections and string matching, and so on.

KSQL adds many useful functions to SQL, permitting economical expression and often better performance by exploiting order. Here are a few annotated examples on a table ordered by date whose schema is trade(stock, date, price).

select last price, 5 avgs price by stock, date.month from trade

- performs a group by stock and year-month in the trade table. Mathematically, the by clause partitions the records based on distinct stock/year-month values, like a group by in SQL. What is different is that each partition is guaranteed to be ordered in the same way as in the table (arrable). For each partition corresponding to stock s and year-month x, the "last price" part will return s, x, and p, where p is the price of the last record in the partition corresponding to stock s and date x. Because the table is ordered by date, the last record will be the one with the most recent date. The 5 avgs price uses the avgs function, which given a vector (price for each stock in this case), returns a vector of the same length, computing a moving average. Five avgs computes the five-day moving average. (In KSQL, "avg" returns the overall average as in SQL, whereas "avgs" returns a vector.)
- If the user specifies a function that is not in the vocabulary of KSQL, then the function (autocorrelate in the following example) can easily be enclosed into the query:

select autocorrelate[10,price] by stock from trade

uses the autocorrelate function to compute the 10-day delayed autocorrelation of prices grouped by stock.

The largest KDB application to date encompasses several billion trades and quotes on 25 Linux Intel boxes—all U.S. equities for the last few years. Most queries return in seconds.

[3]Disclosure note: Shasha uses K and KDB on a daily basis and has made a small contribution to the design of KDB, so he's biased.

C.6 Oracle 8i Time Series

The Oracle 8i Time Series product makes the distinction between regular and irregular data as follows:

- Regular data—data that arrives at fixed intervals such as sensor data or end-of-day stock data. Regular data can be associated with a calendar having a variety of frequencies such as second, minute, hour, day, week, 10-day, semimonth, month, quarter, semiannual, and year. The calendar feature allows the data to be checked for consistency.
- Irregular data—data that arrives at variable intervals, such as news of shipwrecks.

The data itself is stored either as a separate object or as a plain relational table. It is stored in timestamp order so functions such as moving average can be easily implemented.

Typical functions that are supported include

- FirstN—get the first N of a time series.
- TrimSeries—get the time series between certain dates.
- TSMaxN—top N elements of the time series.
- Moving averages in a window or cumulative averages to date. This also works for max, min, product, and sum.
- Scaling functions to move from, say, days to weeks, or vice versa.

C.7 Features You Want for Time Series

Whether you use one of the preceding systems or another, you would do well to ensure that the following features, extending those mentioned in Chapter 6, are available:

- The ability to treat sequences as first-class objects on which you can do useful operations within the database system.
- The ability to treat multiple sequences together for correlations and other purposes.
- A basic collection of functions, including aggregates, moving aggregates, statistics, cross-correlations, interpolation, and so on.
- The ability to integrate user-defined functions into the query engine. It is our view that user-defined functions are essential for time series. The reason is that there is no analog to relational completeness that will satisfy all (or even most) time series applications.
- The main helpful database amenities, including a rich relational vocabulary and the ability to work efficiently with disk- as well as RAM-resident data.
- Special time-related functions, such as the ability to group by month or year based on timestamps.
- Interpolation functions. This requires that values be treated appropriately, as in the level and flow concepts of FAME.

C.8 Time Series Data Mining

Financial traders have done data mining for many years. One trader described his work as follows: "I think about an arbitrage trick (pairs trading is such a trick). Program for a few months. Try the trick and either it works or it doesn't. If it doesn't, I try something new. If it works, I enjoy it until the arbitrage disappears." What does the research community have to offer to such traders? Here are some references:

1. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. AAAI Press/The MIT Press, 1996.
   The article by Berndt and Clifford about finding patterns in time series is particularly relevant to finance.
2. Opher Etzion, Sushil Jajodia, and Sury Sripada, *Temporal Databases—Research and Practice*. Springer-Verlag, 1998.
   In this book, you will find articles about finding unexpected patterns (e.g., fraud) and multigranularity data mining.
3. Christos Faloutsos, *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, 1996.
   This book shows how to do signal processing analysis on time series to solve problems.

Other papers explore the question of similarity search when time scaling and inversion are possible.

1. R. Agrawal, K-I Lin, H. S. Sawhney, and K. Shim, "Fast Similarity Search in the Presence of Noise, Scaling and Translation in Time-Series Databases." *Proceedings of the Twenty-First VLDB Conference*, 1995.
2. D. Q. Goldin and P. C. Kanellakis, "On Similarity Queries for Time-Series Data: Constraint Specification and Implementation." *First International Conference on the Principles and Practice of Constraint Programming.* Springer-Verlag, 1995.
3. Davood Rafiei and Alberto Mendelzon, "Similarity-Based Queries for Time Series Data." *Proceedings of the ACM Sigmod Conference*, May 1997.
4. Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos, "Efficient Retrieval of Similar Time Sequences Under Time Warping." *Proceedings of the Fourteenth International Conference on Data Engineering*, 1998.

As an alternative to seeing whether two sequences or subsequences match, you might want to describe a desirable sequence (e.g., a head-and-shoulders movement of stock prices) to see whether it is present. Relevant papers about this include

1. H. V. Jagadish, A. O. Mendelzon, and T. Milo, "Similarity-Based Queries." *Proceedings of the PODS Conference*, 1995.
2. R. Agrawal, G. Psaila, E. L. Wimmers, and M. Zait, "Querying Shapes of Histories." *Proceedings of the Twenty-First VLDB Conference*, 1995.
3. P. Seshadri, M. Livny, and R. Ramakrishnan, "Sequence Query Processing." *Proceedings of the ACM SIGMOD Conference*, 1994.
   Data model and query language for sequences in general, with time series as a special case.
4. Arie Shoshani and Kyoji Kawagoe, "Temporal Data Management." *Proceedings of the VLDB Conference*, 1986.
   One of the first papers in the literature.
5. Snodgrass, R. T., *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.
   The TSQL2 Language Design Committee consisted of Richard Snodgrass (chair), Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Kaefer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. TSQL2 has time-varying aggregates, including moving window aggregates, aggregates over different time granularities, and weighted over time.
6. Munir Cochinwala and John Bradley, "A Multidatabase System for Tracking and Retrieval of Financial Data." *Proceedings of the VLDB Conference*, 1994.
   A paper discussing the implementation of a tick capture and query system—for those brave enough to roll their own.
7. Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Keven S. Beyer, and Muralidhar Krishnaprasad, "SRQL: Sorted Relational Query Language." *Proceedings of the SSDBM Conference*, 1998.
   A paper discussing a model in which relations are tables that can be ordered. This allows you to do moving averages, find the 10 cheapest, the preceding 15, and so on. The strategy is to extend SQL with order and special operators.

Some books on time series are appropriate for computer scientists.

1. C. Chatfield, *The Analysis of Time Series: Theory and Practice*. Chapman and Hall, fourth edition, 1984.
   Good general introduction, especially for those completely new to time series.
2. P. J. Brockwell and R. A. Davis, *Time Series: Theory and Methods*. Springer Series in Statistics, 1986.
3. B. D. Ripley and W. N. Venables, *Modern Applied Statistics with S-Plus*. Springer, 1994.
   Chapter 14 *has a good discussion of time series. In addition, you can find useful functions at* http://www.stats.ox.ac.uk/ripley/.
   FinTime is a time series benchmark tailored for finance. The description of the queries and data generation tools are available at http://cs.nyu.edu/cs/faculty/shasha/fintime.html.

Appendix D: Understanding Access Plans

Overview

—Alberto Lerner, *DBA, Doctoral Candidate*

DBMSs do not execute SQL queries in the order they are written. Rather, they first capture a query's meaning, translate it into a corresponding *access plan*, and then execute the set of operations contained in this plan in what you hope is an optimal way given the data structures available. So when you ask something like select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY—show me all the customers' names and their corresponding nations—to an orders database,[1] DB2 would execute instead a plan like the one in Figure D.1. Other DBMSs' explainers would present similar plans though the display would differ a little.

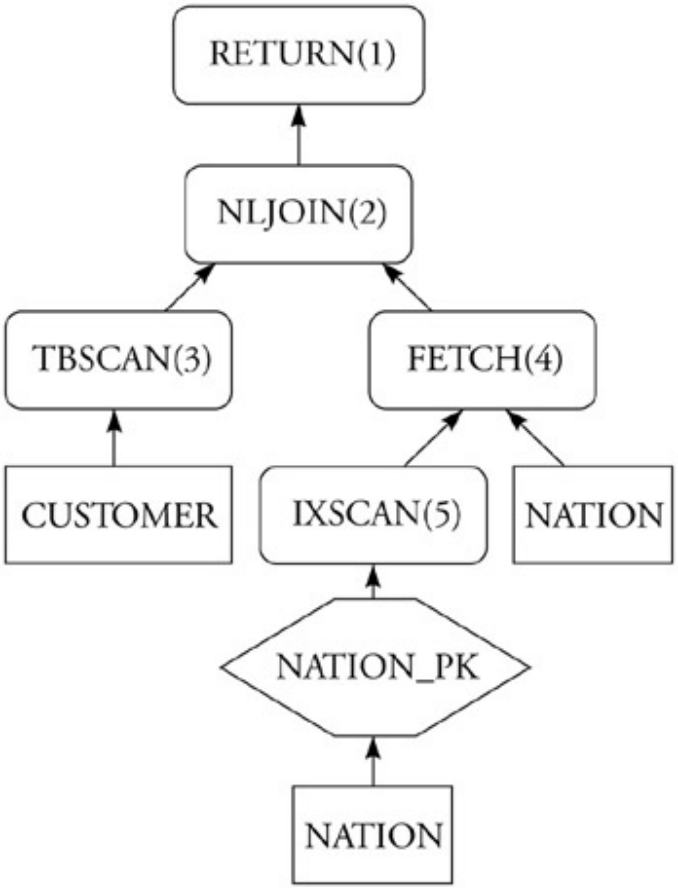


Figure D.1: Query access plan obtained using DB2's Visual Explain for the query select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY. The query's answer is produced by a nested-loops join in which CUSTOMER is the outer table and NATION is the inner one. A table scan operation reads the rows of the former, while the nonclustered index NATION_PK is used to retrieve the latter.

Plans are usually depicted as upward-pointing trees in which sources at the bottom are tables or indexes (squares and diamonds, respectively, in Figure D.1), and internal nodes are operators (octagons). Each operator represents a particular transformation, an intermediate stage in the production of a resulting node.

A plan in fact denotes a complex assembly (or production) line—complex because it's an inverted tree rather than a line. "Raw" rows enter the production line via the operators

connected to the sources, follow the transformation path determined by the arrows, and eventually become resulting rows output by the target node on the top.

Executing a plan means asking its target operator to produce results, in fact, to produce the next row of the result. That triggers a cascading effect that puts the entire assembly line to work in an elegantly synchronized way. In our example, the target operator (RETURN(1)) upon receiving a "produce-next-row" command asks the nested-loops join (NLJOIN(2)) to produce the next row itself. Similarly, the join propagates the command to its left branch, eventually obtaining a CUSTOMER's fragment—just the columns it needs—of a row. It repeats the process on the right branch and obtains the corresponding NATION's fragment row. Finally, the two fragments are combined and sent back to the RETURN(1) operator. The assembly line is ready to restart the process and to produce a subsequent row—until all the resulting rows are produced.

If you understand this mechanism, you understand how plans work and ultimately how queries are answered. It remains only to know the transformations behind operators better and their relation to SQL constructs. Fortunately, only a few types of transformation, and therefore operators, are involved in processing SQL queries. Let us describe the main types in turn and see them in action.

[1]The schema used in this and in the following examples was drawn from the TPC-H benchmark database, and the data was generated with their DBGEN tool (http://www.tpc.org).

D.1 Data Access Operators

These operators read data sources in order to feed rows into the assembly line. Table scans and index scans are the main operators in this class. They can transform incoming rows in two ways. First, they can output a subset of the columns of the rows. For instance, the operator TBSCAN(3) retrieves only three of the CUSTOMER's columns, as indicated by its detail information window in Figure D.2. Second, they can output rows that satisfy a simple predicate, filtering out undesired rows. The operator IXSCAN(5) selects a particular nation key, the value being processed at NLJOIN(2), and its corresponding row ID, or RID, as Figure D.3 shows. The FETCH(4) operator, which is itself a data access operator, can then use the RID to retrieve the N_NAME column of the selected nation.

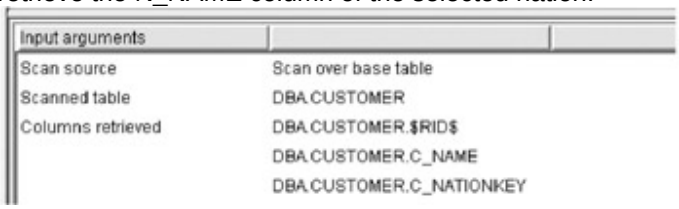


Figure D.2: Input arguments for the operator TBSCAN(3) of the query in Figure D.1. The operator retrieves only three columns from the table, RID, C_NAME, and C_NATIONKEY. No other column will be needed to process the rest of this query.

Input arguments		
Access path	DBA.NATION_PK	
Scanned table	DBA.NATION	
Columns retrieved	DBA.NATION.RID	
	DBA.NATION.N_NATIONKEY	
Sargable predicates	None	
Start predicates	Column	Column
	Number	Name
	0	DBA.NATION.N_NATIONKEY
Stop predicates	Column	Column
	Number	Name
	0	DBA.NATION.N_NATIONKEY

Figure D.3: Input arguments for the operator IXSCAN(5) of the query in Figure D.1. The operator filters out all keys that are different from the key being joined by the NLJOIN(2) operation. Table scans and index scans plus fetches perform equivalent sets of transformations. There is a widespread, and wrong, belief that index scans can do them cheaper. That is not always the case. Suppose we augmented our CUSTOMER-join-NATION query with a predicate over the customer account balance such as…where C_ACCTBAL > 0. Should a DBMS pick a nonclustered index on C_ACCTBAL (ACCTBAL_IX)? That depends on our customer's allowed credit. If the predicate's selectivity were small, it should pick that index. A good plan for that case appears in Figure D.4. Instead of scanning through the 6764 CUSTOMER table pages, the precise ones that contain selected customers are found by IXSCAN(4). However, in our case 136,308 out of 150,000 customers have positive balances. Such a plan would cause more pages to be read than the actual table has! (Remember that ACCTBAL_IX is not clustered, so if *n* selected keys point to a same table page, that page can potentially be read *n* times.) A table scan like the one in Figure D.1 augmented with a filtering predicate inside TBSCAN(3) is better. Table D.1 shows the results of running the positive balance query using both strategies.[2] The table scan plan reads CUSTOMER's pages only once, no matter how many customers were selected, and reads NATION's pages once for every selected customer. This is where 143,072 logical data reads come from (6764 pages +,136,308 selected customers = 143,072). The index scan plan reads a CUSTOMER's page for every selected customer, increasing the logical data reads to 272,616 (136,308 + 136,308 = 272,616), let alone a similar number of index accesses. Needless to say, the response time suffers. Moral: indexes are not a panacea.

