# TABLE OF CONTENTS

## Chapter 1: Database Design and Performance

Excerpted from: *Microsoft® SQL Server™ 2000 Unleashed*, by Ray Rankins, Paul Bertucci, and Paul Jensen

- Basic Tenets of Designing for Performance
- Logical Database Design Issues
  - Normalization Conditions
  - Read Committed
  - Normalization Forms
  - Serializable
- Denormalizing the Database
  - Denormalization Guidelines
  - Querying the syslockinfo Table
  - Viewing Locking Activity with SQL Enterprise Manager
- Essential Denormalization Techniques
- Indexes and Performance
  - Evaluating Index Usefulness
  - Distribution Data
- Index Design Guidelines
  - Clustered Index Indications
  - Nonclustered Index Indications
  - Index Covering
  - Composite Indexes Versus Multiple Indexes
- SQL Server Index Maintenance
  - Running DBCC SHOWCONTIG
  - Extent Fragmentation
  - Minimizing Locking Contention
  - Fixing Fragmented Tables
  - Setting the Fill Factor
  - Reapplying the Fill Factor
- Updates and Performance
  - Deferred Update
  - Lock Granularity Hints
  - Update in Place
- Database File Groups and Performance
  - Optimistic Locking Using the Timestamp Datatype
- RAID Technology
  - RAID Level 0
  - RAID Level 1
  - RAID Level 10
  - RAID Level 15

## Summary

## Chapter 2: Monitoring, Optimizing, and Troubleshooting Server Performance

Excerpted from: *MCSA MCSE 70-290: Managing and Maintaining a Microsoft® Windows® Server 2003 Environment, Exam Cram™2*, by Dan Balter

- Monitoring and Analyzing System Events
  - Working with the Event Viewer
  - Working with the System Monitor
- Monitoring System Performance
  - Using System Monitoring in Real Time with Task Manager
  - Using Real Time Monitoring with System Monitor
Chapter 3: Delivering Business Intelligence and Analytics on Demand
Excerpted from: Proven Portals: Best Practices for Planning, Designing, and Developing Enterprise Portals by Dan Sullivan

Understanding Operations: Business Intelligence and Data Warehousing
Characteristics of Data Warehouses
The Data Warehouse Environment
Source Systems
Extraction, Transformation, and Load Systems
Data Warehouse Repository
Reporting Tools and Portals
Facilitating Operations: Analytic Services
Conclusion
References

Chapter 4: Configuring, Tuning, and Optimizing SQL Server Options

SQL Server Instance Architecture
Configuration Options
Fixing an Incorrect Option Setting
Setting Configuration Options with SQL Enterprise Manager
Obsolete Configuration Options
Configuration Options and Performance
Affinity Mask
Allow Update
AWE Enabled
Cost Threshold for Parallelism
Cursor Threshold
Default Language
Fill Factor
Index Create Memory (KB)
Lightweight Pooling
Locks
Max Degree of Parallelism
Max Server Memory and Min Server Memory
Chapter 5: Using Stored Procedures and Cursors


What Is a Stored Procedure?
Stored Procedure Pros and Cons
How to Create a Stored Procedure
How to Modify a Stored Procedure
Control-of-Flow Language
  The DECLARE Statement
  The GOTO Statement
  The BEGIN...END Statement
  The IF..ELSE Statement
  The WAITFOR Statement
  The RETURN Statement
  The WHILE, BREAK, and CONTINUE Statements
  The PRINT Statement
  The RAISERROR Statement
Comments
Parameters Used with Stored Procedures
  Input Parameters
  Output Parameters
Commonly Used Global Variables
How to Debug a Stored Procedure
  Transact-SQL Debugger
  Transact-SQL Debug Statements
What is a Cursor?
Creating a Cursor
  Step 1: DECLARE the Cursor
  Step 3: FETCH the Cursor
  Step 4: CLOSE or DEALLOCATE the Cursor
  Position Update and Delete
  Global Variables
Putting It All Together
  Example 1: Loop Through a Table
  Example 2: Display Object Names and Object Types
Stored Procedure and Cursor FAQ
Summary
Chapter 6: Planning an Installation or Upgrade

Development an Installation Strategy and Plan
   Step 1: Determine System Strategy and Plan
   Step 2: Select the Right Platform
   Step 3: Answer Required Questions and Understand Why They Are Important
   Step 4: Install SQL Server

Developing an Upgrade Strategy and Plan
   Upgrade/Installation Planning FAQ
   Summary
   The Upgrade Checklist

Chapter 7: Data Access
Excerpted from: *NET Enterprise Design with Visual Basic® .NET and SQL Server 2000* by Jimmy Nilsson

My Proposal for a Data Access Pattern
   How Stored Procedures Are Normally Called with ADO.NET
   My Data Access Proposal in Detail
   The General SQL Script
   External Design of My Data Access Proposal
   Internal Design of My Data Access Proposal
   Debugging
   Connection Strategy
   OUTPUT Parameters
   Tricky Cases of SQL
   Disadvantages with the Proposal

The Data Access Proposal in the Architecture
   What to Use for Carrying Data

Saying More in Fewer Words When Calling Stored Procedures
   XLM
   Other Formats
   Bitmaps
   Data Transportation Between Stored Procedures
   Server-Side Caching
   New Serve-Side Caching Potential with .NET

Dealing with Schema Changes
   Using User-Defined Data Types
   Using Too-Large CHAR and VARCHAR Parameters
   Dragging Out Parameters from Stored Procedures and Using Them as Constants
   Using Views and/or UDFs

Evaluation of My Proposal
   What’s Next
   References

Chapter 8: ADO.NET- The data access model for the .NET Compact Framework
Excerpted from: *Microsoft® .NET Compact Framework Kick Start* by Erik Rubin and Ronnie Yates

Introducing ADO.NET on the .NET Compact Framework
Caching Data with the DataSet
   Looking Inside the DataSet: DataTables, DataRows, and DataColumns
   Inserting Data into a DataSet
   Building a DataSet to Hold a Phone Book
   Extracting Data from a DataSet
   Altering Data in a DataSet
   Designing a PhoneBook Application with a DataSet
Troubleshooting Common DataSet-Related Errors

Understanding Constraints
- Adding Constraints to a Dataset
- Adding a UniqueConstraint
- Working with a UniqueConstraint by Example
- Preventing NULL Values in a DataColumn

Setting Up Autoincremented Fields
- Creating Autoincremented Field Code by Example
- Updating the PhoneBook Application with Constraints and Autoincremented Fields

Modeling Relational Data with the DataSet
- Deriving DataColumn Values with Expressions and Computed Fields
- Expressing Parent-Child Relationships in a DataSet

Creating Bindable Views of Data with a DataView
- Sorting with the DataView
- Sorting and Filtering Data in a Sample Application
- Tracking Changes in a DataRow
- Filtering with the DataView
- Adding Data into a DataView
- Using a DataView in a Sample Application: DataView_SortByRowState_AddTables

Binding Data to a Control
- Binding to a DataGrid
- Using Data Binding in a Sample Application
- Comparing the Compact DataSet with the Desktop DataSet

In Brief

Chapter 9: XQuery and SQL Server

Excerpted from: A First Look at ADO.NET and System.Xml v. 2.0 by Alex Homer, Dave Sussman, and Mark Fussell

A Very Brief Overview of the XQuery Language
- Where's the Data Located?
- FLWOR Expressions

The XQueryProcessor Class
- The Properties and Methods of the XQueryProcessor Class
- The XmlCommand Class
- Querying an XML Document with the XQueryProcessor Class
- Querying Multiple XML Documents with the XQueryProcessor Class
- Using an XQueryProcessor with an XLM View
- Using an XQueryProcessor with Multiple XML Views
- Using an XQueryProcessor with Multiple XML Views and Multiple Database Connections
- Using an XQueryProcessor with Multiple XML Views and XML Documents

The XsltProcessor Class
- Transforming XML Documents with the XsltProcessor Class

The Common Query Architecture

SQL Server as an XML Database
- The XML Data Type in SQL Server “Yukon”
- Typed and Untyped XML Data Columns
- The XML Schema Repository in SQL Server “Yukon”
- Inserting and Selecting Against an XML Data Column
- Querying and updating the Contents of an XML Data Column
- Binding Relational Data Inside XML
- Indexing an XML Data Column

Summary

Index
Chapter 10: Monitoring and Tuning SQL Server Databases

Introduction
Tools Available for Monitoring and Tuning
   Stored Procedures Used to Diagnose and Optimize
   Database Console Command (DBCC)
   Alternative Mechanisms
Optimizing the OS Configuration
   Using Performance/System Monitor
   Counter Values
   The Event Viewer
   The Window Application Log
   Query Governor Cost Limit
Optimizing SQL Server Configuration
   Current Server Activity
   Stored Procedures
Monitor Activity with the Profiler
   Defining a Profiler Trace
   Profiler Traces to Diagnose Locking
   Using Profiler Results
Trace Playback and Diagnosis
   Playback Requirements
   Performing the Replay
   Templates and Wizards for Specific Monitoring
SQL Server Optimizer
   Query Analyzer
   Proactive/Automated Optimization
Security Objects
   C2 Security
   Statement and Object Permissions
   Security Audits
   Exercises
   Review Questions
   Exam Questions
   Answers to Review Questions
   Answers to Exam Questions