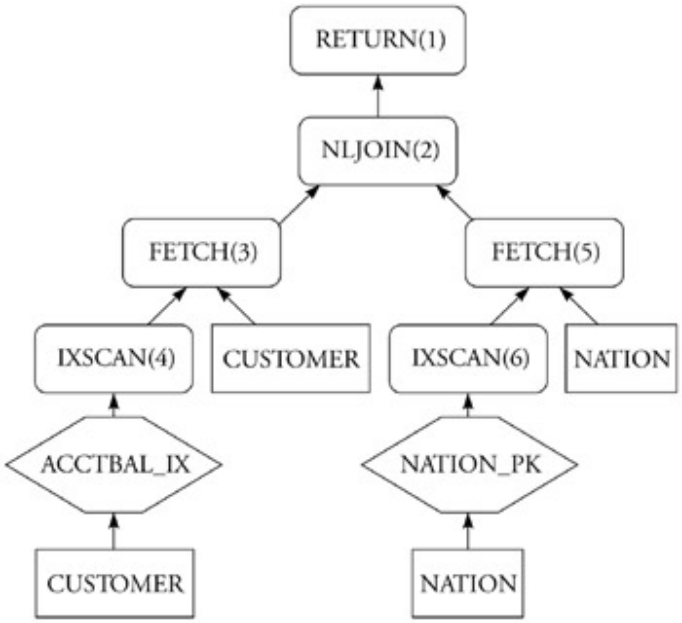


Figure D.4: Query access plan for the query select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY where C_ACCTBAL > 0. The addition of the balance predicate over the query of Figure D.1 can make the use of the index ACCTBAL_IX cost-effective, or not. It will all depend on the predicate's selectivity.

Table D.1: Partial perfomance indicators resulted by executing the query select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY N_NATIONKEY where C_ACCTBAL > 0 with a plan where the CUSTOMER table is scanned and where it is accessed indirectly via an index scan.

	TABLE SCAN	INDEX SCAN
CPU time (avg)	5 sec	76 sec
Data logical reads	143,075	272,618
Data physical reads	6766	131,425
Index logical reads	136,319	273,173
Index physical reads	7	552
Rows selected	136,308 rows	
CUSTOMER table size	6764 pages	
ACCTBAL_IX index size	598 pages	
NATION table size	1 page	
NATION_PK index size	1 page	

Nevertheless, sometimes it is advantageous to use even more than one index to access a single table. Two additional data access operators make it possible: the RID scan and the index AND. Take our original join query and attach two predicates, … C_ACCTBAL > 9900

and … C_MKTSEGMENT = 'AUTOMOBILE'. Consider two indexes, the account balance one used before, and a new nonclustered one over the market segment column, MKTSEGMENT_IX. If the predicates were *OR-ed* then the plan in Figure D.5 would be produced. The transformation performed by the RIDSCN operator is the merging and duplicate elimination of the key/RIDs lists generated by the IXSCANs and SORTs. The rest of the plan works like the pure join one. If, on the other hand, the predicates were *AND-ed*, then both indexes would still be used, but in the way the plan fragment of Figure D.6 shows. The IXAND operator outputs key/row ID pairs that occur in all indexes being scanned.

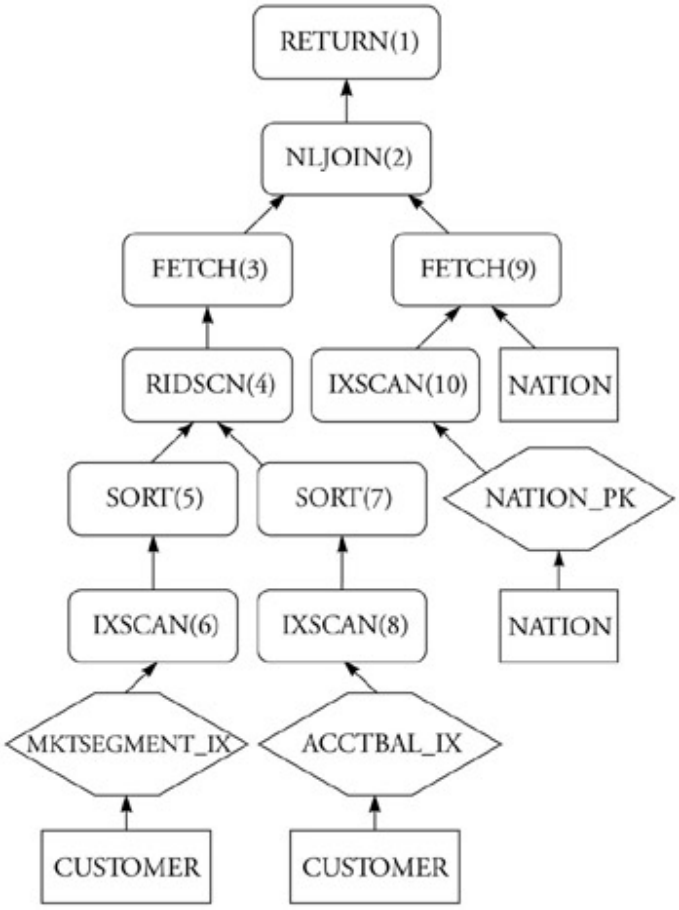


Figure D.5: Query access plan for the query select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY where C_ACCTBAL > 9900 *or* C_MKTSEGMENT = 'AUTOMOBILE'. The operation RIDSCN(4) permits taking advantage of the fact that each predicate of the query is covered by a distinct index.

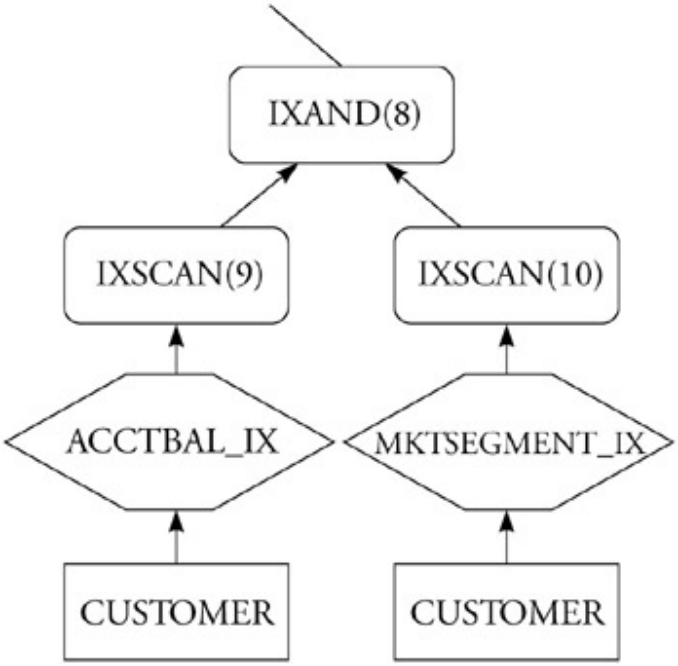


Figure D.6: Fragment of the access plan for the query select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY where C_ACCTBAL > 9900 *and* C_MKTSEGMENT = 'AUTOMOBILE'. The operation IXAND(8) permits merging the two keys/RIDS list and thus taking advantage of more than one index to access the CUSTOMER table.

[2]Given the necessary statistics, an optimizer would pick the correct plan. Here, it was tricked into generating the index scan plan by carefully manipulating the distribution statistics.

D.2 Query Structure Operators

Some operators exist to describe the transformations of specific SQL constructs. The three most common operators in this class are the ones that perform the relational join, each implementing a particular join algorithm.[3] The nested-loops join (NLJOIN) scans the outer relation and looks up each of its rows in the inner relation. The sort-merge join (MSJOIN) scans both relations simultaneously after making sure they are both sorted over the join predicate columns. The hash join (HSJOIN) builds a hash table using the outer relation and probes the inner relation against it. There is no algorithm that wins every time as illustrated in Chapter 3. DBMSs use data distribution and tables' physical storage information to decide which operator to use. Sometimes DBMSs can perform counterintuitive things, whenever the costs invested are realized in terms of using the best join strategy. For instance, take a look at the plan shown in Figure D.7. You might be surprised to hear that there is no ORDER BY clause or any other aspect in its query that justifies the existence of a sort over CUSTOMER. The SORT(4) operation is over C_NATIONKEY—the join column—and was put there to permit the use of the MSJOIN(2). It would not be worth using the ACCTBAL_IX index to retrieve 13,692 customers, far more than the number of pages in that table. On the other hand, that number of rows could be sorted in memory. A sort in memory plus a sort-merge join in that case is a better choice than the nested-loops join plan shown in the positive balance version of the query (Figure D.1). Table D.2 compares the costs of the "natural" merge-sort plan to that of a "forced" nested-loops one for the same negative balance query. The difference in terms of logical reads is remarkable. Had this experiment been done while concurrent queries competed for memory, that is, when many more of the repetitive logical reads of the nested loops would mean physical reads, we would have seen a noticeable gain in the sort version.

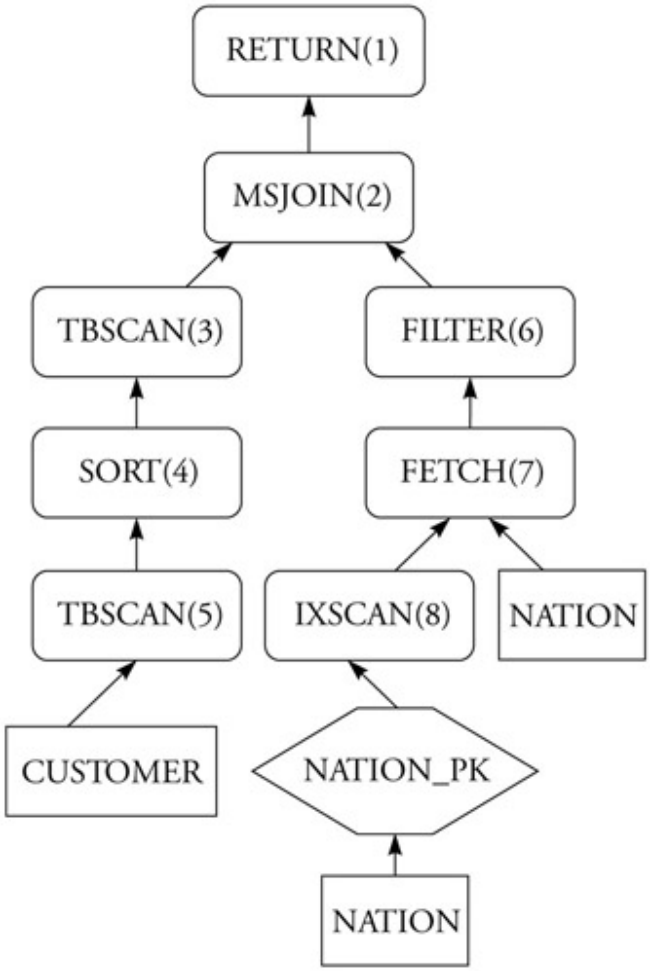


Figure D.7: Query access plan for the query select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY where C_ACCTBAL < 0. The join algorithm chosen was the sort-merge that requires both the inner and outer table to be sorted over the join column. The table CUSTOMER needs an explicit sort, whereas NATION can take advantage of the NATION_PK index.

Table D.2: Performance indicators resulting from executing the query select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY where C_ACCTBAL < 0 with a plan where a nested-loops algorithm was "forced" onto the optimizer instead of the "natural" sort-merge—based plan.

	NESTED LOOPS	SORT-MERGE
CPU time (avg)	2.7 sec	3.1 sec
Data logical reads	20,459	6768
Data physical reads	6766	6767
Index logical reads	13,703	12
Index physical reads	7	7

Table D.2: Performance indicators resulting from executing the query select C_NAME, N_NAME from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY where C_ACCTBAL < 0 with a plan where a nested-loops algorithm was "forced" onto the optimizer instead of the "natural" sort-merge—based plan.

	NESTED LOOPS	SORT-MERGE
Rows selected	13,692 rows	
CUSTOMER table size	6764 pages	
NATION table size	1 page	
NATION_PK index size	1 page	

At this point you should realize how vital accurate statistics are in query optimization decisions. DBMSs must also decide the order in which to perform joins in multiway join queries. (Join is a binary operator; therefore, multiway joins are processed in pairs.) If more that two tables are involved in a join, does it make any difference in which order the joins are processed? Let's see. Suppose we wanted to check which nations had customers with pending orders and when these orders were posted—select C_NAME, N_NAME, O_ORDERDATE from CUSTOMER join NATION on C_ NATIONKEY=N_NATIONKEY join ORDERS on C_CUSTKEY=O_CUSTKEY where O_ORDERSTATUS='P'. There are two possibilities: (CUSTOMER join ORDERS) join NATION or (CUSTOMER join NATION) join ORDERS. Either way, the result has 38,543 rows. CUSTOMER join ORDERS with the status restriction generates 38,543 rows. CUSTOMER join NATION generates 150,000. The latter approach would unnecessarily produce a larger intermediate result. The optimizer chooses the plan shown in Figure D.8.

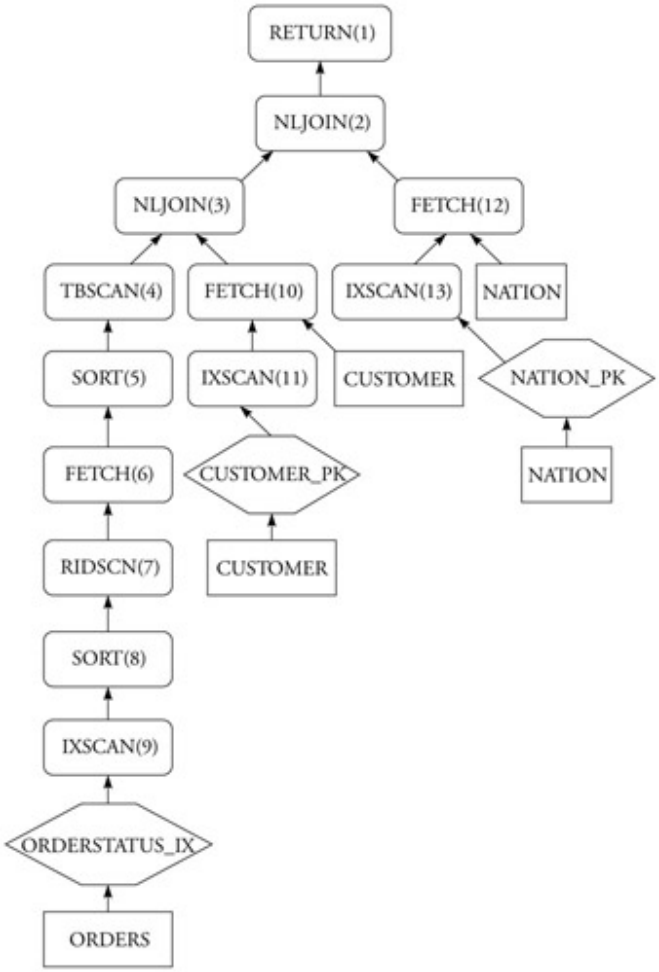


Figure D.8: Query access plan for the query select C_NAME, N_NAME, O_ORDERDATE from CUSTOMER join NATION on C_NATIONKEY = N_NATIONKEY join ORDERS on C_CUSTKEY = O_CUSTKEY where O_ORDERSTATUS = 'P'. The tables ORDERS and CUSTOMER get joined first, and only then do the results get joined with table NATION.

Other examples of operators in this class are the group by (GRPBY) that appears in queries that perform some sort of aggregation (count(), min(), etc.) or have the GROUP BY construct itself; and the UNION operator that merges tuples coming from distinct input streams and appears whenever the = SQL construct after which it is named appears.

[3]A good discussion of the implementation and efficiency of these algorithms can be found in *Database Management Systems* by Raghu Ramakrishnan and Johannes Gehrke, McGraw-Hill, June 2000.

D.3 Auxiliary Operators

The SORT operator is the most common operator in this class. Like other auxiliary operators, it appears either as a result of the use of simpler SQL constructs like ORDER BYs or to prepare input rows for other operators, as was the case of the sort-merge join in Figure D.7.

Bibliography

Goetz Graefe. *Query evaluation techniques for large databases. ACM Computing Surveys*, 25(2), June 1993. This survey is a good starting point for the tuner who wants to understand the algorithms involved in the execution of access plans.