Part II  Final Review

Fast Facts
SQL Server 2000 Requirements
   SQL Server 2000 Enterprise Edition
Database Design for SQL Server 2000
   ER Modeling
   Primary Keys
   Foreign Keys
   One-To-One Relationship
   One-To-Many Relationship
   Many-To-Many Relationship
   Entity Integrity
   Domain Integrity
   Referential Integrity
   Advanced Referential Integrity Options
SQL Server Data Types
Physical Database Design and Implementation
   Database Creation
   Shrinking Files
Column Properties
Check Constraints
Clustered Indexing
Nonclustered Indexing
Encryption Can Secure Definitions
Schema Binding
Indexed Views
Data Integrity Options

Querying and Modifying Data
SQL SELECT
DATEADD, DATEDIFF, DATENAME, DATEPART
Inserting Data
Inserting Data Using SELECT
Deleting Data
Updating Data

Advanced Data Retrieval and Modification
Joins
Using GROUPS BY
Using GROUP BY and HAVING
Using COMPUTE and COMPUTE BY
Using OPENROWSET and OPEN QUERY
Linked Server
XML
Data Transformations

Programming SQL Server 2000
Scripts, Batches, and Transactions
Variable Types
Global Variables
Using Cursors
Lock Isolation Levels
Designing and Managing Transactions
Statement Permissions
Object Permissions
User Roles
Fixed Roles
Application Roles

Working with Views
Views
Partitioned Views
Broken Ownership Chains

Use of Triggers
Recursive Triggers
INSTEAD OF Triggers

Stored Producers and User-Defined Functions
Stored Procedures
Cursor Status
Error Handling
User-Defined Functions

Boosting Performance with Indexes
Clustered Indexes
Nonclustered Indexes
Index Selectivity
Indexed Views
Indexed Views Requirements and Restrictions

Implementing and Understanding Replication Methodologies
Monitoring and Tuning SQL Server Databases
Tools
Simple Network Management Protocol (SNMP)
System Monitor

Study Exam Prep Tips
Learning Styles
Study Tips
Study Strategies
Macro and Micro Study Strategies
Active Study Strategies
Common-Sense Strategies
Pre-Testing Yourself
Exam Prep Tips
The MCP Exam
Exam Format
Fixed-Form
Adaptive Form
Case Study Form
Question Types
Multiple-Choice Questions
Multiple-Rating Questions
Simulation Questions
Hot Area Questions
Drag-and-Drop-Style Questions
Ordered List Questions
Tree Questions
Putting It All Together
More Exam Preparation Tips
During the Exam Session

Chapter 11: XML and SQL Server 2000
Excerpted from: XML and SQL Server: Developing Web Applications by Daniel K. Appelquist

Retrieving Data in XML Format
FOR XML
FOR XML AUTO
FOR XML EXPLICIT
Communicating with SQL Server over the Web
Under the Hood
Retrieving Data in XML Format-Continued
SQL Queries in URLs
Template Files
XPath Queries
HTTP Post Queries
XML Views
Defining XMLViews
Let SQL Server Do the Work
Working with XML Documents
OPENXML
Summary

Chapter 12: SQL Server Internals

SQL Server Memory Management
The Buffer Manager and Memory Pools
The Buffer Manager
Accessing Memory Buffers
The Checkpoint Process
The Lazywriter Process
Keeping Pages with Cache Permanently
Large Memory Support
The Log Manager
SQL Server Process Management
  SQL Server Threads
SQL Server Disk I/O
  Asynchronous I/O
  Scatter-Gather I/O
  Read Ahead Reads
  Merry-go-Round Scan
SQL Server Storage Structures
Database Files and Filegroups
  Primary Data File
  Secondary Data Files
  The Log File
  Using Filegroups
  On-Demand Disk Management
Database Pages
  Page Types
  Examining Page Content
  Data Pages
  Index Pages
  Differential Changed Map Pages
  Bulk Changed Map Pages
Tables
  Size Limits for Rows and Columns
  Heap Tables
  Clustered Tables
Indexes
  Clustered Indexes
  Nonclustered Indexes
  SQL Server Index Maintenance
Data Modification and Performance
  Inserting Data
  Deleting Rows
  Updating Rows
Summary
Introducing the VERITAS “SQL Server Performance Series”

**WHAT IS THE “SQL SERVER PERFORMANCE SERIES?”**

The VERITAS “SQL Server Performance Series” is a yearlong educational program designed by VERITAS and developed in concert with the Pearson Technology Group. The program provides expert content on topical SQL Server development and runtime concerns with a focus on application performance management. The foundation of the program is a custom e-book. The e-book consists of twelve unique chapters from twelve different books. Each month a new chapter will be released and promoted. VERITAS will also conduct a Webcast each month with one or multiple authors. Each Webcast will focus on the specific content of the chapter released that month and will provide a forum to ask industry experts SQL Server questions.

**CONTENT OF THE E-BOOK**

Unlike more traditional e-books that offer content from a single book, the e-book from the “SQL Server Performance Series” is a compilation of content from twelve different books and over twenty different authors. Each chapter will focus on a different aspect of SQL Server application development and deployment.

*Chapter 1   Database Design and Performance*
This chapter will help you understand some of the key application design issues that will ensure you have a reliable and high-performance application. These factors include logical design, physical design, choice of hardware, network bandwidth, client and server configuration, data access techniques, and application architecture.

*Chapter 2   Monitoring, Optimizing, and Troubleshooting Server Performance*
This chapter describes how Server performance can degrade over time as more users, more workstations, and more demands are placed on server resources and ultimately impact the performance of SQL Server. Windows Server 2003 offers administrators several built-in tools for monitoring, optimizing, and troubleshooting a server's performance.

*Chapter 3   Delivering Business Intelligence and Analytics on Demand*
This chapter provides an overview of data warehousing and analytic applications that support business intelligence. The design of these systems is fundamentally different from the more common transaction-oriented relational databases so we delve into some of the implementation details of data warehouses.
Chapter 4  Configuring, Tuning, and Optimizing SQL Server Options
This chapter will delve into what can be done in the SQL Server configurable options—particularly, what can be improved that SQL Server isn't automatically tuning already. By setting the values of several key SQL Server configuration parameters, you can fine-tune SQL Server to provide excellent performance and throughput.

Chapter 5  Using Stored Procedures and Cursors
This chapter will enhance your knowledge of stored procedures and cursors. As a DBA, you use stored procedures frequently. Microsoft supplies many stored procedures that you use to perform database and system maintenance. You will also find that you are frequently required to write your own stored procedures to perform specific DBA tasks for your organization or to help a group of developers solve a complex business problem.

Chapter 6  Planning an Installation or Upgrade
In this chapter, you will learn to develop plans and strategies to help you correctly install or upgrade SQL Server. Why bother with a planning stage? Why not just skip right to the installation or upgrade? SQL Server installation and upgrading is a simple process, but, by planning, you can make the correct decisions that affect the performance and operation of SQL Server before the installation.

Chapter 7  Data Access
Most distributed applications rely on a data tier: a single layer where all information resides. The data tier is one of the most critical areas of a distributed application because a disproportionate amount of the weight tends to fall on it. All too often, the data tier becomes the first bottleneck of a growing system as demand outstrips the system's ability to create connections or retrieve data.

Chapter 8  ADO.Net – The data access model for the .Net Compact Framework
This chapter will first examine the retrieval and update of data from a SQL Server database. It will also examine in detail the process of reconnecting to the data source and sending updated data back into the database.

Chapter 9  XQuery and SQL Server
In this chapter, we'll look in more detail at the XQuery language, the XQueryProcessor class, and how they can be used to perform queries over XML documents loaded into an XPathDocument2 and over SQL Server using XML views.

Chapter 10  Monitoring and Tuning SQL Server Databases
This chapter will review how SQL Server and the operating system work together to provide a productive database management environment. There are many SQL Server hooks into the operating system and OS resources available to observe the database server as it operates. Other tools allow for quick diagnosis of problems that may be affecting the server.
Chapter 11  XML and SQL Server 2000
You will come away from this chapter with concrete knowledge of how XML and SQL can be married in a powerful way at the level of a relational database server. You can apply this knowledge to your own projects, using either SQL Server 2000 or some other relational database server, such as Oracle, that might have similar features.

Chapter 12  SQL Server Internals
This chapter looks at the internal architecture as well as the storage structures in SQL Server and how the storage structures are maintained and managed. This information will help you better understand various issues associated with migrating from an earlier version or from a different RDBMS.
CHAPTER 1

Database Design and Performance

Excerpted from: Microsoft® SQL Server™ 2000 Unleashed, by Ray Rankins, Paul Bertucci, and Paul Jensen
Database Design and Performance

by Bennett McEwan

In This Chapter

- Basic Tenets of Designing for Performance 1216
- Logical Database Design Issues 1217
- Denormalizing the Database 1220
- Indexes and Performance 1227
- Index Design Guidelines 1236
- SQL Server Index Maintenance 1239
- Updates and Performance 1247
- Database File Groups and Performance 1248
- RAID Technology 1249
Various factors contribute to the optimal performance of an application. Some of these factors include logical design (rules of normalization), physical design (denormalization, indexes), choice of hardware (SMP servers), network bandwidth (LAN versus WAN), client and server configuration (memory, CPU), data access techniques (ODBC, ADO, OLEDB), and application architecture (two-tier versus n-tier). This chapter will help you understand some of the key application design issues that will ensure you have a reliable and high-performance application.

Basic Tenets of Designing for Performance

Designing for performance requires making tradeoffs. To get the best write performance out of your database, you must sacrifice read performance. Before tackling database design issues for your application, it is critical to understand your goals. Do you want faster read performance? Write performance? A more understandable design?

Following are some basic truths about physical database design for SQL Server 2000 and their performance implications:

- Keep table row sizes as small as possible. This is not about saving disk space; smaller rows mean more rows will fit on a single 8KB page, which means less physical disk reads are required to read a given quantity of rows.
- Use indexes to speed up read access, but…
- The more indexes that a table has, the longer it takes to insert, update, and delete rows from that table.
- Using triggers to perform any kind of work during an insert, update, or delete will exact a performance toll, and decrease concurrency by lengthening transaction duration.
- Implementing declarative referential integrity (primary and foreign keys) helps maintain data integrity, but enforcing foreign key constraints requires extra lookups on the primary key table to ensure existence.
- Using ON DELETE CASCADE referential integrity constraints likewise helps maintain data integrity, but requires extra work on the server’s part.

Keeping tables as narrow as possible—that is, ensuring that the row size is as small as possible—is one of the most important things you can do to ensure your database performs well. To keep your tables narrow, choose column data types with size in mind. Don’t use an int datatype if a tinyint will do. If you have zero-to-one relationships in
your tables, consider vertically partitioning your table. (See “Vertical Partitioning” under the “Denormalizing the Database” section for details on this scenario.)

Cascading deletes (and updates) causes extra lookups to be done whenever a delete runs against the parent table. In many cases, the optimizer will use worktables to resolve delete and update queries. Enforcing these constraints manually, from within stored procedures, for example, can give better performance. This is not a wholehearted endorsement against referential integrity constraints. In most cases, the extra performance hit is worth the saved aggravation of coding everything by hand. However, you should be aware of the cost of this convenience.

Logical Database Design Issues

A good database design is fundamental to the success of any application. Logical database design for relational databases follows a set of rules called rules of normalization. As a result of normalization, you create a data model that is usually, but not necessarily, translated into a physical data model. A logical database design does not depend on the relational database you intend to use. The same data model can be applied to Oracle, Sybase, SQL Server, or any other relational database. On the other hand, a physical data model makes extensive use of the features of the underlying database engine to yield optimal performance for the application. Physical models are much less portable.

Tip

If portability is a big concern to you, consider using a third-party data modeling tool, such as Erwin. These tools have features that make it easier to migrate your physical data model to a different platform. Of course, this will just get you started; to get the best performance out of your design, you will need to tweak the physical design for the platform you have chosen.

Normalization Conditions

Any database designer must address two fundamental issues:

• Designing the database in a simple, understandable way that is maintainable and makes sense to its developers and users
• Fetching and saving data with the fastest response time, resulting in high performance
Normalization is a technique used by relational databases to organize data across many tables so that related data is kept together based on certain guidelines. Normalization results in controlled redundancy of data; therefore, it provides a good balance between disk space usage and performance. Normalization helps people understand the relationships between data, and enforces rules that ensure the data is meaningful.

Tip
Normalization rules exist, among other reasons, to make it easier for people to understand the relationships between the data. But a perfectly normalized database won’t perform well under certain circumstances, and it can be difficult to understand. There are good reasons to deviate from a perfectly normalized database.

Normalization Forms
Five normalization forms exist. If you follow the rules for the first rule of normalization, your database can be described as “in first normal form.” This is represented by the symbol 1NF for first normal form, 2NF for second normal form, and so on.

Each rule of normalization depends on the previous rule for successful implementation, so to be in second normal form (2NF), your database must also follow the rules for first normal form.

A typical relational database used in a business environment falls somewhere between second and third normal form. It is rare to progress past the third normal form because fourth and fifth normal form are more academic than practical in real-world environments.

Following is a brief description of the first three rules of normalization.

First Normal Form
The first rule of normalization requires removing repeating data values and specifies that no two rows can be identical in a table. This means that each table must have a logical primary key that uniquely identifies a row in the table.

Second Normal Form
A table is in 2NF if it conforms to the first normal form and all nonkey attributes of the table are fully dependent on the entire primary key. If the primary key consists of
multiple columns, then nonkey columns should depend on the entire key and not just on a part of the key.

**Third Normal Form**

A table is in 3NF if it already conforms to the first two normal forms and none of the nonkey columns are dependent on any other nonkey columns. All such attributes should be removed from the table.

Following is an example that comes up often during database architecture. Suppose that the emp table has four columns: EmployeeID (the primary key), salary, bonus, and total_salary, where total_salary = salary + bonus. Existence of the total_salary column in the table violates the third normal form because a nonkey column (total_salary) is dependent on two other nonkey columns (salary and bonus). Therefore, to conform to the third rule of normalization, you must remove the total_salary column from the emp table.

**Benefits of Normalization**

The following are the major advantages of normalization:

- Because information is logically kept together, normalization provides a better overall understanding of the system.
- Because of controlled redundancy of data, normalization can result in fast table scans and searches (less physical data has to be processed).
- Because tables are smaller with normalization, index creation and data sorts are much faster.
- With less redundant data, it is easier to maintain referential integrity for the system.
- Normalization results in narrower tables. Because you can store more rows per page, more rows can be read and cached for each I/O performed on the table. This results in better I/O performance.

**Drawbacks of Normalization**

One result of normalization is that data is stored in multiple tables. To retrieve or modify information, you usually have to establish joins across multiple tables. Joins are expensive from an I/O standpoint. Multitable joins can have an adverse impact on the performance of the system. The following sections discuss some of the denormalization techniques that you can use to improve the performance of the system.
Denormalizing the Database

After a database has been normalized to the third form, database designers intentionally backtrack from normalization to improve the performance of the system. This technique of rolling back from normalization is called denormalization. Denormalization allows you to keep redundant data in the system, reducing the number of tables in your schema and reducing the number of joins to retrieve data.

Tip

A wise, old database architect I knew passed on a wise, old adage to me: “Normalize ‘til it hurts, denormalize ‘til it works.” To put this into use, try to put your database in third normal form. After this is done, when you’re ready to implement the physical structure, drop back from third normal form where excessive table joins are hurting performance.

Denormalization Guidelines

When should you denormalize a database? Consider the following points first:

• Be sure you have a good overall understanding of the logical design of the system. This knowledge helps in determining how other parts of the application are going to be affected when you change one part of the system.

• Don’t make an attempt to denormalize the entire database at once. Instead, focus on the specific areas and queries that are accessed most frequently and are suffering from performance problems.

• Understand the types of transactions and the volume of data associated with specific areas of the application that is having performance problems. You can resolve many such issues by tuning the queries without denormalizing the tables.

Tip

Duplicate data is more helpful when the data does not change very much, such as in data warehouses. If the data changes often, keeping all “copies” of the data in sync can create significant performance overhead, including long transactions and excessive write operations.
• Determine whether you need virtual (computed) columns. Virtual columns can be computed from other columns of the table. Although this strictly violates third normal form, computed columns can provide a decent compromise because they do not actually store another copy of the data in the same table.

• Understand data integrity issues. With more redundant data in the system, maintaining data integrity is more difficult and data modifications are slower.

• Understand storage techniques for the data. Using RAID and SQL Server file groups improves performance without denormalization.

• Determine the frequency at which data changes. If the data is changing too often, the cost of maintaining referential integrity might outweigh the benefits provided by redundant data.

Tip

If you are experiencing severe performance problems, denormalization should not be the first step you take to rectify the problem. Identify specific issues that are causing performance problems. Usually, you will discover factors such as poorly written queries or poorly configured hardware. You should try to fix such issues before taking steps to denormalize database tables.

Essential Denormalization Techniques

You can employ various methods to denormalize a database table and achieve desired performance goals. Some of the useful techniques used for denormalization include the following:

• Keeping redundant data and summary data
• Using virtual columns
• Performing horizontal data partitioning
• Performing vertical data partitioning

Redundant Data

Joins are inherently expensive in a relational database from an I/O standpoint. To avoid common joins, add redundancy to the table by keeping exact copies of the data in multiple tables. The following example demonstrates this point. The example shows a three-table join to get authors’ names and the titles of the books written by each author:
You can improve join performance in this example by adding a title column to the titleauthor table. This will eliminate the join from the title table. Here is the revised query:

```sql
select a.au_lname,
       a.au_fname,
       b.title
from   authors a,
        titleauthor b
where  a.au_id = b.au_id
       and b.title_id = c.title_id
order by au_lname, au_fname
```

As you can see, the column title is now redundantly stored in two places: the title table and the titleauthor table. It is obvious that with more redundant data in the system, maintaining referential integrity is more difficult. For example, if the title of the book changes in the title table, to preserve referential integrity, you must also change the title column value in titleauthor to reflect the correct value. You could use SQL Server triggers to maintain referential integrity, but recognize that update performance would suffer dramatically.