IBM. *Administration Guide V7.1*, Volume 3: Performance (SC09-2945-00). This manual describes how to use DB2's explainer tools and how to interpret their output.

Appendix E: Configuration Parameters

Overview

Tuning parameters described in this book are initialization parameters set for each database at creation time. Depending on the system, these parameters can then be modified or not; possibly these modifications take effect only after the server has been shut down and restarted.

Here is a short list of the main configuration parameters.

- Log file size—should be big enough to hold all updates between dumps.
- Buffer size—should be as big as possible without spilling over to swap space.
- Block size (unit of transfer between disk and memory)—bigger for scan-intensive applications. Should be smaller for online transaction processing.
- Log buffer size—unit of transfer to the log. Should be big enough to allow group commit.
- Group commit size.
- Prefetching size—should be large enough to make scanning efficient—roughly the size of a track.
- Checkpoint interval—a balance between warm start recovery and overhead.
- Degree of parallelism—if too large, blocking can occur; if too small, there can be insufficient parallelism.

As an example, we present the configuration parameters used for TPC-C and TPC-H benchmarks with the three systems we are focusing on.[1]

[1]Results of the TPC benchmarks are published at http://www.tpc.org. This Web site also contains the specifications of these benchmarks.

E.1 Oracle

Here are the configuration parameters used for the Sun Starfire Enterprise 10000 with Oracle 9i (on Solaris 8) TPC-H benchmark published in April 2001. The initialization parameters are regrouped in a file, usually init.ora, that can be modified directly—new parameters take effect when the server is restarted. *Oracle Performance Tuning—Tips and Techniques* by Richard Niemiec (Oracle Press, 1999), gives a description of the main initialization parameters for Oracle 8i.

```
# init.ora
# excerpt from TPC-H full disclosure report
# Sun Starfire Enterprise 10000 with Oracle9i
# http://www.tpc.org/tpch/results/h-result1.idc?id = 101041701


## Database name (multiple instances are defined: instance and
## service names are initialized on a node-specific file)
db_name                         = inst1


## Max number of user processes that connect to Oracle
```

```
processes                    = 1024


## Size of the buffer cache
db_block_buffers             = 1222020
## Block Size
db_block_size                = 8192
## Prefetching: Number of blocks per I/O operation
db_file_multiblock_read_count    = 64
## Prefetching: Number of sequential blocks per I/O in a hash
## join
hash_multiblock_io_count         = 32
## Number of buffers that can be dirty in the buffer cache
## (0 means that there are no limits on the number of dirty
## buffers)
db_block_max_dirty_target        = 0
## Maximum number of I/O that should be needed during recovery
## (0 means that there are no limits)
fast_start_io_target             = 0
## Maximum number of data files
db_files                     = 1023
## Initial number of database writer processes for an instance
db_writer_processes              = 10
## Backward compatibility with older release
compatible                   = 8.1.7
## Location of the Oracle control files (run-time
## configuration)
control_files                = /dev/vx/rdsk/photon3/cntrl_1


## Size in bytes of the memory pool allocated for stored
## procedures, control structures,...
shared_pool_size             = 150000000
## Size in bytes of the memory pool allocated for message
## buffer
## (parallel execution)
large_pool_size              = 2598790772
## Size in bytes of the memory pool allocated for Java run-time
## (0 means that no memory is allocated for Java run-time)
java_pool_size               = 0
```

```
## maximum amount of memory, in bytes, for sorting operations
sort_area_size                  = 30000000
## Maximum amount of memory, in bytes, for hash joins
hash_area_size                  = 70000000


## Maximum number of DML locks (i.e., table lock obtained
## during insert, update or delete if this number is exceeded)
dml_locks                       = 60000
## Number of resources that can be concurrently locked
enqueue_resources               = 50000


## Maximum number of open cursors per session
open_cursors                    = 1024
## Number of transactions accessing each rollback segment
transaction_per_rollback_segment   = 1


## PARALLEL EXECUTION PARAMETERS
## Specifies that parameters should take a default value
## suited for parallel execution
parallel_automatic_tuning       = TRUE
parallel_server                 = TRUE
parallel_adaptive_multi_user    = TRUE
parallel_execution_message_size    = 8192
parallel_max_servers            = 2000
parallel_min_servers            = 600


## DISTRIBUTED LOCK MANAGER
## (1) Mapping of parallel cache management (PCM) locks to file
## 2-350 is the set of data files
## 10 is the number of PCM locks assigned for these files
## EACH means that each file is assigned a separate set of
## locks
gc_files_to_locks               = "2-350=10EACH"
## (2) Number of distributed locks available for simultaneously
## modified rollback blocks.
## 0-400 is the set of rollback segments
```

```
## 32 is the number of locks
## !8 is the number of blocs covered by one lock
## R indicates that these locks are releasable
## EACH indicates that each rollback segment is assigned a
##   separate set of locks
gc_rollback_locks               = "0-400=32!8REACH"


## No check on database link name
global_names                    = FALSE


## Maximum size of trace files
max_dump_file_size              = 100000
## Maximum number of rollback segments
max_rollback_segments           = 512
## Optimizer options
optimizer_features_enable       = 8.1.7.1
optimizer_index_cost_adj        = 300
optimizer_mode                  = CHOOSE
optimizer_percent_parallel      = 100


## Join methods
always_anti_join                = HASH
always_semi_join                = HASH


## Specifies that partitioned views are allowed
partition_view_enabled          = TRUE


## No augmented redo records or auditing records are generated
transaction_auditing            = FALSE
audit_trail                     = FALSE


## Parameters not documented
lm_ress                         = (70000, 70000)
lm_locks                        = (70000, 70000)
```

```
## Specifies that checkpoints should be logged on the alert
## file
log_checkpoints_to_alert          = TRUE


## No parallel propagation of operations on the replicated
## tables
replication_dependency_tracking    = FALSE
recovery_parallelism               = 64


## Maximum number of distributed transactions (0 means that
## no distributed transaction is allowed - in addition the
## recovery process resolving errors due to distributed
## transactions does not start when the instance starts up.)
distributed_transactions           = 0


## format used for the TODATE() function
nls_date_format                    = YYYY-MM-DD
```

E.2 SQL Server

Here are the configuration parameters used for the Compaq ProLiant ML570-3P with SQL Server 2000 Standard Edition (on Windows 2000) TPC-H benchmark published in July 2001. The configuration parameters are managed by SQL Server. They can be displayed, and some of them can be changed dynamically using a system stored procedure. The *Inside SQL Server* book series by Soukup and Delaney details the configuration parameters related to performance.

NAME	MIN	MAX	CONFIG_VALUE	RUN_VALUE
affinity mask	0	2147483647	7	7
allow updates	0	1	0	0
awe enabled	0	1	1	1
c2 audit mode	0	1	0	0
cost threshold for parallelism	0	32767	5	5
cursor threshold	0	2147483647	−1	−1
default full-text language	0	2147483647	1033	1033
default language	0	9399	0	0

fill factor (%)	0	100	0	0
index create memory (KB)	704	2147483647	0	0
lightweight pooling	0	1	0	0
locks	5000	2147483647	0	0
max degree of parallelism	0	32	1	1
max server memory (MB)	4	2147483647	1800	1800
max text repl size (B)	0	2147483647	65536	65536
max worker threads	32	2147483647	350	350
media retention	0	365	0	0
min memory per query (KB)	32	2147483647	1800	1800
nested triggers	0	1	1	1
network packet size (B)	512	65536	4096	4096
open objects	0	2147483647	0	0
priority boost	0	1	1	1
query governor cost limit	0	2147483647	0	0
query wait (s)	−1	2147483647	−1	−1
recovery internal (min)	0	32767	35	335
remote access	0	1	1	1
remote login timeout (s)	0	2147483647	20	20
remote proc trans	0	1	0	0
remote query timeout (s)	0	2147483647	600	600
scan for start-up process	0	1	0	0
set working set size	0	1	1	1
show advanced options	0	1	1	1
two digit year cutoff	1753	9999	2049	2049
user connections	0	32767	360	360

E.3 DB2 UDB

Here are the configuration parameters used for the SGI 1450 using IBM DB2 UDB V7.2 (on Linux) TPC-H benchmark submitted for review in May 2001. Consultants we know start with these parameter settings when configuring their own systems.

Node type = Partitioned Database Server with local and remote clients
Database manager configuration release level = 0x0900
CPU speed (millisec/instruction) (CPUSPEED) = 8.738368e-07
Communications bandwidth (MB/sec) (COMM_BANDWIDTH) = 1.000000e+00
Max number of concurrently active databases (NUMDB) = 1
Data Links support (DATALINKS) = NO
Federated Database System Support (FEDERATED) = NO
Transaction processor monitor name (TP_MON_NAME) =
Default charge-back account (DFT_ACCOUNT_STR) =
Java Development Kit 1.1 installation path (JDK11_PATH) =
Diagnostic error capture level (DIAGLEVEL) = 3
Diagnostic data directory path (DIAGPATH) = /home/db2_local_diaglog
Default database monitor switches Buffer pool (DFT_MON_BUFPOOL) = OFF
Lock (DFT_MON_LOCK) = OFF
Sort (DFT_MON_SORT) = OFF
Statement (DFT_MON_STMT) = OFF
Table (DFT_MON_TABLE) = OFF
Unit of work (DFT_MON_UOW) = OFF
SYSADM group name (SYSADM_GROUP) =
SYSCTRL group name (SYSCTRL_GROUP) =
SYSMAINT group name (SYSMAINT_GROUP) =
Database manager authentication (AUTHENTICATION) = SERVER
Cataloging allowed without authority (CATALOG_NOAUTH) = NO
Trust all clients (TRUST_ALLCLNTS) = YES
Trusted client authentication (TRUST_CLNTAUTH) = CLIENT
Default database path (DFTDBPATH) = /home/tpch
Database monitor heap size (4KB) (MON_HEAP_SZ) = 56
UDF shared memory set size (4KB) (UDF_MEM_SZ) = 256
Java Virtual Machine heap size (4KB) (JAVA_HEAP_SZ) = 512
Audit buffer size (4KB) (AUDIT_BUF_SZ) = 0
Backup buffer default size (4KB) (BACKBUFSZ) = 1024
Restore buffer default size (4KB) (RESTBUFSZ) = 1024

```
 Sort heap threshold (4KB) (SHEAPTHRES) = 273692
 Directory cache support (DIR_CACHE) = YES
 Application support layer heap size (4KB) (ASLHEAPSZ) = 15
 Max requester I/O block size (bytes) (RQRIOBLK) = 32767
 Query heap size (4KB) (QUERY_HEAP_SZ) = 1000
 DRDA services heap size (4KB) (DRDA_HEAP_SZ) = 128
 Priority of agents (AGENTPRI) = SYSTEM
 Agent pool size (NUM_POOLAGENTS) = 0
 Initial number of agents in pool (NUM_INITAGENTS) = 0
 Max number of coordinating agents (MAX_COORDAGENTS) =
   (MAXAGENTS - NUM_INITAGENTS)
 Max no. of concurrent coordinating agents (MAXCAGENTS) =
   MAX_COORDAGENTS
 Max number of logical agents (MAX_LOGICAGENTS) = MAX_COORDAGENTS
 Keep DARI process (KEEPDARI) = YES
 Max number of DARI processes (MAXDARI) = MAX_COORDAGENTS
 Initialize DARI process with JVM (INITDARI_JVM) = NO
 Initial number of fenced DARI processes (NUM_INITDARIS) = 0
 Index re-creation time (INDEXREC) = RESTART
 Transaction manager database name (TM_DATABASE) = 1ST_CONN
 Transaction resync interval (sec) (RESYNC_INTERVAL) = 180
 SPM name (SPM_NAME) =
 SPM log size (SPM_LOG_FILE_SZ) = 256
 SPM resync agent limit (SPM_MAX_RESYNC) = 20
 SPM log path (SPM_LOG_PATH) = TCP/IP
 Service name (SVCENAME) = tpch
 APPC Transaction program name (TPNAME) =
   IPX/SPX File server name (FILESERVER) =
   IPX/SPX DB2 server object name (OBJECTNAME) =
   IPX/SPX Socket number (IPX_SOCKET) = 879E
 Discovery mode (DISCOVER) = SEARCH
 Discovery communication protocols (DISCOVER_COMM) =
 Discover server instance (DISCOVER_INST) = ENABLE
 Directory services type (DIR_TYPE) = NONE
 Directory path name (DIR_PATH_NAME) = /.:/subsys/database/
 Directory object name (DIR_OBJ_NAME) =
 Routing information object name (ROUTE_OBJ_NAME) =
 Default client comm. protocols (DFT_CLIENT_COMM) =
 Maximum query degree of parallelism (MAX_QUERYDEGREE) = ANY
 Enable intra-partition parallelism (INTRA_PARALLEL) = YES
```

No. of int. communication buffers(4KB)(FCM_NUM_BUFFERS) = 4096
Number of FCM request blocks (FCM_NUM_RQB) = 4096
Number of FCM connection entries (FCM_NUM_CONNECT) =
(FCM_NUM_RQB * 0.75)
Number of FCM message anchors (FCM_NUM_ANCHORS) =
(FCM_NUM_RQB * 0.75)
Node connection elapse time (sec) (CONN_ELAPSE) = 10
Max number of node connection retries (MAX_CONNRETRIES) = 5
Max time difference between nodes (min) (MAX_TIME_DIFF) = 1440
db2start/db2stop timeout (min) (START_STOP_TIME) = 10

Database configuration release level = 0x0900
Database release level = 0x0900
Database territory = US
Database code page = 819
Database code set = ISO8859-1
Database country code = 1
Dynamic SQL Query management (DYN_QUERY_MGMT) = DISABLE
Directory object name (DIR_OBJ_NAME) =
Discovery support for this database (DISCOVER_DB) = ENABLE
Default query optimization class (DFT_QUERYOPT) = 7
Degree of parallelism (DFT_DEGREE) = 10
Continue upon arithmetic exceptions (DFT_SQLMATHWARN) = NO
Default refresh age (DFT_REFRESH_AGE) = 0
Number of frequent values retained (NUM_FREQVALUES) = 0
Number of quantiles retained (NUM_QUANTILES) = 300
Backup pending = NO
Database is consistent = YES
Rollforward pending = NO
Restore pending = NO
Multi-page file allocation enabled = NO
Log retain for recovery status = NO
User exit for logging status = NO
Data Links Token Expiry Interval (sec) (DL_EXPINT) = 60
Data Links Number of Copies (DL_NUM_COPIES) = 1
Data Links Time after Drop (days) (DL_TIME_DROP) = 1
Data Links Token in Uppercase (DL_UPPER) = NO
Data Links Token Algorithm (DL_TOKEN) = MAC0
Database heap (4KB) (DBHEAP) = 6654
Catalog cache size (4KB) (CATALOGCACHE_SZ) = 64

Log buffer size (4KB) (LOGBUFSZ) = 128
Utilities heap size (4KB) (UTIL_HEAP_SZ) = 10000
Buffer pool size (pages) (BUFFPAGE) = 39170
Extended storage segments size (4KB) (ESTORE_SEG_SZ) = 16000
Number of extended storage segments (NUM_ESTORE_SEGS) = 0
Max storage for lock list (4KB) (LOCKLIST) = 40000
Max appl. control heap size (4KB) (APP_CTL_HEAP_SZ) = 384
Sort list heap (4KB) (SORTHEAP) = 6842
SQL statement heap (4KB) (STMTHEAP) = 4096
Default application heap (4KB) (APPLHEAPSZ) = 200
Package cache size (4KB) (PCKCACHESZ) = (MAXAPPLS*8)
Statistics heap size (4KB) (STAT_HEAP_SZ) = 4384
Interval for checking deadlock (ms) (DLCHKTIME) = 10000
Percent. of lock lists per application (MAXLOCKS) = 20
Lock timeout (sec) (LOCKTIMEOUT) = -1
Changed pages threshold (CHNGPGS_THRESH) = 60
Number of asynchronous page cleaners (NUM_IOCLEANERS) = 30
Number of I/O servers (NUM_IOSERVERS) = 30
Index sort flag (INDEXSORT) = YES
Sequential detect flag (SEQDETECT) = YES
Default prefetch size (pages) (DFT_PREFETCH_SZ) = 32
Track modified pages (TRACKMOD) = OFF
Default number of containers = 1
Default tablespace extentsize (pages) (DFT_EXTENT_SZ) = 32
Max number of active applications (MAXAPPLS) = 80
Average number of active applications (AVG_APPLS) = 1
Max DB files open per application (MAXFILOP) = 128
Log file size (4KB) (LOGFILSIZ) = 14080
Number of primary log files (LOGPRIMARY) = 25
Number of secondary log files (LOGSECOND) = 30
Changed path to log files (NEWLOGPATH) =
Path to log files =
/home/filesystems/disk1/tpch_data/tpch/NODE0000/SQL00001/
SQLOGDIR/
First active log file =
Group commit count (MINCOMMIT) = 1
Percent log file reclaimed before soft chckpt (SOFTMAX) = 500
Log retain for recovery enabled (LOGRETAIN) = OFF
User exit for logging enabled (USEREXIT) = OFF
Auto restart enabled (AUTORESTART) = ON

Index re-creation time (INDEXREC) = SYSTEM (RESTART)
Default number of loadrec sessions (DFT_LOADREC_SES) = 1
Number of database backups to retain (NUM_DB_BACKUPS) = 12
Recovery history retention (days) (REC_HIS_RETENTN) = 366
TSM management class (TSM_MGMTCLASS) =
TSM node name (TSM_NODENAME) =
TSM owner (TSM_OWNER) =
TSM password (TSM_PASSWORD) =

Glossary

Symbol and Numbers

→

Same as functionally determines (functional dependencies are introduced in Chapter 4). A → B if any two rows having the same A value also have the same B value.