### Computed Columns

A number of queries calculate aggregate values derived from one or more columns of a table. Such computations can sometimes be CPU intensive and can have an adverse impact on performance if they are performed frequently. One of the techniques to handle such situations is to create an additional column that stores the computed value. Such columns are called virtual columns or contrived columns. Starting with SQL Server 7, computed columns are natively supported. You can specify such columns during `create table` or `alter table` commands. The following example demonstrates the use of computed columns:

```sql
create table emp (  
    empid int not null primary key,
    salary money not null,
    bonus money not null default 0,
    total_salary as ( salary+bonus )
)  
```
Virtual columns are not physically stored in SQL Server tables. SQL Server internally maintains a column \texttt{iscomputed} in the system table \texttt{syscolumns} to determine whether a column is computed. The value of the virtual column is calculated at the time the query is run. Computed columns cannot pull data from more than one table, however, so if this is required, you must create a physical column and use stored procedures or triggers to maintain its value.

In SQL 2000, computed columns can participate in joins to other tables, and they can be indexed. Creating an index that contains a computed column creates a physical copy of the computed column in the index tree. Whenever a base column participating in the computed column changes, the index must also be updated.

**Summary Data**

Summary data is most helpful in a decision support environment. To satisfy reporting requirements, calculate sums, row counts, or other summary information and store it in a separate table. You can create summary data in a number of ways:

- **Real time**—Every time your base data is modified, recalculate the summary data using the base data as a source.
- **Real time incremental**—Every time your base data is modified, recalculate the summary data using the old summary value and the new data. This is more complex, but it could save time if the increments are relatively small compared to the entire data set.
- **Delayed**—Use a scheduled job to recalculate summary data on a regular basis. This is the only method you should use in an OLTP system.

**Horizontal Data Partitioning**

As tables grow larger, data access time also tends to increase. For queries that need to perform table scans, the query time is proportional to the number of rows in the table. Even when you have proper indexes on such tables, access time slows as the depth of the index trees increase.
The solution is splitting the table into multiple tables such that each table has the same table structure as the original one but stores a different set of data. Figure 38.1 shows a billing table with 90 million records. You can split this table into 12 monthly tables (each with an identical table structure) to store billing records for each month.

**Figure 38.1**
*Horizontal partitioning of data.*

You should carefully weigh the options when performing horizontal splitting. Although a query that only needs data from a single month gets much faster, other queries that need a full year’s worth of data become more complex. Also, queries that are self-referencing do not benefit much from horizontal partitioning. For example, the business logic might dictate that each time you add a new billing record to the billing table, you need to check any outstanding account balance for previous billing dates. In such cases, before you do an insert in the current monthly billing table, you must check the data for all the other months to find any outstanding balance.

**Tip**

Horizontal splitting of data is useful where a subset of data might see more activity than the rest of the data. For example, in a healthcare provider setting, 98% of the patients are inpatients and only 2% are outpatients. In spite of the small percentage involved, the system for outpatient records sees a lot of activity. In this scenario, it makes sense to split the patient table into two tables, one for the inpatients and one for the outpatients.
When splitting tables horizontally, you must perform some analysis to determine the optimal way to split the table. Try to find a logical dimension along which to split the data. The best choice will take into account the way your users use your data. In the previous example, date was mentioned as the optimal split candidate. However, if your users often did ad hoc queries against the billing table for a full year's worth of data, they would be unhappy with your choice to split that data among 12 different tables. Perhaps a customer type or other attribute would be more useful.

**Vertical Data Partitioning**

As you know, a database in SQL Server consists of 8KB pages, and a row cannot span across multiple pages. Therefore, the total number of rows on a page depends on the width of a table. This means the wider the table, the fewer the number of rows per page. You can achieve significant performance gains by reducing the number of I/Os on the table. Vertical splitting is a method of reducing the width of a table by splitting the columns of a table into multiple tables. Usually, all frequently used columns are kept in one table and others are kept in the other table. This way, more records can be accommodated per page, fewer I/Os are generated, and more data can be cached into SQL Server memory. Figure 38.2 illustrates a vertically partitioned table. The frequently accessed columns of the authors table are stored in the `author_primary` table, whereas less frequently used tables are stored in `author_secondary` table.
Performance Implications of Zero to One Relationships

Suppose that one of the development managers in your company, Bob, approaches you to discuss some database schema changes. He is one of several managers whose groups all use the central User table in your database. Bob’s application makes use of about 5% of the users in the User table. Bob has a requirement to track five yes/no/undecided flags associated with those users. He would like you to add five, one-character columns to the user table to track this information. What do you tell Bob?

Bob has a classic zero-to-one problem. He has some data he needs to track, but it applies to only a small subset of the data in the table. You can approach this problem in one of three ways:

- **Option one:** Add the columns to the User table—95% of your users will have NULL values in those columns, and the table will become wider for everybody.
- **Option two:** Create a new table with a vertical partition of the User table—The new table will contain the User primary key and Bob’s five flags. 95% of your users still have NULL data in the new table, but the User table is safe from the effects of this. Because other groups don’t need to use the new partition table, this is a nice compromise.
- **Option three:** Create a new vertically partitioned table as in option two, but populate it only with rows that have at least one non-NULL value for the columns in the new partition—This is great for database performance, and searches in the new table will be wonderfully fast. The only drawback to this is that Bob’s developers will have to add additional logic to their applications to determine if a row exists during updates. Bob’s folks will need to use an outer join to the table to cover the possibility that a row doesn’t exist.

Depending on the goals of your project, any one of these options is appropriate. The first option is simple and is the easiest to code for and understand. The second option is a good compromise between performance and simplicity. The third option gives the best

---

**Tip**

Make the decision to split data very carefully, especially when the system is already in production. Changing the data structure might have a system-wide impact on a large number of queries that reference the old definition of the object. In such cases, to minimize risks, SQL Server views can be effective in providing vertical partitioning of data.
performance in certain circumstances, but impacts performance in certain other situations and definitely causes more coding work to be done.

Indexes and Performance

I/O, or the reading and writing of data pages to and from the disk, is the slowest part of a database application. Creating indexes on database tables allows SQL Server to access data with a reduced number of I/Os. Defining useful indexes during the logical and physical data modeling step is crucial.

The SQL Server optimizer relies heavily on index key distribution and density. The optimizer in SQL Server can use multiple indexes in a query (through index intersection) to reduce the number of I/Os to retrieve information. In the absence of indexes, the optimizer will choose to perform a table scan, or it will create a dynamic index. Dynamic indexes are temporary indexes that are created for the duration of the query and immediately dropped. In rare cases, it is cheaper to go through the steps of indexing the table temporarily and immediately discarding that index than it would be to repeatedly scan the table.

Although indexes provide a means for faster access to data, they slow down data modification statements. Therefore, it is important to choose indexes carefully for a good balance between data search and data modification performance.

The application environment usually governs the choice of indexes. If the application is mainly Online Transaction Processing (OLTP) with transactions requiring fast response time, creating too many indexes will have an adverse impact on performance. On the other hand, if the application is a decision support system (DSS) with few transactions performing data modifications, create as many indexes as necessary to support queries.

In DSS environments, the constraint on the number of indexes to create is the load window. If you have, for example, eight hours to load data, and your load times are taking 40 minutes, adding more indexes and bumping the load time to 2 hours is a prudent choice.

The following sections explain how to evaluate useful indexes and understand index density and key distribution. More information on how indexes work is available in Chapter 31, “Indexes and Performance.”

Evaluating Index Usefulness

SQL Server provides indexes for faster access to data and to enforce the uniqueness of the data in the database tables. Creating the appropriate indexes for a database is one of
the most important aspects of database design. To define the usefulness of an index for solving a query, refer to the index’s *selectivity*. The selectivity of an index is a percentage that can be defined as follows:

\[
\text{Selectivity} = \frac{\text{Total number of rows uniquely identified by the key}}{\text{Total number of rows in the table}}
\]

If the selectivity is high—that is, a large number of rows can be uniquely identified by the key—then the index is more likely to be useful to the optimizer for singleton selects and resolving joins. Based on the selectivity of data, the SQL Server optimizer decides whether to use indexes in a query. The higher the selectivity, the faster SQL Server can get to a specific row.

As an example, perhaps you are evaluating useful indexes on the authors table. Based on the needs of your users, you know that most of the queries access the table either by the author’s last name or by state. Because a large number of concurrent users modify data in this table, you are allowed to choose only one index: author’s last name or state. Which one would you choose? You need to perform some analysis to see which one is a more useful index. First determine the selectivity based on the author’s last name:

```sql
select au_lname, Total = count(*)
from authors
group by au_lname
```

<table>
<thead>
<tr>
<th>au_lname</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bennet</td>
<td>1</td>
</tr>
<tr>
<td>Blotchet-Halls</td>
<td>1</td>
</tr>
<tr>
<td>Carson</td>
<td>1</td>
</tr>
<tr>
<td>DeFrance</td>
<td>1</td>
</tr>
<tr>
<td>Dull</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>1</td>
</tr>
<tr>
<td>Greene</td>
<td>1</td>
</tr>
<tr>
<td>Gringlesby</td>
<td>1</td>
</tr>
<tr>
<td>Hunter</td>
<td>1</td>
</tr>
<tr>
<td>Karsen</td>
<td>1</td>
</tr>
<tr>
<td>Locksley</td>
<td>1</td>
</tr>
<tr>
<td>MacFeather</td>
<td>1</td>
</tr>
<tr>
<td>McBadden</td>
<td>1</td>
</tr>
<tr>
<td>O’Leary</td>
<td>1</td>
</tr>
<tr>
<td>Panteley</td>
<td>1</td>
</tr>
<tr>
<td>Ringer</td>
<td>2</td>
</tr>
<tr>
<td>Smith</td>
<td>1</td>
</tr>
<tr>
<td>Straight</td>
<td>1</td>
</tr>
<tr>
<td>Stringer</td>
<td>1</td>
</tr>
<tr>
<td>White</td>
<td>1</td>
</tr>
<tr>
<td>Yokomoto</td>
<td>1</td>
</tr>
<tr>
<td>del Castillo</td>
<td>1</td>
</tr>
</tbody>
</table>
As you can see, the key can uniquely identify all the authors except Ringer. Selectivity is high for all the key values in this table. Therefore, a key on the au_lname column is a good candidate for an index to support lookups on last name.

Now, consider the selectivity based on the state column:

```
select state, Total = count(*)
from authors
group by state
```

<table>
<thead>
<tr>
<th>state</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>15</td>
</tr>
<tr>
<td>IN</td>
<td>1</td>
</tr>
<tr>
<td>KS</td>
<td>1</td>
</tr>
<tr>
<td>MD</td>
<td>1</td>
</tr>
<tr>
<td>MI</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
</tr>
<tr>
<td>TN</td>
<td>1</td>
</tr>
<tr>
<td>UT</td>
<td>2</td>
</tr>
</tbody>
</table>

Now determine the selectivity ratio for the state of California:

```
Selectivity ratio = 15 / 23 = ~65%
```

As you can see, the selectivity on the state column is poor for the state of California; therefore, state is not as good a candidate for an indexed column. For other states, this would be a helpful index. The selectivity ratio is high for Indiana and Kansas. The optimizer will know this based on the distribution data maintained on the indexes.

**Distribution Data**

Each physical index in your database has a set of distribution data associated with it. If you took every key value in your index, plotted it in ascending order on a graph, and then sampled this data by selecting 100 or 200 points on the curve, the values of those points would be distribution data. Knowing where on the curve key points are located, together with the selectivity of the index, allows the optimizer to estimate how many rows are between any two values.

Distribution data is an important piece of information for the query optimizer. Because distribution data isn’t as relevant with small numbers of rows, as in the pubs database, consider a different kind of database. For this exercise, imagine two different index cases in a larger table, and see how the distribution data would look.

The selectivity of a key is an important factor that determines whether an index will be used in a query. SQL Server stores the selectivity and the distribution values of the key in pages pointed to by the STATBL0B column on the sysindexes system table. Based on the
distribution values, the query optimizer decides which index to use. To see statistical
information about the distribution page, use the DBCC SHOW_STATISTICS command,
which returns the selectivity and density of an indexed column on a table. Following is
the syntax for this command:

```
DBCC SHOW_STATISTICS (tablename, target)
```

tablename is the name of the table (not the ID), and target is the name of the index or
statistics collection.

Imagine a grocery store with millions of rows in an order detail table. The order detail
table has one row for every item that the grocery store sells. This table has two indexed
columns at which you are going to look. The first is an OrderDetailID column, which is
the primary key of the table and is therefore as selective as you can get (it’s unique). The
second is ScanDT, a datetime column that identifies when the item was scanned through
the checkout. The table also has an ItemID column that is not indexed. For the purposes
of this discussion, you are looking at just one day’s worth of sales for the store.

Following is the show_statistics output for the primary key index:

```
Statistics for INDEX 'PK_OrderDetail'.
Updated               Rows    Rows Sampled   Steps  Density        Average key
-------------------- -------- ------------- ------ -------------- -------------
Sep 9 2001 12:44PM  1326958  1326958        3      7.5360333E-7   12.0
(1 row(s) affected)
```

```
All density              Average Length           Columns
------------------------ ------------------------ ----------------------
 7.5360333E-7             4.0                      OrderDetailID
 7.5360333E-7             12.0                     OrderDetailID, ScanDT
(2 row(s) affected)
```

```
RANGE_HI_KEY   RANGE_ROWS    EQ_ROWS        DISTINCT_            AVG_RANGE_
-------------- ------------- -------------- -------------------- ---------------
   1              0.0     1.0                  0               0.0
1327323        1326955.0     1.0            1326955               1.0
1327324        0.0     1.0                  0               0.0
(3 row(s) affected)
```
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Statistics for INDEX 'PK_OrderDetail'.

Updated  
---------  
Sep 9 2001 12:44PM  
1326958  
1326958  
3  
7.536033E-7  
12.0  

(1 row(s) affected)

All density             Average Length            Columns             
------------------------ ------------------------ ----------------------  
7.536033E-7             4.0                      OrderDetailID       
7.536033E-7             12.0                     OrderDetailID, ScanDT 

(2 row(s) affected)

RANGE_HI_KEY RANGES ROWS EQ_ROWS DISTINCT_RANGE_ROWS AVG_RANGE_ROWS 
----------------- --------------- ------ ------------------- ---------------  
1 0.0 1.0 0 0.0 
1327323 1326955.0 1.0 1326955 1.0 
1327324 0.0 1.0 0 0.0 

(3 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

“Updated” tells you when the distribution statistics for this index were last acquired. If much of the data has changed since that time, your statistics are out of date and should be refreshed so the optimizer can take advantage of new information. If autostats are enabled (they are by default), the indexes should be updated if a certain number of rows have changed in the table since the last time statistics were run.

Note

Auto statistics gathering will run UPDATE STATISTICS on a table automatically whenever a "significant" number of rows change in the table. Following is the real set of rules that are used by autostats. In this list, a modification is any insert or delete on the table, or an update that modifies a column that participates in the statistics.

• Tables containing less than 500 rows are updated every 500 modifications.
• Tables with more than 500 rows are updated after 500 + (rows * 0.2) modifications. For a one million row table, autostats would run after 200,500 modifications.
• The one exception to this is if a table exists in tempdb, and it has less than six rows. For these tables, autostats runs every six modifications.
Rows reports the number of rows in the table at the time statistics were run, although if a sample was used instead of a fullscan, this number could be slightly inaccurate. Auto stats will run samples on large tables.

Rows Sampled shows how many sample values were scanned from the table to construct the statistics. This sample was taken with FULLSCAN.

Steps shows how many sample values were retained to make up the statistics. The server recognized that this was a very even distribution, so it kept only three distribution steps.

Average Key Length reports the average size of the index keys. This might change over time in indexes that contain variable length keys, such as varchar() columns. Because this is a nonclustered index with a 4-byte key, and the clustered index has an 8-byte key, the total key length will be 12. Remember that a nonclustered index uses the clustered index keys as part of its key value to locate the row in the table.

Density describes the uniqueness of the index. It is the inverse of all unique values of the key. An index with all duplicate values would have a density of 1 (not a very helpful index!). Note that the density value on the first row reports the key density for the first part of the key. The All Density figures beneath that report density for different portions of the key. For composite keys, statistics will show densities for each useful part of the index. Useful parts are ascending orders of the key starting from the front and working toward the end, so a four-part nonclustered key would track density for a, a+b, a+b+c, a+b+c+d, and finally the entire key plus the clustered key.

The density of an index key is calculated by the following formula:

\[ \text{Key density} = \frac{1.00}{\text{rows in the table}} \]

The density for the OrderDetailID can be calculated like this:

```sql
SELECT Density = 1.00 / (
    SELECT COUNT(DISTINCT OrderDetailID) FROM OrderDetail
)
```
go
Density
.................
.00000075360333

Beneath this is each of the distribution steps that the server has selected. The index at which you are looking is actually an identity column and can be described as a monotonically increasing key. This means that the key increases by a set value for every additional row.

Range_Hi_Key shows the top of the range for this step. The low range is the step preceding it, or the first key value in the case of the first distribution step.

Range_Rows shows how many rows from this range were sampled. Note that unless the FULLSCAN option was used to generate statistics, this number will not be the same as the actual number of rows between the two index keys. The algorithm that chooses a sampling method introduces some strange rounding, so although this number should be an integer, the results you see could have fractional rows.

EQ_ROWS shows how many key values are exactly equal to the distribution step. In cases where a query matches one of the distribution steps, the server has precise information about exactly how many rows will match the query, and can make decisions about index selectivity more appropriately than if it simply estimates the number of rows. It is much more common for query results to fall between two distribution values.

Distinct_range_rows shows how many distinct values are within the sampled range. Because this is a unique index, the values should be equal to the Range_Rows value.

Avg_Range_Rows complements distinct_range_rows, showing how many duplicate values exist in the distribution step. For this index, the value should be exactly 1, and is based on Range_Rows/Distinct Range_Rows.

If you graphed the distribution values for this index, you would see a straight line rising at a 45 degree angle. Each key value is exactly one higher than the last, with a completely predictable distribution.