2-3 tree

B-tree whose maximum branching factor is 3 and minimum branching factor is 2. A tree with a small branching factor such as a 2-3 tree or a T-tree should be used for small data structures that remain in random access memory (Chapter 3).

4GL

Same as fourth-generation language (Chapter 5). These languages often generate very poor SQL.

A

abort

Every database transaction completes either by committing or aborting. When a transaction aborts, all its writes are undone by the recovery subsystem. A transaction abort is often caused by a deadlock (Chapter 2).

access path

Strategy used to access the rows of a table. The most common alternatives are a direct access (i.e., a table scan) or an access through an index (i.e., an index scan) (Chapter 7 and Appendix D).

after image

The after image of a data item *x* with respect to transaction *T* is the value of *x* that *T* last writes (Chapter 2).

aggregate maintenance

Maintenance of one or more redundant relations that embody an aggregate. For example, if the sum of sales by store is frequently requested, then it may be worthwhile to maintain a relation containing that sum and then to update it whenever a sale occurs (Chapters 4 and 10).

aggregate targeting

A technique for using the data in a data warehouse in which aggregates are computed as an input to business rules that then "target" individual customers (Chapter 10).

application servers

The middle tier in the e-commerce architecture. Often implemented in Java, these embody business logic (Chapter 8).

arrable

An array table. A table whose order can be exploited by a query language (e.g., by doing a moving average over a column) (Appendix C).

arrival rate

Number of requests that arrive in a given time interval.

atomicity guarantees

Database theorists use atomicity to denote the indivisibility of transactions. That is, a transaction should *appear* to execute in isolation (without interleaving with other transactions) and in an all-or-nothing manner (either all its effects are registered with the database or none are) (Chapter 2).

attribute

Name of the head of a column in a table.

B

B-tree

The most used data structure in database systems. A B-tree is a balanced tree structure that permits fast access for a wide variety of queries. In virtually all database systems, the actual structure is a B+ tree in which all key-pointer pairs are at the leaves (Chapter 3).

batch transaction

Transaction that performs many (possibly millions of) updates, normally without stringent response time constraints.

before image
The before image of a data item *x* with respect to transaction *T* is the value of *x* when *T* first reads it (Chapter 2).

bind variable
Variable defined in some embedding programming language (usually, COBOL, PL/1, RPG, C_) that is used for communication with the database system. For example, if x is a bind variable, then a typical database query might include the expression R.A = @x, where x has been previously assigned a value in the embedding programming language.

bitmap
Index structure used for WHERE clauses having many unselective conditions (e.g., on sex, race, country quadrant, hair color, and height category). Bitmaps work well in such situations because intersecting bit vectors is an extremely fast operation (Chapter 9).

bottleneck
System resource or query that limits the performance of an entire database application; breaking a bottleneck usually entails speeding up one or more queries or using partitioning.

branching factor
The average number of children of each nonleaf node in a B-tree or other tree structure. The larger the branching factor, the fewer the levels. Compression is a technique for increasing the branching factor (Chapter 3).

broadcast technology network
Network medium sharing strategy in which only one sender can be using the network at any time. Ethernet is the most prominent example of such a protocol.

buffer
See database buffer (Chapter 2).

buffered commit strategy
Logging and recovery algorithm in which a transaction's updates are not forced to the database disks immediately after commit, but rather written when the write would cause no seek. This is almost always a good strategy to choose (Chapter 2).

C

capacity planning
The activity of building a system that is sufficiently powerful to handle all traffic within desired response time limits. This involves three components: estimating demand (an exercise in market analysis), an estimate of the probabilities of flow between services (perhaps by looking at industry standards), and a measurement of service times of each server (Chapter 8).

case-based reasoning
The process of describing a situation as attribute-value pairs and looking up "similar" cases from a database. Similarity depends on metrics that grade the relative importance of different attributes. For example, sex may be more important than income level for marketing in certain kinds of magazines. (Chapter 10).

catalog
Place where data type and statistical information is held in a database management system. For example, the catalog may hold information about the number of records in a relation, the indexes on its attributes, and the data type of each attribute. It is important not to modify the catalog too often during busy periods (e.g., by creating temporary tables). Otherwise, the catalog becomes a lock contention bottleneck (Chapters 2 and 3).

checkpoint
Activity of backing up the log onto the database disks. This reduces the time to recover from a failure of main memory, but costs something in online performance (Chapters 2 and 7).

chopping
See transaction chopping.

class
A data description and set of operations that characterizes some collection of objects. For example, an image class may define a data description consisting of a bitmap and a set of

operations, including zooming, rotation, and so on. This is an important concept in object-oriented and object-relational systems.

cluster key
Attribute(s) that determine a table clustering in Oracle. For example, Customer and Sale may be clustered based on the customer identifier in which case the Customer record with that identifier will be colocated with all Sales records having that identifier (Chapter 4).

clustering
A technique for grouping data based on some notion of similarity. Many clustering techniques are possible depending on the goals (e.g., the number of clusters, the largest difference within a cluster, and so on). The practitioner should look for a clear separation before distinguising one cluster from another (Chapter 10).

clustering index
Index structure that dictates table organization. For example, if there is a clustering index based on a B-tree on last name, then all records with the same last name will be stored on consecutive pages of the table. Because a table cannot be organized in two different ways, there can only be one clustering index per table (Chapter 3).

colocate
Putting several objects or records close to one another on disk. For example, a clustering index based on a B-tree would tend to colocate records having lexicographically close keys (e.g., the records containing Smith and the records containing Smithe). By contrast, a hash-based clustering index will colocate the Smith records but not necessarily the Smiths with the Smithes (Chapter 3).

commit
Every database transaction completes either by committing or aborting. When a transaction commits, the database management system guarantees that the updates of that transaction will be reflected in the database state even if there are failures (provided they are failures that the database system can tolerate such as main memory failures) (Chapter 2).

composite index
Index whose key is a sequence of attributes (e.g., last name, first name, city) (Chapter 3).

compression
Same as key compression (Chapter 3).

concatenated index
Same as composite index (Chapter 3).

concurrency control
Activity of synchronizing the database accesses of concurrent transactions to ensure some degree of isolation. Weaker degrees of isolation can result in higher performance but may corrupt the data (Chapter 2 and Appendix B).

concurrent
Activities (in this book, transactions) A and B are concurrent if there is some point in time *t* where both have begun and neither has completed (Chapter 2).

connection pooling
A general client-server technique for multiplexing a large number of application server connections over a fixed number of database connections (Chapter 8). This reduces the overhead needed to obtain such a connection.

covering index
An index on table T covers a query if the processing of the query requires access to the index only rather than the rows of T. For example, if the employee table has a composite index (department, name), then the query SELECT name FROM employee WHERE department = ‘information’ would be covered by that index (Chapter 3).

cursor stability
Assurance given by a database management system's concurrency control algorithm that while a transaction holds a cursor, no other transaction will modify the data associated with that cursor (Chapter 2).

customer relationship management
Using data about a customer to improve an enterprise's relationship with that customer. For example, if an air traveler has encountered many delays, the airline may send him or her free

travel vouchers. Such systems embody rules (like the one about vouchers). Those rules are derived from data mining usually on a large data warehouse (Chapters 9 and 10).

cylinder

Cylinder *i* is the set of all track *i*'s for a given disk spindle. Because all the disk heads will be on the same track of all platters, reading track *i* on one platter after reading track *i* on another requires no head movement (i.e., no seek). Therefore, it is good to keep frequently scanned data on the same cylinder. This argues for large extents (Chapter 2).

D

data item

Unit of locking when discussing concurrency control and a unit of logging when discussing the recovery subsystem. Common data items are records, pages, and tables in relational systems; objects, pages, and class extents in object-oriented systems (Chapter 2).

data mining

The activity of finding useful patterns in data. An example is that customers who have several accounts at a bank are unlikely to switch banks. This might suggest a customer relationship management rule of giving incentives to customers to acquire more accounts (Chapters 9 and 10).

data page

Page of a file containing records; by contrast, a data structure page contains keys and pointers (Chapters 3 and 4).

data warehouse

Repository for a large amount of data, often historical. Changes to the data warehouse are usually pure inserts in time order. Most queries have an online analytical processing (OLAP) character (Chapters 9 and 10).

database buffer

Repository of databases pages in virtual memory (though, in a well-tuned system, the whole buffer should be held in random access memory). Ideally, most database reads will be to pages held in the buffer, thus eliminating the need for a disk access. Database writes can be stored in the buffer and then written asynchronously to the database disks, thus avoiding seeks, in a strategy known as buffered commit (Chapter 2).

database cache

In an e-commerce application, the database cache stores tables or materialized views on the middle tier. The application server avoids a round-trip to the database server every time it accesses data in the database cache (Chapter 8). The database buffer is also called database cache.

database disks

Stable storage is divided into two parts: the database disks and the log. The log contains the updates of committed transactions, and the database disk contains the initial database state (or value of last database dump) modified by some of those committed transactions (Chapter 2).

database dump

State of the database at some point in time. The current state of the database is equal to the value of the last database dump plus all the writes of transactions committed since then (Chapter 2).

database state

At any time, the (logical) database state is the result of applying all committed transactions (in the order of commitment) to the last database dump. The database state can be physically reconstructed from the database disks and the log or, in case of disk failure, from the database dump and the log (Chapter 2).

DDL

Data description language. The language used to manipulate catalog data (e.g., create table and create index statement). It is usually a bad idea to execute DDL updates during online transaction processing because the catalog may then become a locking hot spot (Chapters 2 and 4).

deadlock

Property of a set of transactions in which each transaction waits for another in the set, so none can proceed. Two-phase locking allows deadlock, unless users design transactions specifically to avoid it (Chapter 2 and Appendix A).

decision tree
A tree in which the edges are labeled and the edges descending from each interior node are mutually exclusive. The leaf nodes are labeled with a "decision." For example, a decision tree may start with a node labeled age and the edges leaving may be labeled with 17–25, 26–38, 39–50, 51–65, 66–100. The target node of the edge labeled 17–25 may concern rock music, whereas the target node of the edge labeled 51–65 may concern second homes. Decision tree building techniques are usually based on statistical and information theoretic techniques (Chapter 10).
degree 0 isolation
Write locks held while writes occur, no read locks. So, reads can access dirty data and writes can overwrite other transactions' dirty writes (Chapter 2).
degree 1 isolation
Write locks acquired in two-phase manner, no read locks. So, reads can access dirty data, but writes cannot overwrite other transactions' dirty writes (Chapter 2).
degree 2 isolation
Write locks acquired in two-phase manner, read locks held while reads occur. So, reads cannot access dirty data, but reads are not repeatable (Chapter 2).
degree 3 isolation
Read and write locks acquired in a two-phase locking manner. Degree 3 isolation is the assurance given by a concurrency control algorithm that in a concurrent execution of a set of transactions that may contain reads and updates but no inserts or deletes, whichever transactions commit will appear to execute one at a time. Lesser degrees give lesser guarantees (Chapter 2). There is a higher level called serializable that ensures isolation even in the case of inserts and deletes.
denormalization
Activity of changing a schema so relations do not enjoy third normal form for the purpose of improving performance (usually by reducing the number of joins). Should not be used for relations that change often or in cases where disk space is scarce (Chapter 4). Data warehouses sometimes use denormalization, but aggregate maintenance is normally a better policy for them.
dense index
Index in which the underlying data structure has a pointer to each record among the data pages. Clustering indexes can be dense in some systems (e.g., DB2). Nonclustering indexes are always dense (Chapter 3).
determines
See functionally determines (Chapter 4).
dirty data
Data item *x* is dirty at time *t* if it has been modified by some transaction that has not yet committed (Chapter 2).
dirty read
Read of data item *x* written by an uncommitted transaction (WR conflict) (Chapter 2).
disk (with movable head)
Device consisting of one or more spindles, each of which consists of a set of platters stacked one on top of the other with a head positioned at the same track of each platter (except the top and bottom platters). Only one head can read or write at a time (Chapter 2). Reading a disk sequentially is at least ten times faster than reading randomly.
disk head
Device for reading or writing data on a disk. It is held by an arm that moves the head from one track to another along a single platter (Chapter 2). Such movement is called a seek.
disk queues
Internal operating system structures that store disk requests not yet serviced (Chapter 7).
durable media
Media, such as disks and tapes, that will not lose data in the event of a power failure. Random access memory can be made durable if it has battery backup (but then the length of this durability is limited by the energy storage capacity of the battery) (Chapter 2).

E

e-commerce

Business conducted over a digital network, usually the Web. Implementations often use a three-tier architecture. In this book, we don't distinguish between end-user e-commerce and business-to-business e-commerce (Chapter 8).

encapsulation
Facility of a language whereby the internal data of an object can be accessed only through its methods (i.e., its operations). In this way, the internal data structures of the object can be changed without functionally affecting users of that object. Object-relational systems rely on encapsulation for user-defined type definitions (Chapter 4).

entity
Notion from the entity-relationship model. Entities denote the objects of interest (e.g., employee, hospital, doctor). An entity has attributes that it must functionally determine. An attribute must not have attributes of its own (otherwise, it should be promoted to entity status). Entity-relationship diagrams are often used in table design (Chapter 4).

equality join query
Join in which the join predicate is equality. For example, R.A = S.B is an equality join, whereas R.A > S.B is not (Chapters 3 and 4).

equality selection
Clause of the form R.A = 5, that is, a relation-attribute pair compared by equality to a constant, constant expression, or bind variable (Chapter 3).

equivalent executions
Two executions *E* and *F* of transactions are equivalent if they consist of the same transactions and the same database accesses, every database read returns the same value in *E* and *F*, and the final writes on each data item are the same (Chapter 2).

escalation
See lock escalation (Chapter 2).

exclusive lock
See write lock (Chapter 2).

execution site independence
Ability to move computation between server site and client site without changing the input/output behavior of a computation. This is useful for low contention/high computation activities and is provided in many commercial object-relational database systems (Chapter 4).

extent
Contiguous area on disk used as a unit of file allocation (Chapter 2). Because it is contiguous, accessing an extent requires few seeks and thus is fast.

extremal query
Query that obtains the records or parts of records with the minimum or maximum of a set of values. For example, the following query finds the maximum salary in the employee relation: SELECT MAX(salary) FROM employee (Chapter 3).