Following are the distribution statistics for ScanDT, the clustered index. The Scan Date tracks when an item passes over the grocery scanner. The grocery store is busier at certain times of the day (4–8 p.m. is busier than midnight–4 a.m.) and the distribution values reflect that:
Statistics for INDEX 'xOrderDetail_ScanDT'.
Updated             Rows     Rows Sampled  Steps  Density      Average key length
------------------- -------- ------------- ------ ------------- -------------------
Sep 9 2001 12:44PM 1326958 1326958       21     1.160693E-5 8.0
(1 row(s) affected)

All density              Average Length           Columns
------------------------ ------------------------ ---------------
1.1605775E-5             8.0                      ScanDT
(1 row(s) affected)

RANGE_HI_KEY               RANGE_     EQ_ROWS    DISTINCT_       AVG_
ROWS                  RANGE_ROWS      RANGE_ROWS
------------------- ------ ---------- ----------- -------------- ---------------
2001-08-15 00:00:00.000        0.0     6.0           0              0.0
2001-08-15 01:38:59.000    32797.0    14.0        5913              5.5465922
2001-08-15 02:06:06.997     8165.0    11.0        1605              5.0872273
2001-08-15 02:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 04:04:12.000    13765.0    13.0        2709              5.4072746
2001-08-15 04:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 05:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 05:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 06:04:12.000     1354.0    46.0         122              8.2382363
2001-08-15 06:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 07:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 07:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 08:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 08:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 09:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 09:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 10:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 10:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 11:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 11:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 12:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 12:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 13:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 13:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 14:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 14:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 15:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 15:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 16:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 16:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 17:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 17:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 18:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 18:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 19:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 19:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 20:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 20:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 21:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 21:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 22:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 22:34:51.000     1354.0    46.0         122              8.2382363
2001-08-15 23:04:51.000     1354.0    46.0         122              8.2382363
2001-08-15 23:34:51.000     1354.0    46.0         122              8.2382363

SQL Server Internals and Performance Tuning
PART V
<table>
<thead>
<tr>
<th>Date/Time</th>
<th>Range Hi Key</th>
<th>Range Rows</th>
<th>EQ Rows</th>
<th>Distinct</th>
<th>AVG RANGE Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-08-15 20:00:06.997</td>
<td>6475.0</td>
<td>19.0</td>
<td>184</td>
<td>35.190216</td>
<td></td>
</tr>
<tr>
<td>2001-08-15 21:55:22.997</td>
<td>190750.0</td>
<td>39.0</td>
<td>6914</td>
<td>27.584961</td>
<td></td>
</tr>
<tr>
<td>2001-08-15 22:01:14.000</td>
<td>8189.0</td>
<td>8.0</td>
<td>350</td>
<td>23.397142</td>
<td></td>
</tr>
<tr>
<td>2001-08-15 23:59:58.997</td>
<td>65768.0</td>
<td>5.0</td>
<td>7123</td>
<td>9.2318926</td>
<td></td>
</tr>
<tr>
<td>2001-08-16 00:00:00.000</td>
<td>0.0</td>
<td>5.0</td>
<td>0</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

(21 rows affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.
Activity from around 4 p.m. (16:01) to 8 p.m. (19:57) accounts for about half the rows in the table. If you wrote a query that asked for data between the hours of 4–6 p.m., for example, the server would know that it would be likely to match a lot of rows, and would probably opt to use a table scan rather than an index. On the other hand, if you looked for data between 2–4 a.m. (step 3 and 4), the index would be much more likely to be used because the distribution of data shows the number of rows in that range to be low.

**Note**

What actually happens when you execute `sp_autostats` or use the `NORECOMPUTE` option in the `UPDATE STATISTICS` command to turn off auto update statistics for a specific index or a table? SQL Server internally sets a bit in the status column of the `sysindexes` table to inform the internal SQL Server process not to update the key distribution statistics for the indexes that are turned off using these commands. To turn on auto-update, either run `UPDATE STATISTICS` without the `NORECOMPUTE` option or pass the `ON` parameter to the `sp_autostats` system stored procedure.

Much more activity is going through the checkout between the hours of 5–7 p.m., when people are getting out of work and buying dinner, than between the hours of 2–4 a.m.

**Index Design Guidelines**

SQL Server indexes are mostly transparent to users with a few exceptions where users can pass optimizer hints. Based on the key distribution values, the SQL Server optimizer chooses the index that is least expensive from an I/O standpoint. Query optimization is covered in detail in Chapter 32, “Understanding Query Optimization.” Following are some of the main guidelines that will help you build useful indexes in your environment.

**Clustered Index Indications**

The following guidelines will help you decide when to choose a clustered index:

- Columns in that index have a few unique values—Because the data is physically sorted, all the duplicate values are kept together. Any query that tries to fetch records against such keys will find all the values with a minimum number of I/Os.
- Columns that are often specified in the order by clause—Because the data is already sorted, SQL Server does not have to re-sort the data.
• Columns that are often searched for a range of values—SQL Server can use this index to locate the starting and ending page for the specified range, and use a fast index scan to retrieve all the results, rather than many index seek operations requiring a complete traversal of the tree for each row.

• Columns that are most frequently used in the join clause—Clustered indexes tend to be one I/O faster than a nonclustered index, which on single row lookups is mostly insignificant, but on large multirow joins can add up quickly.

• Queries that might return large result sets with adjacent key values—A clustered index stores the data in the table in the order of the key values that define the index. If you have adjacent key values returned, defining a clustered index on the key guarantees that many rows can be returned without needing to jump around to different physical locations on the disk.

Tip

Clustered index key selection affects the size of the nonclustered indexes. When a nonclustered index is created, the clustered key is used to identify the row to which the nonclustered index key refers. If you have a wide clustered key, your nonclustered indexes will still be huge, and performance will suffer dramatically.

I once had a client with a table that had three single integer nonclustered indexes. The nonclustered indexes alone had four byte keys. However, the clustered key was 60 bytes wide! This caused the nonclustered indexes to be a total of 64 bytes per key, at a relative performance loss of 16x for all index scans.

Nonclustered Index Indications

SQL Server 2000 allows you to create a maximum of 249 nonclustered indexes on a table. Automatically generated statistics (those indexes whose names start with "_WA_"") also count against this limit. Always keep in mind that as you add more indexes to the system, database modification statements get slower. The following guidelines will help you choose the right nonclustered index in your environment:

• Columns that have a large number of unique values or queries that return small result sets—For singleton selects (a single row lookup), nonclustered indexes are as fast as, or one I/O slower than, a clustered index. In the larger scheme of things, one 9 millisecond (ms) lookup will not be a significant delay. Nonclustered indexes are great for primary key enforcement.
• Columns that support table joins—Because you are limited to one clustered index, and it is likely you will need to support joins on more than one column, nonclustered indexes are helpful for supporting joins to related tables.
• Queries that use indexed columns in the where and the order by clauses—If the optimizer selects a nonclustered index, the order of the key values in the B-tree will be the same as the columns specified in the order by clause. In such cases, SQL Server can avoid creating an internal temporary worktable for a data sort.

Index Covering

Index covering is a situation in which all the columns in the select and where clauses of the query are also part of the nonclustered index. This results in faster retrieval of data because all the information can come directly from the index page, and SQL Server avoids trips to the data pages. Pairing this with a separate index file group will give you the fastest possible access to your data.

This example uses a nonclustered index on the au_lname and au_fname columns of the authors table:

```sql
select au_lname, au_fname
from authors
where au_lname like "M%"
```

Many other queries that use an aggregate clause (such as min, max, avg) or check for existence of a criteria (exists()) also benefit from index covering. The following query is an example of index covering using aggregates:

```sql
select count(*) from authors where au_lname like "m%"
```

Tip

When writing a query to count the number of rows that match a particular set of criteria, always use count(*) rather than a count of a column name. This allows the optimizer to pick the narrowest (fastest) index to satisfy the count.

Composite Indexes Versus Multiple Indexes

As your key becomes wider, the selectivity of the key becomes higher as well. It might appear as if creating wide indexes should result in better performance. This is generally not true. The wider the key, the fewer rows that SQL Server stores on the index pages, eventually resulting in a deeper index B-tree.
To get better performance from queries, instead of creating a few wide indexes, create multiple narrower indexes. The advantage here is that with smaller keys, the query optimizer can quickly scan through multiple indexes to create the most efficient access plan. Also, with more indexes, the optimizer can choose from various alternatives. If you are deciding on a wide key, check the distribution of each member of the composite key individually. If the selectivity on the individual columns is high, you might want to break up the index into multiple indexes. If the selectivity of individual columns is low but is high for combined columns, it makes sense to have wider keys on a table. To get to the right combination, populate your table with real-world data, experiment with creating multiple indexes, and check the distribution of each column. Based on the distribution steps and index density, you can make a decision that works best for your environment.

You can also make use of the Query Analyzer’s showplan feature. Use Ctrl-L to generate a graphical showplan of your query in the Query Analyzer, and see if your indexes are being used. Of course, because distribution statistics factor so heavily into the selection of an index, you must have production data to do meaningful analysis in this way. Attempting to analyze queries using small result sets is pointless because the indexes are less likely to be helpful in small tables.

**SQL Server Index Maintenance**

SQL Server indexes are self-maintained, which means that any time a data modification (insert, update, or delete) takes place on a table, the index B-tree is automatically balanced. This does not solve the problem of fragmentation in the data and index pages. Fragmentation on an index page can happen for the following reasons:

- As more records are added to a table, space is used on the data page and on the index page. As a result, the page eventually becomes completely full. If another insert takes place on that page, because the new row has no more room, SQL Server might perform a 50/50 page split. Each new page becomes 50% full. Figure 38.3 illustrates page splitting due to inserts.

- Frequent update statements can cause fragmentation in the database at the data and index page level. Figure 38.4 depicts an update scenario, where as a result of an update statement, most of the rows move to a different page (a deferred update). The page that used to be 80% full becomes only 10% full. The index page will be balanced automatically, but this page will have much unused space. A number of these deferred updates will lead to heavy fragmentation due to sparse pages.
Figure 38.5 shows a scenario in which fragmentation can take place because of delete statements. As you can see from the figure, a delete statement causes a page to become only 10% full. This page will remain allocated to the extent even if it has only a single row.

Running DBCC SHOWCONTIG

DBCC SHOWCONTIG will give you information about how badly fragmented your indexes are. DBCC SHOWCONTIG takes two arguments: the table ID and the index ID. Following is a sample for the OrderDetail table from earlier in the chapter. Remember that the OrderDetail table has about 1.3 million rows and two indexes, PK_OrderDetail (non-clustered, enforcing a primary key) and xOrderDetail_ScanDT, the clustered index.
**Tip**

The index parameter to `DBCC SHOWCONTIG` is optional. If you do not specify an index number, you will get a report based on the clustered index. If the table is a heap table (with no clustered index), you will get a report on the base table.

dbcc showcontig(1458104235, 1)
dbcc showcontig(1458104235, 2)

**DBCC SHOWCONTIG scanning 'OrderDetail' table...**

Table: 'OrderDetail' (1458104235); index ID: 1, database ID: 8

TABLE level scan performed.
- Pages Scanned: 11567
- Extents Scanned: 1455
- Extent Switches: 10824
- Avg. Pages per Extent: 7.9
- Scan Density [Best Count:Actual Count]: 13.36% [1446:10825]
- Logical Scan Fragmentation: 47.66%
- Extent Scan Fragmentation: 34.78%
- Avg. Bytes Free per Page: 4310.8
- Avg. Page Density (full): 46.74%

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

**DBCC SHOWCONTIG scanning 'OrderDetail' table...**

Table: 'OrderDetail' (1458104235); index ID: 2, database ID: 8

LEAF level scan performed.
- Pages Scanned: 5917
- Extents Scanned: 752
- Extent Switches: 3915
- Avg. Pages per Extent: 7.9
- Scan Density [Best Count:Actual Count]: 18.90% [740:3916]
- Logical Scan Fragmentation: 29.37%
- Extent Scan Fragmentation: 67.29%
- Avg. Bytes Free per Page.....................: 2938.9
- Avg. Page Density (full).....................: 63.69%

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Pages Scanned shows how many index pages were scanned by SHOWCONTIG. Fewer pages is better.

Extents Scanned shows the number of extents that were examined. Taken with Avg Pages per Extent this number can show whether your extents are sparse.

Extent Switches shows how many times the dbcc command had to jump to a different extent to continue its index scan. Lower is better, with the best number equal to the number of scanned extents. The extent switches evident in these indexes show that the extent fragmentation is likely to be high.

Avg Pages per Extent describes how full the extents in the index are. 8.0 is the best possible value, although anything higher than 7 is fine. Tables that change clustered index values frequently or that perform many updates and deletes will often have low values here. Low pages per extent is causing wasted space in your indexes, as well as poor performance during index scans and read-ahead operations.

Scan Density shows how fragmented the index is. The first number is the ideal number of extent switches, followed by the actual number of switches. Anything under 60–70% should be addressed.

Logical Scan Fragmentation is an estimate of how out-of-order the leaf level of an index is. This is acquired by scanning the lowest index level top to bottom and counting the number of times the next page is not contiguous on the disk.

Extent Scan Fragmentation is a similar estimate for each extent in the index.

Avg Bytes Free per Page shows how much wasted space there is per index page. Large rows and low fillfactors will cause more wasted space. Completely full indexes are not necessarily desirable. A full index that is inserted or updated will need to split the index page. See the discussion on fillfactor titled “Setting the fillfactor” for more details.

Avg Page Density shows how full an index page is. 100% is ideal but rare. Consider fixing fragmentation when this figure gets below 60–70%.

Tip

You can get the tableid from sysobjects by using the following query:

```sql
select object_id('tablename')
```
Extent Fragmentation

It is also possible for updates and deletes to cause extent fragmentation. In this scenario, a full extent of eight, 8KB pages is mostly full, and has some contiguous portion of data deleted. Imagine that all the rows on six of the eight pages are deleted. With no more data on the page, SQL Server removes the page from the page chain. But because some part of the extent is still in use, the extent is not returned to the free pool. If the server needs to allocate a new page to this table, it might take this extent and link it into the page chain at a different area in the table.

Over time, this will cause noncontiguous pages on the disk to be linked together in a spaghetti maze of fragmentation. Delays will be most noticeable on disk operations that read entire extents, such as the read ahead manager or large index and table scans.

Fixing Fragmented Tables

To defragment an index, SQL Server 2000 provides two choices:

- New to 2000, you can run `dbcc indexdefrag` on each of the indexes.
- Drop and re-create the index. Re-creating the clustered index will rebuild the entire table and all indexes. (You can also use `DBCC DBREINDEX()` to accomplish this.)

This example shows how to use `indexdefrag` on the `OrderDetail` table. The first parameter is the database ID. Use 0 to use the current database.
dbc indexdefrag(0, OrderDetail, PK_OrderDetail)
dbcc indexdefrag(0, OrderDetail, xOrderDetail_ScanDT)

Pages Scanned | Pages Moved | Pages Removed
------------- | ----------- | -------------
5911         | 3878        | 1914         

(1 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Pages Scanned | Pages Moved | Pages Removed
------------- | ----------- | -------------
11560        | 11551       | 1

(1 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Pages Scanned shows the original number of pages in the index.

Pages Moved shows the number of scanned pages that were shuffled to reduce fragmentation. The goal of the defragmenter is to make the leaf level of the index contiguous on the disk so that index scan performance will be fast.

Pages Removed shows the number of empty index pages that were removed from the index. The defragmenter will move index keys between pages to improve scan density. In a sparse index, this will result in empty pages being removed from the page chain.

Index defragmentation is a nice addition to SQL 2000. The major benefit is that it lends itself to use during production hours. Although performance will be affected, defragmenting indexes will use lower-level blocking locks to reorganize the index, thus allowing processing to continue during its use. The observed blocking will be similar to what you would see during a deferred update on the table.

Index defragmentation will not always be able to reapply fillfactor appropriately. To get the best results, drop and rebuild the index to fix it.

Following is the output for showcontig after running indexdefrag on the OrderDetail table. Note that the clustered index retains a low page density, which would be fixed by re-creating the indexes:

DBCC SHOWCONTIG scanning 'OrderDetail' table...
Table: 'OrderDetail' (1458104235); index ID: 1, database ID: 8
TABLE level scan performed.
- Pages Scanned..........................: 11566
Nonclustered indexes use the clustered index keys as part of their key. To rebuild all indexes, you will save time by doing things in the following order: dropping all the nonclustered indexes, dropping the clustered index, building the clustered index, and then building the nonclustered indexes.

Following is the showcontig output for the OrderDetail table after rebuilding its indexes:

DBCC SHOWCONTIG scanning 'OrderDetail' table...
Table: 'OrderDetail' (1458104235); index ID: 1, database ID: 8
TABLE level scan performed.
- Pages Scanned.................................: 5335
- Extents Scanned.............................: 671
- Extent Switches.............................: 670
- Avg. Pages per Extent......................: 8.0
- Scan Density [Best Count:Actual Count]...: 99.40% [667:671]
- Logical Scan Fragmentation.................: 0.02%
- Extent Scan Fragmentation...................: 4.23%
- Avg. Bytes Free per Page....................: 17.2
- Avg. Page Density (full)....................: 99.79%
Tip

Rebuilding indexes can take a while on large tables. The indexdefrag function will usually run faster, because like a disk defragmenter, it doesn't move pages around when they are already in the correct place.

Setting the Fill Factor

The Fill Factor option allows you to define the percentage of free space on an index page when you create an index. The value can be from 1–100, with special meaning assigned to a 0 value. Setting the value to 80 would mean that each page would be 80% full at the time you create the index.

Fillfactor is a one-time operation. It is applied when the index is created, and when indexdefrag runs, but it is not maintained. Over a period of time, each index page will develop a different percentage of fullness.

Specify fillfactor during index creation like this:

```
create clustered index xOrderDetail_ScanDT
  on OrderDetail(ScanDT)
  with fillfactor=100
```

You can specify the default fill factor value at the server level by using `sp_configure`:

```
exec sp_configure 'fill factor', <% fillfactor>
```
If you don’t specify a value for fillfactor, the default value is zero, which has special meaning. A fillfactor value of zero fills the leaf level of the index (data pages for a clustered index) to 100%. On intermediate levels, a space for two rows is left open. This leaves some room open for inserts or updates that shift key values.