F

fail-stop failure
Failure in which a processor stops. The failure may destroy the processor's random access memory but not its disk storage. Such failures characterize hardware failures of machines that have redundant error-detecting circuitry, perhaps in the form of an extra processor. Software failures may corrupt data and therefore are not fail-stop (Chapter 2).

fanout
See branching factor (Chapter 3).

federated data warehouse
A data warehouse constructed on top of a set of independent databases (Chapter 10).

field
See attribute.

finer-grained locking
A-level locking is said to be finer grained than *B*-level locking if *A* is a smaller unit of storage than *B*. For example, if records are smaller than pages, then record-level locking is finer grained than page-level locking (Chapter 2).

foreign key

Relationship between two tables in a relational system. The best way to define it is by example. Suppose table SP has fields supplier_ID, part_ID, quantity, and there is another relation Supplier with information about supplier. Then Supplier.supplier_ID is a primary key for Supplier and a foreign key for SP.supplier_ID. Every supplier_ID value in SP is in some record of Supplier (Chapter 4).

fourth-generation language
Language used mostly for data entry and report production that makes calls to the database system, usually through dynamically generated SQL statements (Chapter 5). Sometimes, this SQL results in poor performance.

free list
List data structure that indicates which pages in the database buffer can be used for newly read pages without interfering with other transactions or overwriting data that should be put on the database disks (Chapter 2).

frequency-ordered linked list
Linked list in which frequently accessed nodes are near the root of the list.

functional dependency
A is functionally dependent on *B* if *B* functionally determines *A* (Chapter 4).

functionally determines
"*X* determines *A*" holds for a given table *T* if, in every instance *I* of *T*, whenever two records *r* and *r′* have the same *X* values, they also have the same *A* values (*X* is one or more attributes, and *A* is a single attribute) (Chapter 4).

G

genetic algorithm
Genetic algorithms emulate the Darwinian model of evolution, in which "chromosptomes" represent candidate solutions, reproduction involves crossover, and mutation of chromosomes and the "fittest" (so far, best) chromosomes have the highest chance to succeed. The hope is to get to a better and better solution with each generation (Chapter 10).

granule
See data item (Chapter 2).

group commit
Logging strategy in which the updates of many committing transactions are written to the log at once. This reduces the number of writes to the log (Chapter 2). Its effectiveness can be improved by ensuring that the region in virtual memory holding data to be written to disk (called the log buffer in some systems) is big enough.

grouping query
Query that partitions a set of records according to some attribute(s) usually in order to perform an aggregate on those partitions. Each partition has records with the same values in those attributes. For example, the following query finds the average salary in each department. SELECT AVG(salary) as avgsalary, dept FROM employee GROUP BY dept (Chapter 4).

H

hash structure
Tree structure whose root is a function, called the hash function. Given a key, the hash function returns a page that contains pointers to records holding that key or is the root of an overflow chain. Should be used when point or selective multipoint queries are the dominant access patterns (Chapter 3).

head
Same as disk head (Chapter 2).

heap
A table organization based on insertion order. That is, each newly inserted record is added to the end of the table. Records are never moved between pages. May cause a locking bottleneck when there are many concurrent inserts (Chapter 3).

Heisenbug

Failure that appears once, corrupts no data, and never reappears. Studies have shown that the vast majority of software failures that appear in well-written mature systems are Heisenbugs (Chapter 2).

hit ratio
Number of logical accesses satisfied by the database buffer divided by the total number of logical accesses (Chapter 2). The number of logical accesses satisfied by the database buffer is the number of logical accesses minus the number of physical accesses (Chapter 7).

horizontal partitioning
Method of partitioning a set of records (or objects) among different locations. For example, all account records belonging to one branch may be in one location and the records belonging to another branch may be in another location (Chapter 4).

hot spot
Data item that is the target of accesses from many concurrent transactions (Chapter 2). If some of those transactions write the data item, a hot spot may become the object of lock contention bottleneck.

hot table
Table that is accessed by many concurrent transactions (Chapter 3).

I

identifier
An integer that uniquely identifies an object or record within a database system. This is a generalization of the notion of address from normal programming languages. An object with an identifier can be shared ("pointed to") by other objects (Chapter 4).

idle time
Wall clock time less the user and system time. Usually, this is time spent waiting for disks, networks, or other tasks (Chapter 2).

impedance mismatch
Performance and software engineering problems caused by the fact that a set-oriented language must interface with a record-oriented language. This term was popularized by David Maier to characterize the situation that occurs when programmers use a set-oriented language such as SQL embedded within a record-oriented language such as COBOL, C, or RPG (Chapter 5).

index
Data organization to speed the execution of queries on tables or object-oriented collections. It consists of a data structure (e.g., a B-tree or hash structure), and a table organization (Chapter 3).

index scan
Access path in which the keys of an index are read before or instead of the corresponding table's data pages. Index scans can often be better access paths than table scans (Chapter 7 and Appendix D).

inheritance
Facility whereby one class *A* can be defined based on the definition of a class *B*, plus some additional operations. For example, a map image class may inherit from an image class and add the operation locate_city (Chapter 4).

interesting functional dependency
"*X* determines *A*" (or $X \rightarrow A$) is interesting if *A* is not an attribute in *X* (Chapter 4). Sometimes called nontrivial.

interior node
In a data structure characterized by a tree (e.g., a B-tree structure), an interior node is anything other than a leaf node or overflow page (Chapter 3).

internal node
See interior node.

interrupt driven
Action is interrupt driven if it occurs whenever a certain event occurs. It should entail negligible overhead until that event occurs. Triggers have this property (Chapter 4).

ISAM structure
In the original IBM implementation, an ISAM was a balanced tree structure with a predetermined number of levels. Interior nodes of an ISAM structure never change, but there may be overflow

chains at the leaves. Use ISAM when range queries are important and there are few updates. Newer ISAM structures bear a closer resemblance to B-trees.

isolation level
Guarantee provided by the concurrency control algorithm (see degree 0-3 definitions as well as read committed, read uncommitted, repeatable read, and serializable).

item
See data item (Chapter 2).

J

join attribute
In a join clause of the form R.A = S.B, A is the join attribute of R, and B is the join attribute of S (Chapters 3 and 4).

join query
Query that links two or more table instances based on some comparison criterion. For example, the following query finds all students who have a higher salary than some employee: SELECT studname FROM students, employee WHERE student.salary > employee.salary. The two table instances may come from the same table. For example, the following query finds employees who earn more than their manager: SELECT e1.name FROM employee e1, employee e2 WHERE e1.manager = e2.ssnum AND e1.salary > e2.salary.

K

key
When talking about an index, the key is the set of attributes the index is defined on (e.g., *A* is the key of an index on *A*); by contrast, in normalization theory, a key of a table is a minimal set of attributes such that no two distinct records of the table have the same value on all those attributes. Notice that the two notions are related but distinct (Chapters 3 and 4).

key compression
Encoding of keys in the interior nodes of B-trees and hash structures to make them shorter. This improves the branching factor at the cost of somewhat more computation time when updating or scanning (Chapter 3).

L

link implementation
A data structure has a link implementation if every leaf node has a pointer to its right neighbor. This can improve the performance of range queries and, in a few systems, even the performance of concurrency control (Chapter 3).

load
An insertion of many records into a table. If a load occurs in isolation, then no locks are needed and indexes should be dropped for best performance. Further, if the load inserts records in sorted order into a B-tree, then check to make sure that your system performs splits in a way that achieves high utilization. All systems provide tools that bypass the query manager and possibly the storage manager to load data (Chapter 5).

lock escalation
Certain systems have the property that they will use a certain granularity of locking, say, record-level locking, for a transaction *T* until *T* has acquired more locks than some user-specified threshold. After that point, the system obtains coarser granularity locks, say, page-level locks, on behalf of *T*. The switch to coarser granularity locks is called lock escalation (Chapter 2). Escalating too early may cause unnecessary blocking. Escalating too late may cause wasted work if transactions thereby do extra work before discovering an inevitable deadlock.

locking
Denotes the activity of obtaining and releasing read locks and write locks for the purposes of concurrent synchronization (concurrency control) among transactions (Chapter 2).

log
Section of stable storage (normally disk) that contains before images or after images or both for the purposes of recovery from failure. It should always be possible to re-create the current logical state of the database (up to the last committed transaction) (provided the log has the necessary

data) by combining the log and the active database disks or by combining the log and the last database dump (Chapter 2).

logging
Denotes the activity of storing either before images or after images or both on a log for recovery purposes (Chapter 2).

logical access
Page reads and writes requested by applications. Some of these may be satisfied by accesses to the database buffer (Chapter 2).

logical logging
Technique that consists of writing the operation that caused an update as opposed to the pages modified by the update; the recorded operation might look like this: insert into tycoon (780-76-3452, Gates, Bill, ….) (Chapter 2).

lookup table
Small, read-only table used by many applications (e.g., a table that translates codes to city names). Should not be put in the database system, but rather kept in application space for best performance (Chapters 3 and 6).

M

materialization (of intermediary results)
Step in the processing of a query where a temporary table is created and has its data saved on the fly. Materialization can happen in sorts or in complex correlated subqueries, for instance (Chapter 7).

materialized view
A view normally is stored in a database only as a definition. Queries using that view translate the definition on the fly into accesses to "base" tables. A materialized view stores the data associated with the view. By definition, a materialized view is redundant information, but it can speed up queries substantially (Chapters 4 and 10).

merge join
Join technique in which the two join arguments are sorted on their join attributes and then a mergelike procedure takes place in which matching records are concatenated and output. For example, if the records of *R* are (1, 1), (2, 4), (2, 3) and the records of *S* are (2, 5), (2, 1) and they are joined on their first attributes, then the four records (2, 4, 2, 5), (2, 4, 2, 1), (2, 3, 2, 5), and (2, 3, 2, 1) would be output in that order (Chapter 3).

method
An operation that can be applied to one or to a few objects in an object-relational or object-oriented system (Chapter 4).

minibatch
A long batch transaction may use up all the memory in the database buffer, causing poor performance because of paging. Minibatch is the style of programming that consists of chopping such a long batch transaction into smaller ones. It is important to check whether this causes a violation of desired isolation properties (Chapter 2, Appendix B).

minimal
(1) For sets: no smaller set satisfies some given required properties; and (2) for functional dependencies: a set of functional dependencies without redundancies and whose left-hand sides are the minimal sets possible that express the same dependency information (Chapter 4).

mirrored disks
Set of disks that are synchronized as follows: each write to one disk goes to all disks in the mirrored set; reads can access any of the disks (Chapter 2). Several RAID levels rely on mirrored disks.

multidimensional data structure
Tree data structure that is useful for spatial queries (e.g., find all cities in some range of latitudes and longitudes) (Chapters 3 and 10). R-trees and quadtrees are prominent examples.

multiple inheritance
Object-oriented and object-relational concept implying the ability to inherit definitions from more than one class (Chapter 4).

multipoint query

Equality selection that may return several records (e.g., a query whose where clause is employee.department = 'personnel'). Such queries always benefit from clustering indexes and will benefit from nonclustering indexes if the query is selective (Chapter 3).

multiversion read consistency

A lock-free concurrency control method for read-only transactions that has the following semantic effect: each read-only transaction *T* reads the committed values that were present when *T* began. Thus, *T* may read a value *x* after another transaction *T′* has committed a change to *x*, yet the read may not reflect the write of *T′*. Instead, *T* will read the value of *x* that was last committed when it (*T*) began. Multiversion read consistency preserves serializability when read/write transactions use two-phase locking. Snapshot isolation uses multiversion read consistency but may not guarantee serializability (Chapter 2).

multiway join query

Join query in which more than two tables need to be joined (Appendix D).

N

nested-loop join

Join technique in which for each record of the outer table (external loop), all the records of the inner table are read and compared (internal loop) (Chapter 7 and Appendix D).

neural nets

Neural nets are acyclic networks in which edges represent excitation or inhibition, source nodes correspond to input data, and end nodes represent a decision (Chapter 10).

node

When speaking about a graph structure such as a circuit or a data structure such as a hash structure, B-tree, or ISAM structure, a node is a section of storage that may have pointers to other nodes or that is pointed to by pointers in other nodes or both (Chapters 3 and 5).

nonclustering index

Dense index that puts no constraints on the table organization. Several nonclustering indexes can be defined on a table. Nonclustering indexes are also called secondary indexes. See, for contrast, clustering index (Chapter 3).

nonsequential key

Opposite of sequential key (Chapter 3).

normalized

Relation *R* is normalized if every interesting functional dependency "*X* functionally determines *A*," where *A* and the attributes in *X* are contained in *R*, has the property that *X* is the key or a superset of the key of *R* (Chapter 4).

number of levels

In a B-tree structure, the number of different nodes on any path from the root to a leaf (Chapter 3). You want few levels because large data structures have their bottom levels on disk, so every dereference costs a possibly random disk access. For example, if we have 1 gigabyte of RAM and we store an index that requires 100 gigabytes, a binary tree would have approximately 7(2^7 > 100) levels on disk. A B-tree having a fanout of at least 100 would have just one.

O

object

Collection of data, an identifier, and operations (sometimes called methods) (Chapter 4).

online analytical processing (OLAP)

Denotes the class of applications where queries return aggregate information or perform searches on millions or even billions of rows. The frequency of updates is very low. These are typical of a data warehouse.

online transaction processing (OLTP)

Denotes the class of applications where the transactions are short, typically ten disk I/Os or fewer per transaction, the queries are simple, typically point and multipoint queries, and the frequency of updates is high.

operating system paging

Activity of performing disk reads to access pages that are in the virtual memory of the database buffer but not in random access memory. For a well-tuned buffer, this should never happen (Chapter 2).

optimizer
DBMS subsystem responsible for enumerating possible access plans for an SQL query and for determining the one (hopefully the best) among them to be executed.

ordering query
Query that outputs a set of records according to some sorted order on one or more attributes. For example, the following query orders the records in the employee relation from lowest salary to highest salary. SELECT * FROM employee ORDER BY salary (Chapter 4).

overflow chaining
In classical ISAM structures and hash structures (with the exception of certain exotic hash structures such as extendible hashing), when an insertion applies to a full page *p*, a new page *p′* is created and a pointer from *p* to *p′* is added. This is called overflow chaining (Chapter 3).

P

page replacement
Activity of replacing pages in the database buffer by other pages when no free pages are available. A large value of this parameter indicates either that the buffer is too small or that there are too few asynchronous write daemons (Chapters 2 and 7).

page-level lock
A page-level lock on page *p* will prevent concurrent transactions from obtaining a write lock on *p*. If the lock is a write lock, then the lock will prevent concurrent transactions from obtaining a read lock on *p* (Chapter 2).

page-level logging
Property of a logging algorithm in which the smallest data items written to the log are pages (Chapter 2).

performance indicator
A quantitative measurement of a specific DBMS internal activity that allows the tuner to assess how well that activity is performing. See hit ratio for an example of an indicator (Chapter 7).

persistent data item
One that exists after the execution of the program that creates it.

phantom problem
A failure to achieve serializability when inserts or deletes are concurrent with other transactions. See serializability for a discussion.

physical accesses
Those logical reads and writes that are not satisfied by the database buffer and that result in accesses to secondary storage (Chapter 2).

physical data independence
Assurance given by a database system that changing a data structure (e.g., adding, dropping, or reorganizing a B-tree, or replacing a B-tree by a hash structure) will not change the meaning of any program. SQL gives this assurance to the extent that an SQL query will have the same input/output behavior regardless of the indexes that support the query (Chapter 4).

point query
Equality selection that returns a single record because the where clause pertains to a key of the table (e.g., where employee.socialsecurityid = ….). Such queries benefit from indexes (Chapter 3) because an index will provide a rapid alternative to scanning an entire table.

poll
Repeatedly access a data location or table to see if it has been changed in a certain way. Much less efficient than using an interrupt-driven mechanism such as a trigger (Chapter 5).

prefetching (or sequential prefetching)
Strategy used to speed up table or index scans by physically reading ahead more pages than requested by a query at a specific point in the hope that future requests be logically fulfilled (Chapters 2 and 7).

prefix compression

Technique used in the interior nodes (the nonleaf ones) of a B-tree to reduce the length of the key portion of each key-pointer pair. For example, if three consecutive keys are Smith, Smoot, and Smythe, then only Smi, Smo, and Smy need be stored (Chapter 3).

prefix match query
Query on the prefix of a given sequence of attributes. For example, if the sequence is lastname, firstname, then lastname = 'DeWitt' AND firstname LIKE 'Da%' would be a prefix match query as would lastname LIKE 'DeW%'. By contrast, firstname LIKE 'Da%' would not be a prefix match query because firstname is not a prefix of lastname, firstname (Chapter 3).

primary index
See clustering index (Chapter 3).

priority inheritance
Scheme to avoid lock-induced priority inversion by allowing a low-priority thread that holds a lock to acquire the priority of the highest-priority thread waiting for that lock (Chapter 2).

priority inversion
Scheduling anomaly in which a higher-priority thread waits for lower-priority one. This can occur in first-in, first-out queues and in situations in which there are locks (Chapter 2).

privileged
Table *T* is privileged in a select if the fields returned by the select contain a key of *T*. For example, if ssnum is a key of Employee, then any statement of the form SELECT ssnum,… FROM Employee.… WHERE.… privileges Employee. Important when trying to eliminate the keyword DISTINCT (Chapter 4).

Q

query plan
The plan produced by an optimizer for processing a query. For example, if there is an index on R.B and on S.D, then a smart optimizer will apply the index on S.D and then scan the result from S against the index on R.B in the query SELECT R.A FROM R, S WHERE R.B = S.C AND S.D = 17 (Appendix D).

queueing theory
A mathematical theory that computes the response time of a service given the service time and arrival rate, usually assuming a Poisson arrival model. If *A* is the arrival rate and *S* is the service time, then the utilization *U* is $A \times S$ and the response time is $S/(1-U)$ (Chapter 8).

R

RAID disks
Stands for redundant array of inexpensive disks, and describes a variety of configurations for laying data out on multiple disks (Chapter 2). Most of those configurations offer some form of fault tolerance.

random access memory
Electronic (solid state memory) whose access times are in the range of 10–100 nanoseconds though access is faster if it is sequential. Frequent synonyms are real memory and main memory.

range query
Selection of the form R.A >= 5 AND R.A >= 10. That is, a selection on an interval containing more than one value (Chapter 3).

reaches
Table *S* reaches a table *T* if *S* is joined by equality on one of its key fields to *T*. If dept is a key of Techdept, then Techdept reaches Employee in the query SELECT … FROM Employee, Techdept WHERE Techdept.dept = Employee.dept. Important when trying to eliminate the frequency of DISTINCT; see privileged (Chapter 4).

read committed
Isolation level in the ANSI/ISO SQL standard. Equivalent of degree 2 isolation.

read lock
If a transaction *T* holds a read lock on a data item *x*, then no other transaction can obtain a write lock on *x* (Chapter 2).

read uncommitted

Isolation level in the ANSI/ISO SQL standard. Equivalent of degree 1 isolation.

real-time database
One in which response time guarantees must be met. These can be statistical (soft real time) or absolute (hard real time) (Appendix A).

record
Row in a relational table.

record-level lock
A record-level lock on record *r* will prevent concurrent transactions from modifying *r*. If the lock is a write lock, then the lock will prevent concurrent transactions from accessing *r* altogether (Chapter 2).

record-level logging
Property of a logging algorithm for a relational database system in which the data items held on the log are relational records (Chapter 2).

redo-only
Property of a logging algorithm in which a transaction *T* performs no updates to the database disks before *T* commits (Chapter 2).

redo-undo
Property of a logging algorithm in which a transaction *T* may perform updates to the database disks before *T* commits. The implication is that before images must be written to the log while *T* is active (Chapter 2). Also known as write-ahead logging, the most frequent recovery method used because it gives maximum freedom to the buffer manager and thereby avoids seeks.

regression
A statistical technique that infers an equation from a set of points (Chapter 10 and Appendix C). The equation then permits predictions about the future.

relation instance
Set of records that conforms to some relation schema (Chapter 4).

relation schema
Relation name and a set of attributes, or, equivalently, a table name and a set of column headers. Each attribute has a data type. In normalization theory, the functional dependencies on the attributes of the schema are considered to be part of the schema (Chapter 4).

relationship
Notion from the entity-relationship model. A relationship links two entity types (e.g., the relationship works_in links entity types employee and organization) (Chapter 4).

reorganization
Strategy in which a table's rows or an index's keys get physically rearranged in disk in an optimal manner (Chapter 7).

repeatable reads
Assurance given by a concurrency control algorithm that if a transaction *T* issues two reads to the same data item *x* without modifying *x* in between, then the two reads will return the same value (Chapter 2). This is a property of degree 3 isolation. Repeatable read is defined as an isolation level in the ANSI/ISO SQL standard.

replicated state machine
A distributed fault-tolerance mechanism that is used in airplanes and could be used in main memory databases. The idea is simple: if a set of processing sites begin in the same state, execute a set of operations (or transactions) sequentially and in the same order, and all operations are deterministic (they neither access external sources of randomness nor the time), then the sites will end in the same state. Main memory databases on uniprocessors can execute transactions sequentially without loss of efficiency because each transaction can utilize the processor from beginning to end. On multiprocessor devices, you can get the effect of a replicated state machine by ensuring that only nonconflicting transactions execute concurrently.

response time
Time it takes to get a response after submitting a request (Chapter 8).

rollback
Action of undoing all effects of a transaction that has not yet committed. Rollback may result from internal program logic (e.g., roll back the purchase if cash is low) or from a deadlock in the concurrency control subsystem (Chapter 2).

rotational delay
Time required to wait for the proper portion of the track to pass underneath the head when performing a read or write. Around 2–7 milliseconds for magnetic disks (Chapter 2).
row-expanding updates
UPDATE command in which a variable-length column is updated to a larger value, possibly causing the row to be displaced. See also rowID (Chapters 2 and 7).
rowID or RID
Internal, unique identifier of a row. Usually made up of the page number and the sequence number where the row was initially stored (Chapter 7).

S

scan
See table scan (Chapter 3).
secondary allocation
Storage allocated for a database object when it runs out of the space reserved for it at creation time (Chapter 7).
secondary index
See nonclustering index (Chapter 3).
seek
Head movement required to position a head over a given track (Chapter 2).
seek time
Time required to do a seek (around 5–10 milliseconds for most magnetic disks depending on the distance that must be traveled) (Chapter 2).
selectivity
A selection (e.g., R.C = 5) has high selectivity if it returns few records (i.e., far fewer records than there are pages in R). Point queries have extremely high selectivity, whereas multipoint queries may or may not have high selectivity (Chapter 3).
semijoin condition
Property of a join between two tables *R* and *S* such that no fields of one of the tables, say, *S*, are in the result (e.g., SELECT R.A, R.B FROM R, S WHERE R.C = S.E). If *S* is indexed on *E*, then the data records of *S* never need to be accessed. Some systems take advantage of this (Chapter 3). See covering index.
sequential key/field
Field whose values are monotonic with the time of insertion (i.e., later insertions get higher field values) (Chapter 3). For example, if the timestamp were a field, it would be sequential.
serializability
Serializability is the assurance given by a concurrency control algorithm that in a concurrent execution of a set of transactions that may contain any operations, whichever transactions commit will appear to execute one at a time. This is stronger than degree 3 isolation, which does not make this assurance in the case of inserts and deletes. The problem with inserts and locking can be illustrated as follows: if one transaction T_1 begins to sum the salaries in the employee table, and another transaction T_2 inserts two new employees, one in the middle of the table and one at the end, T_1 may see the effect of the first insert but not the second even though it is using two-phase locking. This phenomenon is known as the phantom problem. One way to solve the phantom problem is to have the summing transaction like the "end of table" pointer, so the conflict with the second insert would result in blocking or deadlock.
serializable
Isolation level in the ANSI/ISO SQL standard. Provides serializability guarantee.
service time
The time it takes for a particular server to handle one request. If there is a cluster of *N* servers that use no common resource, then the effective service time we would suggest for the cluster is the single server service time/(0.8*N*). The 0.8 is a fudge factor that reflects the fact that there is nearly always some common resource (Chapter 8).
shared disk
Hardware configuration in which each processor has its private random access memory but all disks are shared. This is a very common configuration for database systems (Chapter 2).

shared lock
See read lock (Chapter 2).
shared nothing
Hardware configurations in which each processor has its private random access memory and disks (Chapter 2). An example would be a set of personal computers linked by a network.
snapshot isolation
A concurrency control technique in which reads obtain no locks but writes are "certified" upon commit. Snapshot isolation does not guarantee serializability in all cases. Here are the details of the protocol: A transaction *T* executing under snapshot isolation reads data from the committed state of the database as of time start(*T*) (the "snapshot"). *T* holds the results of its own writes in local memory store so if it reads data a second time it will read its own output. Aside from its own output, *T* reads data only from the committed state of the database as of time start(*T*). Thus, writes performed by other transactions that were active after *T* started are not visible to *T*. When *T* is ready to commit, it obeys the first-committer-wins rule: *T* will successfully commit if and only if no other concurrent transaction *T′* has already committed writes to data that *T* intends to write. Here is an example of the failure to guarantee serializability: if transaction T_1 assigns *x* to *y* and T_2 assigns *y* to *x*, then in any serializable execution, *x* and *y* will have the same values after T_1 and T_2 end. With snapshot isolation, this may not be the case. Snapshot isolation is a form of multiversion read consistency (Chapter 2).
sparse index
Index in which the underlying data structure contains exactly one pointer to each data page. Only clustering indexes can be sparse (Chapter 3).
split
When an insertion occurs to a full B-tree page *p*, a new page is created and the tree is locally restructured, so the new page and *p* are at the same distance from the root (Chapter 3).
stable storage
That portion of memory that tolerates "normal" failures. For example, if you assume that multiple disks will never fail (perhaps because disks are mirrored or some version of RAID is used), then disk storage is stable (Chapter 2). Note, however, that this assumption may be violated if disks come from the same manufacturing batch in which case several may fail nearly at the same time.
star schema
A set of tables whose "center" (analogous to the nucleus of the star) is a very large "fact table" (e.g., sales) having foreign key dependencies to other tables (e.g., on itemid to itemdescription, on storeid to storelocation, and on customerid to customerdescription) (Chapter 10).
statistics
Set of information stored in the catalog about cardinality of relations, distributions/frequency of values of important columns, and so on. Statistics should be up to date for the SQL optimization process to work correctly (Chapter 7 and Appendix D).
storage parameters
Space allocation parameters of a database object at creation time. These parameters determine how existing free space is managed and how page replacement is done when needed (Chapter 7 and Appendix E).
system event
Important developments in the internal operations of a DBMS like the beginning or end of execution of a statement. Performance monitors that record events can help the tuner solve problems as deadlocks (Chapter 7).
system time
The processor time some activity takes while executing operating system code (Chapter 2).

T

table
In technical usage, a table is the same as a relation.
table clustering
The interleaving of two tables for the purpose of reducing the cost of a join. For example, each customer record might be followed by the sales records pertaining to that customer. Oracle is the

main vendor offering table clustering, and it recommends the use of that facility only in rare cases (Chapter 4).

table scan
Examination of every record of every page of a table.

table-level lock
A table-level lock on table *t* will prevent concurrent transactions from modifying *t*. If the lock is a write lock, then the lock will prevent concurrent transactions from accessing *t* altogether (Chapter 2).

tablespace
Physical portion of disk storage containing either an index or one or more tables.

thrashing
Phenomenon in which paging escalates to unbearable levels, bringing the performance of the whole system significantly down (Chapter 7).

thread
Unit of execution consisting of a program counter, a possibly size-limited stack with references to a shared address space and program text (Chapter 2).

three-tier architecture
Web servers at the outer layer are responsible for presentation. Application servers contain the code embodying business logic. Database servers hold the data (Chapter 8).

throughput
The number of transactions that complete per unit time.

tightly coupled
Hardware configuration in which all processors share random access memory and disks (Chapter 2).

timeout
Name given to transactions that were aborted due to excessive lock waiting time. See also deadlock (Chapter 7).

track
Narrow circle on a single platter of a disk. If the disk head over a platter does not move, then a track will pass under that head in one rotation. One implication is that reading or writing a track does not take much more time than reading or writing a portion of a track (Chapter 2).

transaction
Unit of work within a database application that should appear to execute atomically (i.e., either all its updates should be reflected in the database or none should; it should appear to execute in isolation). In IBM protocols, transactions are known as logical units of work (Chapter 2).

transaction chopping
The division of a transaction into smaller transactions for the purposes of reducing concurrency control–induced contention or buffer contention (Chapter 2, Appendix B).

trie
A special kind of tree structure that is good for an in-memory lookup table that supports prefix queries (Chapter 3). Suffix trees and suffix arrays are generalizations of tries.

trigger
Stored procedure that performs an action when some table is updated in a certain way (e.g., whenever there is an insertion to table T, print the records that are inserted) (Chapter 4).

tuple
See record.

two-phase commit
Algorithm for terminating transactions that are distributed over many sites (specified by the LU6.2 protocol in the IBM world). Two-phase commit entails an overhead of between two and four messages per transaction per site and may cause one site to block because of the failure of another one (Chapter 6). Contrast with replicated state machine.

two-phase locking
Algorithm for concurrency control whereby a transaction acquires write locks on data items that it will write, read locks on data items that it will only read, and the transaction never obtains a lock after releasing any lock (even on another data item). Two-phase locking can encounter deadlock

(Chapter 2). Two-phase locking guarantees degree 3 isolation and can be extended to guarantee serializability.

U

unbuffered commit strategy
Logging algorithm in which a transaction's updates are forced to the database disks immediately after commit. This implies that there will be many random writes to those disks, thus hurting performance. This strategy should never be used (Chapter 2).

unnormalized
Opposite of normalized (Chapter 4).

usage factor
The percentage of a page that can be utilized, yet still permit a further insertion. This influences utilization (Chapter 2).

user
In the context of this book, a user is someone who uses a database management system as opposed to a developer of the database management system itself. That is, a user may be a database administrator or an informed application developer.

user time
The processor time some activity takes while executing nonoperating system code.

utilization
That percentage of an index or data page that actually has data on it as opposed to free space. For example, a B-tree has an average utilization of about 69% (Chapter 3).

V

vertical antipartitioning
Technique for storing small sets in a single record. For example, if one relation contains the attributes (bond, descriptive information) and another one contains (bond, date, price), then vertical antipartitioning may consist of putting some of the prices (e.g., the last ten days' worth) with the descriptive information (Chapter 4).

vertical partitioning
Method of dividing each record (or object) of a table (or collection of objects) such that some attributes, including a key, of the record (or object) are in one location and others are in another location, possibly another disk. For example, the account ID and the current balance may be in one location and the account ID and the address information of each record may be in another location (Chapters 4 and 5).