When you use PAD_INDEX, the value specified by fillfactor cannot be such that the number of rows on each index node falls below two. If you do specify such a value, SQL Server will internally override it.

**Reapplying the Fill Factor**

As data is modified in a table, the value of fillfactor is not maintained at the level specified during the create index statement. As a result, each page can reach a different level of fullness. Over a period of time, this can lead to heavy fragmentation in the database and will impact performance.

If you have a table with heavy inserts, and the indexes are relatively full, you will see heavy page splitting in the indexes. The server must split the index pages and link in a fresh page to accept more data. To combat this, reapply a fillfactor with a lower setting, leaving ample room throughout the system to accept new entries. This will, however, have an adverse effect on read performance.

The best way to reapply the fillfactor is to run indexdefrag. You can also reindex the table with DBCC DBREINDEX or by using drop index and create index.

**Updates and Performance**

SQL Server supports two types of update mechanisms: deferred update and update in place.

**Deferred Update**

Deferred mode occurs when the original rows are deleted and new rows can be inserted on different pages from those of the original rows. This expensive mode of update is treated as a DELETE followed by an INSERT statement.

A deferred update always occurs when the column being updated is a part of a clustered index or a nonclustered unique index. When a clustered index is updated, all nonclustered indexes must also be updated because the clustered key is part of the nonclustered key. On heavily indexed tables, avoid choosing a clustered key that is changed often.
Update in Place

Internally, SQL Server implements an update statement by performing a delete and then an insert statement. This means that the row is physically deleted from the page and potentially inserted on a different page. An update in place involves no physical movement of the rows being updated; that is, no insert follows a delete. Instead, the row is updated on the current page. Update in place is efficient because no movement occurs at the data page level or at the index page level.

Database File Groups and Performance

File groups allow you to decide where on the disk a particular object will be placed. You can do this by defining a file group within your database, extending the database onto a different drive or set of drives, and then placing a database object on the new file group.

File groups are most often used in high performance environments to isolate key tables or indexes on their own set of disks, which are in turn part of a high-performance RAID array. Assuming that you start with a database with just a PRIMARY file group (the default), this example shows how you would add an index file group on a new drive and move some nonclustered indexes over to it.

Note

Because the leaf level of a clustered index is the data page, if you create a clustered index on a file group, the entire table moves from the existing file group to the new file group. If you want to put indexes on a separate file group, reserve this space for nonclustered indexes, only.

```
-- add the file group
alter database Grocer
    add filegroup FG_INDEX

-- Create a new database file and add it to the FG_INDEX filegroup
alter database Grocer
add file(  
    NAME = Grocer_Index,  
    FILENAME = 'g:\Grocer_Index.ndf',  
    SIZE = 2048MB,  
    MAXSIZE = 8192MB,  
    FILEGROWTH = 10%
) to filegroup FG_INDEX
```
create nonclustered index xOrderDetail.ScanDT
  on OrderDetail.ScanDT
  on FG_INDEX

With your indexes on a separate filegroup, you get the following advantages:

- Index scans and index page reads come from a separate disk group, so they need
  not compete with other database processes for disk time.
- Inserts, updates, and deletes on the table are spread across two separate disk arrays.
  The clustered index, including all the table data, is on a separate array from the
  nonclustered indexes.
- You can target your budget dollars more precisely because faster disks will
  improve system performance more if they are given to the index filegroup rather
  than the database as a whole.

The next section on RAID will give specific recommendations on how to architect a
hardware solution based on a separate filegroup for indexes.

## RAID Technology

Redundant Array of Inexpensive Disks (RAID) is used to configure a disk subsystem to
provide better performance and fault tolerance for an application. The basic idea behind
using RAID is that you spread data across multiple disk drives so that I/Os are spread
across multiple drives. RAID has special significance for database-related applications,
where you want to spread random I/Os (data changes) and sequential I/Os (for the trans-
action log) across different disk subsystems to minimize head movement and maximize
I/O performance.

The four significant levels of RAID implementation that are of most interest in database
implementations are as follows:

- RAID 0 is data striping, no fault tolerance.
- RAID 1 is mirroring, where every disk in the array has a mirror (copy).
- RAID 5 is striping with parity, where each block of data is written to two separate,
  physical disks.
- RAID 10, or 0+1, is a combination of RAID 0 and 1. It is a stripe set that mirrors
  every drive in the stripe.

### RAID Level 0

RAID Level 0 provides the best I/O performance among all other RAID levels. A file has
sequential segments striped across each drive in the array, wrapping back around to the
first drive. If a media failure occurs, no fault tolerance is provided. RAID 0 is occasion-
ally used for tempdb to provide the best possible read and (especially) write perfor-
mance. RAID 0 is helpful for random read requirements, such as those that occur on
tempdb and in data segments.

RAID 0 is the cheapest of the RAIDs because 100% of the disks in the array are avail-
able for data, and none are used to provide fault tolerance.

Tip

Failure of a RAID 0 stripeset for a production database would be a hassle
because you would like to keep that data. Because tempdb's data is temporary,
if tempdb fails, recovery is relatively simple. To rebuild a suspect tempdb on a
RAID 0 array, use the following steps:

1. Restore the disk array to service by either replacing the failed drive or by
re-creating the array on surviving disks.
2. Remove the old tempdb files—they are useless without all the pieces.
3. Restart SQL Server with traceflag 3608, which bypasses recovery of all
databases except master.
4. Reset the tempdb status with sp_resetstatus tempdb.
5. Restart SQL Server. New tempdb files will be created during the recovery
phase.

Figure 38.6 depicts a RAID 0 disk array configuration.

RAID Level 1

RAID Level 1 is known as disk mirroring. Every write to the primary disk is written to the
mirror set. Either member of the set can satisfy a read request. RAID 1 devices provide
excellent fault tolerance because in a media failure, either on the primary disk or the mir-
rored disk, the system can still continue to run. Writes are much faster than RAID 5 arrays.
RAID 1 arrays are best for transaction logs and for index file groups. RAID 1 provides the best fault tolerance, and the best write performance, which is critical to log and index performance. Log writes are sequential write operations, and are best supported by a RAID 1 configuration.

RAID 1 arrays are the most expensive because only 50% of the disk array is available for active use. The rest is used to provide fault tolerance.

Figure 38.7 shows a RAID 1 configuration.

**FIGURE 38.7**
*RAID Level 1.*

---

**RAID Level 10**

RAID 10, or RAID 0+1, is a combination of mirroring and striping. If you find that your transaction log or index segment is pegging your RAID 1 array at 100% usage, you can implement a RAID 10 array to get better performance. This type of RAID carries with it all the fault tolerance (and cost!) of a RAID 1 array, with all the performance benefits of RAID 0 striping.

**RAID Level 5**

RAID 5 is most commonly known as striping with parity. In this configuration, data is striped across multiple disks in large blocks. At the same time, parity bits are written across all the disks for a given block. Information is always stored in such a way that any one disk can be lost without losing any information in the array. In the event of a disk failure, the system can still continue to run (at a reduced performance level) without downtime by using the parity information to reconstruct the data that was lost on the missing drive.

Some arrays provide “standby” disks. The RAID controller uses the standby disk to rebuild a failed drive using the parity information stored on all the other drives in the array. During the rebuild process, performance is markedly worse.

The fault tolerance of RAID 5 is usually sufficient, but if more than one drive in the array fails, you will lose the entire array. RAID 1 can sustain damage to up to half its drives without losing data.
RAID 5 provides excellent read performance, but expensive write performance. A write operation on a RAID 5 array requires two writes: one to the data drive and one to the parity drive. After the writes are complete, the controller will read the data to ensure that the information matches (no hardware failure has occurred). A single write operation will cause four I/Os on a RAID 5 array. For this reason, it is a bad idea to put log files or tempdb on a RAID 5 array. Index file groups, which suffer worse than data file groups from bad write performance, are also poor candidates for a RAID 5 array.

Note that if write performance is not an issue in your environment—for example, in a DSS/Data Warehousing environment—you should, by all means, use RAID 5 for your index segments.

In any environment, never put tempdb on a RAID 5 array. tempdb uses heavy writes, and belongs on a RAID 1 or RAID 0 array.

RAID 5 is relatively economical. No matter how many drives are in the array, only one is used to support fault tolerance. This method is more economical with more drives in the array. You must have at least three drives in a RAID 5 array. Three drives would use 33% of available disk space on fault tolerance, four would use 25%, and so on.

Figure 38.8 shows a RAID 5 configuration.

**Summary**

A good database design is the best place to start to ensure a smoothly running system. You can take steps to ensure a solid design. If you inherit a database with an inadequate design, you can take steps to ensure good performance. The primary goals of a database designer should be to index the database effectively and keep table row sizes as narrow as possible. The selection of an appropriate clustered index goes a long way toward ensuring excellent performance.
Using the information you learned in this chapter about designing for performance, the next chapter, “Configuring, Tuning, and Optimizing SQL Server Options,” will show you how to take your new database design concepts into production. You will learn how to configure a server for the best performance under different circumstances, and identify and optimize the server when performance is inadequate.