W

wall clock time
Time some activity takes as recorded by an external observer looking at a clock (Chapter 2).

Web cache
Cache for Web pages in the Web server tier of an e-commerce application (Chapter 8).

Web servers
The component of an e-commerce application that delivers pages (e.g., in XML or HTML) to client browsers (Chapter 8).

workload
Mix of all queries and commands issued to a particular database application (Chapter 7).

write lock
If a transaction *T* holds a write lock on a data item *x*, then no other transaction can obtain any lock on *x* (assuming degree 2 or 3 isolation) (Chapter 2).

Index

Symbol and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

List of Figures

Chapter 2: Tuning the Guts

scan is performed and row locks are acquired unless table locking is forced using the LOCK TABLE command. This graph was obtained using DB2 UDB V7.1 on Windows 2000.
Figure 2.11: **Fine-grained locking (Oracle).** A long transaction (a summation query) with multiple short transactions (debit/credit transfers). Because snapshot isolation is used the summation query does not conflict with the debit/credit transfers. Table locking forces debit/credit transactions to wait, which is rare in the case of row locking. As a result, the throughput is significantly lower with table locking. This graph was obtained using Oracle 8i EE on Windows 2000.
Figure 2.12: **Multiple insertion points and page locking.** There is contention when data is inserted in a heap or when there is a sequential key and the index is a B-tree: all insertions are performed on the same page. Use multiple insertion points to solve this problem. This graph was obtained using SQL Server 7 on Windows 2000.
Figure 2.13: **Multiple insertion points and row locking.** Row locking avoids contention between successive insertions. The number of insertion points thus becomes irrelevant: it is equal to the number of inserted rows. This graph was obtained using SQL Server 7 on Windows 2000.
Figure 2.14: **Counter facility.** There is a significant difference in throughput between *system* insertions that rely on a counter facility for generating counter values (sequence for Oracle, identity data type for SQL Server) and *ad hoc* insertions that explicitly increment a counter attribute in an ancillary table for generating counter values. This difference is due to blocking when accessing the ancillary counter table. Note that in Oracle, we use the default sequence iteration mechanism that caches blocks of 20 sequence numbers. This graph was obtained using SQL Server 7 and Oracle 8i EE on Windows 2000.
Figure 2.15: **Transaction states.** Once a transaction commits or aborts, it cannot change its mind. The goal of the recovery subsystem is to implement this finite state automaton.
Figure 2.16: **Stable storage holds the log as well as data.** Consider a transaction T that writes values stored on two pages P_i and P_j. The database system generates log records for these two write operations: lr_i and lr_j (which contain the after images of P_i and P_j—or a logical representation of the write operation). The database system writes the log records to stable storage before it commits transaction T. The dirty pages P_i and P_j are written after the transaction is committed, unless the database buffer is full and these pages are chosen as victims by the page replacement algorithm. If the buffer fails, then data will be moved from the log to the database disks.
Figure 2.17: **Group commit.** Increasing the group commit size improves performance. This experiment was performed using DB2 UDB V7.1 on Windows 2000.
Figure 2.18: **Log file location.** For this experiment, we use the lineitem table from the TPC-H benchmark and we issue 300,000 insert or update statements. This experiment was performed with Oracle 9i on a Linux server with internal hard drives (no RAID controller). Each statement constitutes a separate transaction, and each transaction forces writes. This graph shows the throughput obtained with the log and the data on separate disks as opposed to the throughput obtained with the log and the data on the same disk. Locating the log file on a separate disk gives approximately a 30% performance improvement. Note that we also performed this experiment using separate disks on a RAID controller. In that case, the disk controller cache hides much of the negative impact of the seeks that are necessary to switch from the log to the data when they are both located on the same disk. The use of the disk controller cache is further discussed in the storage subsystem section.
Figure 2.19: Disk organization.
Figure 2.20: **Checkpoints.** A small log file forces checkpoints. If the log file cannot accommodate all the long entries generated during a long update transaction, it forces dirty data on disk. Here, four checkpoints are triggered: they have a negative impact on throughput. This experiment was performed on Oracle 8i EE on Windows 2000.
Figure 2.21: **Priority inversion.** T_1 waits for a lock that only T_3 can release. But the system runs T_2, which has a higher priority than T_3, thus blocking both T_1 and T_3.
Figure 2.22: **Buffer organization.** The database buffer is located in virtual memory (RAM and paging file). Its greater part should be in RAM. It is recommended to have the paging file on a

separate disk. If not possible, the paging file should be on a data disk rather than on the log disk so that paging does not disrupt sequential access to the log.
Figure 2.23: **Buffer size.** These experiments are performed with a warm buffer (the table is scanned before each run). The scan query is processed inside the RAM if the table fits in the buffer. Otherwise, the table is entirely read from disk because the LRU (least recently used) replacement policy will systematically evict pages before they are reread. The performance of the multipoint query increases linearly as the buffer size increases, up to the point where the table fits entirely in memory. This experiment was performed with SQL Server 7 on Windows 2000.
Figure 2.24: **Disk usage factor.** For this experiment, we use a simple query that scans the lineitem table of the TPC-H benchmark (http://www.tpc.org) when it is located on disk and the memory is cold. We use an aggregation query to reduce the effects of transmitting the lineitem tuples through the database client interface. Throughput increases from approximately 10% as the disk usage increases from 70% to 100%. This experiment was performed using DB2 UDB V7.1 on Windows 2000.
Figure 2.25: **Prefetching.** We use the lineitem table from TPC-H. The scan query is an aggregation query to reduce the effects of transmitting the lineitem tuples through the database client interface. Throughput increases by about 10% when the prefetching size increases from 32 Kb to 128 Kb and doesn't change thereafter. This experiment was performed using DB2 UDB V7.1 on Windows 2000.
Figure 2.26: **RAID and write-intensive applications.** The negative impact of the additional read and write operations required by RAID 5 is obvious for software RAID 5. When the controller is responsible for these operations (hardware RAID 5), however, it manages to hide their impact on performance thanks to its cache. This experiment was performed using SQL Server 7 on Windows 2000.
Figure 2.27: **RAID and read-intensive applications.** RAID 1 slightly improves on a single disk solution. The striped RAID levels (RAID 0, RAID 5, and RAID 10) significantly improve read performance by partitioning the reads across multiple disks. This experiment was performed using SQL Server 7 on Windows 2000.
Figure 2.28: **Controller cache.** Using the cache controller (in write-back mode) provides similar benefits whether the write-intensive application is *cache friendly* (the volume of update is slightly larger than the controller cache) or *cache unfriendly* (the volume of update is ten times larger than the controller cache). This experiment was performed using SQL Server 7 on Windows 2000.
Figure 2.29: **Degree of parallelism.** This experiment is performed with DB2 UDB V7.1 on a dual-processor shared memory multiprocessor running Windows 2000. When data is located in memory, the CPU is the critical resource and the cost of synchronization (between the threads that produce the data and the threads that consume them in order to transmit them to the client) and of context switches is higher than the benefit of increased data throughput that would result from increased multithreading. By contrast, when data is located on disk, a higher degree of parallelism significantly improves the performance of random access transactions as are typically found in online transaction processing. (The improvement for random access is further aided by the list prefetching mechanism implemented in DB2 that collects random accesses to a table and sorts them in order to minimize disk seek time.) For sequential access transactions such as are found in data warehouse applications, the disk is the bottleneck, and as a result, the throughput remains constant with increasing levels of multithreading.
Figure 2.30: **Hardware architectures.** This diagram illustrates the various hardware architectures described in the text. Site 1 implements a shared everything architecture: a mainframe provides tight coupling. Site 2 implements a shared disk architecture. This is an increasingly popular alternative to tight coupling—disks are either tethered to servers or directly attached to the network (NAS). Site 3 implements a shared nothing architecture with a cluster (where each node usually implements tight coupling). A shared nothing architecture is actually implemented between these different sites.

Chapter 3: Index Tuning

Figure 3.1: **Place of indexes in the architecture of a typical database system.** Indexes are provided by the storage manager. They organize the access to data in memory and, for

clustering indexes, also organize the layout of data on disks. Indexes are tightly integrated with the concurrency control mechanisms. They are heavily used by the query processor during query optimization.
Figure 3.2: **Levels and branching factor.** This tree has five levels and a fanout of four.
Figure 3.3: **Example of B+ tree.** Leaf nodes contain data entries (in this diagram, the data is represented by the box next to each key). All data entries are at the same distance from the root; that is the meaning of balance. Nonleaf nodes contain key-pointer pairs. There are actually *m* keys and m+1 pointers on each nonleaf node. Each pointer P_i, associated to a key K_i, points to a subtree in which all key values *k* lie between K_i and K_i+1 (P_0 points to a subtree in which all key values are less than K_0 and P_m points to a subtree in which all key values are greater than K_m). In most implementations, leaf nodes (and nonleaf nodes at the same level) are linked in a linked list to facilitate range queries and improve concurrency.
Figure 3.4: **Hash structure with chain overflow.**
Figure 3.5: **Data organization.** This diagram represents various data organizations: a heap file (records are always inserted at the end of the data structure), a clustering index (records are placed on disk according to the leaf node that points to them), a nonclustering index (records are placed on disk independently of the index structure), a sparse index (leaf nodes point to pages), and a dense index (leaf nodes point to records). Note that a nonclustering index must be dense, whereas a clustering index might be sparse or dense.
Figure 3.6: **Clustering index.** For all three systems, a clustering index is twice as fast as a nonclustering index for a multipoint query and orders of magnitude faster than a full table scan (no index). Each multipoint query returns 100 records out of the 1,000,000 that the relation contains. These experiments were performed on DB2 UDB V7.1, Oracle 8i and SQL Server 7 on Windows 2000.
Figure 3.7: **Index maintenance after insertions—DB2.** The performance of the clustered index degrades with insertions. Once the index is full, additional records are simply appended to the relation. Each access is thus composed of a traversal of the clustering index followed by a scan of the additional records. In this experiment, a batch of 100 multipoint queries is asked. After each table reorganization, the index regains its original performance. This experiment was performed using DB2 UDB V7.1 on Windows 2000.
Figure 3.8: **Index maintenance after insertions—SQL Server.** The performance of the clustered index degrades fast with insertions. Once the index is full, pages are split to accommodate new records in the index structure. For the multipoint query we are running in this experiment, the page split results in extra I/O for each of the 100 queries in our batch. After dropping and re-creating the index, performances are back to what they were before the insertions. This experiment was performed on SQL Server 7 on Windows 2000.
Figure 3.9: **Index maintenance after insertions—Oracle.** In Oracle, the notion of clustering and indexing are orthogonal. All indexes are nonclustering (except for index-organized tables whose application is restricted to unique indexes on a primary key). In the general case, a clustering index can be approximated by an index defined on a clustered table. There is, however, no automatic physical reorganization of the clustered table when the index is reorganized. The only way to perform maintenance is to export and reimport the table. This experiment was performed on Oracle 8i EE on Windows 2000.
Figure 3.10: **Index maintenance after 10% updates.** The updates do not concern the key attributes. For DB2 and SQL Server, updates introduce a penalty comparable to the one caused by insertions (see experiments). In Oracle, updates just add data to the existing clusters (by default one page is reserved for each cluster value). In this case, there is no overflow. Consequently, Oracle's index performance is not affected by the updates. These experiments were performed on DB2 UDB V7.1, Oracle 8i, and SQL Server 7 on Windows 2000.
Figure 3.11: **Covering index.** This experiment illustrates that a covering index can be as good as or even better than a clustering index as long as the prefix match query that is asked matches the order in which the attributes have been declared. If it is not the case, then the composite index does not avoid a full table scan on the underlying relation. A covering index is also significantly faster than a nonclustering index that is not covering because it avoids access

to the table records. This experiment was performed with SQL Server 7 on Windows 2000; that is, the clustering index is sparse.
Figure 3.12: **Nonclustering index.** We use DB2 UDB V7.1 on Windows 2000 for this experiment. We use a range query and observe that the nonclustering index is advantageous when less than 15% of the records are selected. A scan performs better when the percentage of selected records is higher.
Figure 3.13: **Join with few matching records.** A hash join with no index is faster than an indexed nested loop join relying on a clustering index on the joining attribute because the clustering index is sparse. The nonclustering index is ignored: a hash join is used. This experiment was performed using SQL Server 7 on Windows 2000.
Figure 3.14: **Join with many matching records.** A hash join with no index performs worse than the other methods because setting up the buckets requires a lot of memory and disk reorganization. This experiment was performed using SQL Server 7 on Windows 2000.
Figure 3.15: **Indexes and updates.** This graph shows the potential benefits for updates of creating an index on a small table (100 tuples). Two concurrent processes update this small table; each process works for 10 ms before it commits its update. When no index is used, the small table needs to be scanned when performing an update, and locks are requested for all the rows that are traversed by the scan operation. Concurrent updates are thus impossible. On the contrary, the presence of a clustered index on the attribute on which the update condition is expressed permits concurrent updates. This experiment was performed with SQL Server 2000 on Windows 2000.
Figure 3.16: **Low index overhead for insertion.** Using SQL Server 7 on Windows 2000, we insert 100,000 records in the TPC-B table account(number, branchnum, balance). We observe that the cost of inserting data in a single clustering or a single nonclustering index is similar to the cost of inserting data in a heap; the overhead becomes significant when the number of nonclustering indexes increases, and the number of concurrent threads performing the insertions increases.
Figure 3.17: **High index overhead for insertion.** Using Oracle 9i on a Linux server, we insert 100,000 records in the table Order(ordernum, itemnum, quantity, purchaser, vendor). We measure throughput with or without a nonclustered index defined on the ordernum attribute. The presence of the index significantly affects performances.
Figure 3.18: **Text indexing.** Oracle 8i with the Intermedia extension supports inverted indexes for substring. An inverted index defined on a text column is a sequence of (key, pointer) pairs where each word in the text is a key whose associated pointer refers to the record it has been extracted from. This experiment compares the performance of multipoint queries based on equality and substring predicates using an inverted index and a B+ tree. We defined these indexes on the comments attribute of the TPC-H lineitem relation; this attribute is of type varchar(44). Using a B+ tree the equality predicate is =, and the substring predicate is LIKE. When an inverted index is defined, both equality and substring predicates are constructed using the CONTAINS function that searches through the inverted index. The graph shows that the performance of the inverted index and B+ tree are comparable for exact matches, while the inverted index performs slightly better on substring search. Note that inverted indexes can be defined on large objects(LOBs), whereas B+ trees cannot.
Figure 3.19: **Comparison of B-tree, hash, and bitmap index multipoint queries.** One hundred records are returned by each equality query. In the hash structure, these 100 records map to the same hash key, thus requiring an overflow chain. The clustered B-tree offers good performance because the records returned are on contiguous pages. The bitmap index does not perform well because it is necessary to traverse the entire bitmap to fetch just a few matching records. This experiment was performed using Oracle 8i EE on Windows 2000.
Figure 3.20: **Comparison of B-tree, hash, and bitmap index—range queries.** As expected, hash indexes don't help when evaluating range queries. This experiment was performed using Oracle 8i EE on Windows 2000.
Figure 3.21: **Comparison of B-tree and hash index—point queries.** Hash index outperforms B-tree on point queries. This experiment was performed using Oracle 8i EE on Windows 2000.
Figure 3.22: **Distributing indexes.** In this experiment, placing the index on a separate disk provides some advantage when inserting data. Point or multipoint queries do not always

benefit from separating data and indexes. We used Oracle 8i on Windows 2000 for this experiment.
Figure 3.23: **Partitioning indexes.** Data is range partitioned on three disks; on each disk reside index and data. Insertions are slower compared to a single disk storage. Point queries benefit slightly from this partitioning. This experiment was performed on Oracle 8i EE on Windows 2000.

Chapter 4: Tuning Relational Systems

Figure 4.1: **Database system architecture**. Responsibility of people with different skills.
Figure 4.2: **Purchase table.**
Figure 4.3: **Entities.** This entity-relationship diagram represents the hospital and doctor entities in our sample hospital application as well as the works_in relationship connecting those entities.
Figure 4.4: **Vertical partitioning and scan queries.** A relation *R* is defined with three attributes *X* (integer), *Y*, and *Z* (large strings); a clustered index is defined on *X*. This graph compares the performance of scan queries that access all three attributes, or only two of them (*X* and *Y*) depending on whether vertical partitioning is used or not. Vertical partitioning consists of defining two relations R_1 and R_2: R_1 has two attributes *X* and *Y* with a clustered index on *X*, whereas R_2 has two attributes *X* and *Z* with a clustered index on *X*, too. As expected, the graph shows that vertical partitioning exhibits poorer performance when all three attributes are accessed (vertical partitioning forces a join between R_1 and R_2 as opposed to a simple lookup on *R*) and better performances when only two attributes are accessed (fewer pages need to be scanned). This graph was obtained with SQL Server 2000 on Windows 2000.
Figure 4.5: **Vertical partitioning and single point queries.** A relation *R* is defined with three attributes *X* (integer), *Y*, and *Z* (large strings); a clustered index is defined on *X*. A mix of point queries access either all three attributes, or only *X* and *Y*. This graph shows the throughput for this query mix as the proportion of queries accessing only *X* and *Y* increases. We compare query execution with or without vertical partitioning of *R*. Vertical partitioning consists of defininning two relations R_1 and R_2: R_1 has two attributes *X* and *Y* with a clustered index on *X*, whereas R_2 has two attributes *X* and *Z* with a clustered index on *X*, too. The graph is obtained by running five clients that each submit 20 different point queries either on *X, Y, Z* (these require a join if vertical partitioning is used) or only on *X, Y*. The graph shows that vertical partitioning gives better performances when the proportion of queries accessing only *X* and *Y* is greater than 20%. In this example the join is not really expensive compared to a simple lookup—indeed the join consists of one indexed access to each table followed by a tuple concatenation. This graph was obtained with SQL Server 2000 on Windows 2000.
Figure 4.6: **Denormalization.** We use the TPC-H schema to illustrate the potential benefits of denormalization. This graph shows the performance of a query that finds all lineitems whose supplier is in Europe. With the normalized schema, this query requires a four-way join between lineitem, supplier, nation, and region. If we denormalize lineitem and introduce the name of the region each item comes from, then the query is a simple selection on lineitem. In this case, denormalization provides a 30% improvement in throughput. This graph was obtained with Oracle 8i EE running on Windows 2000.
Figure 4.7: **Aggregate maintenance.** We implemented the *total amount* example given in Section 4.4 to compare the benefits and costs of aggregate maintenance using SQL Server 2000 on Windows 2000. The aggregate maintenance solution relies on triggers that update the VendorOutstanding and StoreOutstanding relations whenever a new order is inserted. The graph shows the gain obtained with aggregate maintenance. For inserts this gain is negative: the execution of the triggers slows down insertions by approximately 60%. For queries, however, the gain is spectacular: response time is reduced from 20 seconds to 0.1 second. The reason is that the "total amount" queries are simple scans on very small relations when aggregate maintenance is used, whereas they are three- or four-way joins on the large order relations otherwise.
Figure 4.8: **Aggregate maintenance with materialized views (queries).** We implemented the *total amount* example given in Section 4.4 to compare the benefits and costs of aggregate maintenance using Oracle 9i on Linux. Materialized views are transparently maintained by the system to reflect modifications on the base relations. The use of these materialized views is also transparent; the optimizer rewrites the aggregate queries to use materialized views if

appropriate. We use a materialized view to define VendorOutstanding. The speed-up for queries is two orders of magnitude.
Figure 4.9: **Aggregate maintenance with materialized views (insertions).** There are two main parameters for view materialization maintenance: (1) the materialized view can be updated in the transaction that performs the insertions (ON COMMIT), or it can be updated explicitly after all insert transactions are performed (ON DEMAND); (2) the materialized view is recomputed completely (COMPLETE) or only incrementally depending on the modifications of the base tables (FAST). The graph shows the throughput when inserting 100,000 records in the orders relation for FAST ON COMMIT and COMPLETE ON DEMAND. On commit refreshing has a very significant impact on performance. On demand refreshing should be preferred if the application can tolerate that materialized views are not completely up to date or if insertions and queries are partitioned in time (as it is the case in a data warehouse).
Figure 4.10: **Query tuning.** This graph shows the percent increase in throughput between the original query and the rewritten query for the rewriting techniques we have presented in this section: distinct refers to the suppression of unnecessary DISTINCTs, subquery refers to the use of joins instead of uncorrelated subqueries without aggregates, correlated subquery refers to the decomposition of correlated subqueries using intermediate tables, join and numeric attribute refers to the use of a numeric attribute rather than an equivalent string attribute as the joining attribute, join and clustered index refers to the use of attributes on which a clustered index is defined on join attributes, having refers to the incorporation of selection conditions in the WHERE clause rather than in the HAVING clause, and view refers to the use of a selection on a base table instead of a view expanded to a join. These experiments were performed using IBM UDB V7.1, Oracle 8i, and SQL Server 2000 on Windows 2000.
Figure 4.11: **Purchase table.**

Chapter 5: Communicating With the Outside

Figure 5.1: ODBC versus native call-level interface. We compare the throughput obtained using ODBC and OCI (the Oracle call-level interface) to retrieve records from the database server into the application. Note that the ODBC driver we use for this experiment is implemented on top of OCI. The results show (1) that the connection and query preparation overhead is much lower with OCI than ODBC, almost twice as low, and (2) that the fetching overhead is lower for ODBC than OCI. When the number of records fetched from the result set increases, the throughput increases more with ODBC than with OCI. The reason is that the ODBC driver does a good job at implicitly prefetching records compared to our straightforward use of OCI.
Figure 5.2: **Client-server connection.** Client-server communication is mediated by a buffer on the server site, usually one per connection.
Figure 5.3: **Loop constructs.** This graph compares two programs that obtain 2000 records from a large table (lineitem from TPC-H). The *loop* program submits 200 queries to obtain this data, whereas the no loop program submits only one query and thus displays much better performance. This graph was obtained using SQL Server 2000 on Windows 2000.
Figure 5.4: **User-defined functions.** We compare processing a function on the client site (retrieving all data + executing the function) with executing a function as a UDF within a query. The function computes the number of working days between two dates; the query selects the records in the lineitem table where the number of working days between the date of shipping and the data the receipt was sent is greater than five working days (80% of the records) or smaller than five working days (20% of the records). Using the UDF reduces the amount of data transferred, but applying the function at the application level happens to be faster when there are many records. This graph was obtained using SQL Server 2000 on Windows 2000.
Figure 5.5: **Retrieve needed columns only.** This graph illustrates the impact on performance of retrieving a subset of the columns as opposed to retrieving all columns. In this experiment, we compare retrieving one-fourth of the columns with retrieving all columns using *select **. We performed this experiment in two situations: (1) without indexes and (2) with a nonclustering index covering the projected columns. The overhead of crossing the database interface with larger amounts of data is significant. Using covered indexes yields an additional performance boost to the carefully written query. This experiment was run on Oracle 8i on Windows 2000.

Figure 5.6: **Beware of cursors.** This experiment consists in retrieving 200,000 rows from the table Employee (each record is 56 bytes) using a set-oriented formulation (SQL) or a cursor to iterate over the table contents (cursor). Using the cursor, records are transmitted from the database server to the application one at a time. This has a very significant impact on performance. The query takes a few seconds with the SQL formulation and more than an hour using a cursor. This experiment was run on SQL Server 2000 on Windows 2000.
Figure 5.7: **Benefits of precompiled queries.** This graph illustrates a benefit of precompiled queries. We have run a simple query (uncorrelated subquery without aggregate) several times either by submitting the query each time (using ODBC direct execution) or by compiling it once (using ODBC prepare command) and executing it repeatedly. The results show that precompilation is advantageous when the query is executed more than twice.
Figure 5.8: **Batch size.** This graph shows the influence of the batch size on performance. We used the BULK INSERT command to load 600,500 tuples into the lineitem relation on SQL Server 2000 on Windows 2000. We varied the number of tuples loaded in each batch. The graph shows that throughput increases steadily until batch size reaches 100,000 tuples, after which there seems to be no further gain. This suggests that a satisfactory trade-off can be found between performance (the larger the batches the better up to a certain point) and the amount of data that has to be reloaded in case of a problem when loading a batch (the smaller the batches the better).
Figure 5.9: **Direct path.** This graph illustrates the performance benefits obtained by bypassing the SQL engine (conventional usage of SQL * Loader with a commit every 100 records) and the storage manager (direct path option of SQL * Loader) compared to the performance of inserts (using one thread and a commit after each insertion). These results were obtained by inserting 600,500 tuples into the lineitem relation on Oracle 8i on Windows 2000.
Figure 5.10: **Storage engine parameters.** This graph illustrates the influence of three parameters on the performance of the DB2 UDB data loading utility. We first loaded 600,500 records in the lineitem relation into DB2 UDB V7.1 on Windows 2000 using the recoverable option (before images are maintained so that the original relation can be restored), no statistics were collected, and a clustering index was rebuilt after data was loaded. We varied in turn each of these parameters. As expected, performing a nonrecoverable load increases throughput, whereas collecting statistics decreases throughput. The impact of these parameters on performance is, however, not dramatic. Incremental index maintenance (as opposed to rebuilding the index after the load has terminated) decreases throughput significantly.

Chapter 6: Case Studies From Wall Street

Figure 6.1: **Circumventing superlinearity.** This graph compares the four techniques that we describe for circumventing superlinearity: (a) insertion followed by a check for deletions, (b) same as (a) with an index on the table used to check for deletions, (c) inserting sales and checking for deletions in small batches, and (d) using outer join. We use the unsuccessful sales example given in the text with two configurations of the data: small (500,000 sales, 400,000 items, 400,000 customers and 10,000 stores, and 400,000 successful sales) and large (1,000,000 sales, 800,000 items, same customers and stores tables as for the small workload, and around 800,000 successful sales). The experiment is performed using SQL Server 2000 on Windows 2000. Using the small workload, the minibatch approach does not provide any benefit. Indeed, the successfulsales table is small enough so that the overhead of the successive iterations (insertions/deletions in a temporary table) is high compared to the benefit of checking the deletion condition on a reduced number of records. There is a benefit in using an index on successful sales (SQL Server 2000 uses an index nested loop in that case instead of a hash join in the absence of an index). The outer join approach is very efficient because the detection of unsuccessful sales is performed using a selection (itemtest is null, or storetest is null, or customertest is null) as opposed to a join with the successfulsales table. The large workload illustrates the benefit of the batch approach. The outer join approach still gives the best performance.
Figure 6.2: **Buffer table.** Source system transaction write to buffer tables and back-office systems read from them. If back-office systens must respond, then either build a clustering index on the buffer tables or use a second response table written by the back-office system.

Chapter 7: Troubleshooting

Chapter 8: Tuning E-Commerce Applications

Chapter 9: Celko on Data Warehouses—Techniques, Successes, and Mistakes

Chapter 10: Data Warehouse Tuning

Figure 10.1: **Bitmap.** The l_returnflag attribute takes three possible values (A, N, R), while l_linestatus takes two values (O, F). The bitmap associates a value to each record in the lineitem relation. In the figure, records are represented vertically, and the associations are marked with black rectangles: the first record (from the left) has value A in the l_returnflag attribute and value F in the l_linestatus attribute.

Figure 10.2: **Bitmaps.** We use the lineitem relation from the TPC-H benchmark for this experiment. We run a summation query, and we vary the WHERE clause so that it involves 1, 2, or 3 attributes. The attributes involved are l_returnflag, l_linenumber, and l_linestatus. The entire lineitem relation contains 600,000 records; the conditions on each attribute have approximately a 50% selectivity. The query on one attribute selects 300,000 records, the query on two attributes selects 100,000 records, and the query on three attributes selects 2000 records. To avoid crossing the application interface too often, each query is an aggregate that returns one record. We compare the performance of these aggregation queries using linear scan and bitmap indexes constructed on single attributes. This graph shows that bitmaps yield a significant performance improvement: the throughput is increased by an order of magnitude compared to a linear scan. Note that the experiment is conducted with a warm buffer, so CPU cost dominates. The graph shows that there is a slight overhead when combining several bitmaps. This experiment was conducted on Oracle 8i EE on Windows 2000.

Figure 10.3: **R-tree.** We use a synthesized relation on top of the spatial extension to Oracle 8i to compare point and range queries on two-dimensional data using an R-tree and bitmaps. We define two-dimensional points as a spatial data type, and we use spatial functions in the queries. Using bitmaps, the *X* and *Y* coordinates of each point are simply represented as integers. In both cases, the relation contains eight other attributes; there are few distinct points stored in this relation. The graph shows the response time of a point query and a range query using bitmaps and the R-tree. The R-tree index loses. This is largely due to the over-head of using spatial functions to encode simple point or range queries. This result suggests that an R-tree (or at least this implementation of an R-tree) should be used only to support far more complex spatial operations, such as overlap or nearest neighbor search, on spatial objects such as polygons. This graph comes from data from Oracle 8i running on Windows 2000.

Figure 10.4: **Materialized view creation graph.** This graph illustrates the result of the materialized view creation algorithm applied to the TPC-H schema, and two aggregate materialized views, revenue_per_nation and revenue_per_region (in TPC-H regions actually denote continents). The revenue is computed using a six-way join (the TPC-H query Q5 is used to define the aggregate materialized view). The graph shows that the revenue_per_region view should be computed from the existing revenue_per_nation view and two base relations, instead of being computed from scratch using the base relations. The algorithm suggests the order of view construction: first revenue_per_nation, then revenue_per_region.

Figure 10.5: **Foreign key dependencies.** The base tables in the TPC-H schema form a *constellation schema*. Lineitem and PartSupp are fact tables; Supplier, Part, and Order are dimension tables. The arrows on the diagram correspond to foreign key dependencies. If we consider the two aggregate materialized views balance_per_nation and balance_per_region (sum of supplier account balances s_acctbal grouped by nation or per region), foreign key dependencies link dimension tables (e.g., Supplier) and aggregate fact tables (e.g., balance_per_nation).

Figure 10.6: **Approximation on one relation.** We sample 1% and 10% of the lineitem table by selecting the top *N* records on an attribute in the fact table (here l_linenumber). That is, we are taking an approximation of a random sample. We compare the results of a query (Q1 in TPC-H) that accesses only records in the lineitem relation. The graph shows the difference between the aggregated values obtained using the base relations and our two samples. There are eight aggregate values projected out in the select clause of this query. Using the 1% sample, the difference between the aggregated value obtained using base relations and sample relations is never greater than 9%; using a 10% sample, this difference falls to around 2% in all cases but one.

Figure 10.7: **Approximation on a six-way join.** As indicated in this section, we take a sample of the lineitem table and join from there on foreign keys to obtain samples for all tables in the

Appendix B: Transaction Chopping

Appendix D: Understanding Access Plans

List of Tables

Chapter 3: Index Tuning

Chapter 7: Troubleshooting

Appendix D: Understanding Access Plans

List of Listings

List of Examples

Chapter 2: Tuning the Guts

Chapter 3: Index Tuning