Ken England
Gavin Powell

dp

Microsoft® SQL Server™ 2005 Performance Optimization and Tuning Handbook

DATA MANAGEMENT

Microsoft® SQL Server™ 2005 Performance Optimization and Tuning Handbook

Microsoft® SQL Server™ 2005 Performance Optimization and Tuning Handbook

Ken England
Gavin Powell

ELSEVIER

Amsterdam • Boston • Heidelberg • London • New York • Oxford
Paris • San Diego • San Francisco • Singapore • Sydney • Tokyo
Digital Press is an imprint of Elsevier

Digital Press

Digital Press is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Library of Congress Cataloging-in-Publication Data
Application Submitted.

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-1-55558-319-4

For information on all Elsevier Digital Press publications visit our Web site at www.books.elsevier.com

Printed in the United States of America
07 08 09 10 11 12 10 9 8 7 6 5 4 3 2 1

Contents at a Glance

Contents

Introduction

What is the goal of tuning an SQL Server database? The goal is to improve performance until acceptable levels are reached. Acceptable levels can be defined in a number of ways. For a large online transaction processing (OLTP) application the performance goal might be to provide sub-second response time for critical transactions and to provide a response time of less than two seconds for 95 percent of the other main transactions. For some systems, typically batch systems, acceptable performance might be measured in throughput. For example, a settlement system may define acceptable performance in terms of the number of trades settled per hour. For an overnight batch suite acceptable performance might be that it must finish before the business day starts.

Whatever the system, designing for performance should start early in the design process and continue after the application has gone live. Performance tuning is not a one-off process but an iterative process during which response time is measured, tuning performed, and response time measured again.

There is no right way to design a database; there are a number of possible approaches and all these may be perfectly valid. It is sometimes said that performance tuning is an art, not a science. This may be true, but it is important to undertake performance tuning experiments with the same kind of rigorous, controlled conditions under which scientific experiments are performed. Measurements should be taken before and after any modification, and these should be made one at a time so it can be established which modification, if any, resulted in an improvement or degradation.

What areas should the database designer concentrate on? The simple answer to this question is that the database designer should concentrate on those areas that will return the most benefit. In my experience, for most database designs I have worked with, large gains are typically made in the area of query and index design. As we shall see later in this book, inappropriate indexes and badly written queries, as well as some other contributing factors, can negatively influence the query optimizer such that it chooses an inefficient strategy.

To give you some idea of the gains to be made in this area, I once was asked to look at a query that joined a number of large tables together. The query was abandoned after it had not completed within 12 hours. The addition of an index in conjunction with a modification to the query meant the query now completed in less than eight minutes! This magnitude of gain cannot be achieved just by purchasing more hardware or by twiddling with some arcane SQL Server configuration option. A database designer or administrator's time is always limited, so make the best use of it! The other main area where gains can be dramatic is lock contention. Removing lock bottlenecks in a system with a large number of users can have a huge impact on response times.

Now, some words of caution when chasing performance problems. If users phone up to tell you that they are getting poor response times, do not immediately jump to conclusions about what is causing the problem. Circle at a high altitude first. Having made sure that you are about to monitor the correct server, use the System Monitor to look at the CPU, disk subsystem, and memory use. Are there any obvious bottlenecks? If there are, then look for the culprit. Everyone blames the database, but it could just as easily be someone running his or her favorite game! If there are no obvious bottlenecks, and the CPU, disk, and memory counters in the System Monitor are lower than usual, then that might tell you something. Perhaps the network is sluggish or there is lock contention. Also be aware of the fact that some bottlenecks hide others. A memory bottleneck often manifests itself as a disk bottleneck.

There is no substitute for knowing your own server and knowing the normal range of System Monitor counters. Establish trends. Measure a set of counters regularly, and then, when someone comments that the system is slow, you can wave a graph in front of him or her showing that it isn't!

Also there are special thanks to be made to Craig Mullins for his work on technical editing of this book.

So, when do we start to worry about performance? As soon as possible, of course! We want to take the logical design and start to look at how we should transform it into an efficient physical design.

Gavin Powell can be contacted at the following email address:

ezpowell@ezpowell.com

Performance and SQL Server 2005

1.1 Partitioning tables and indexes

Partitioning lets you split large chunks of data in much more manageable smaller physical chunks of disk space. The intention is to reduce I/O activity. For example, let's say you have a table with 10 million rows and you only want to read 1 million rows to compile an analytical report. If the table is divided into 10 partitions, and your 1 million rows are contained in a single partition, then you get to read 1 million rows as opposed to 10 million rows. On that scale you can get quite a serious difference in I/O activity for a single report.

SQL Server 2005 allows for table partitioning and index partitioning. What this means is that you can create a table as a partitioned table, defining specifically where each physical chunk of the table or index resides.

SQL Server 2000 partitioning was essentially manual partitioning, using multiple tables, distributed across multiple SQL Server computers. Then a view (partition view) was created to overlay those tables across the servers. In other words, a query required access to a view, which contained a query, not data. SQL Server 2005 table partitions contain real physical rows.

Physically partitioning tables and indexes has a number of benefits:

- Data can be read from a single partition at once, cutting down enormously on performance hogging I/O.
- Data can be accessed from multiple partitions in parallel, which gets things done at double the speed, depending on how many processors a server platform has.
- Different partitions can be managed separately, without having to interfere with the entire table.

1.2 Building indexes online

Building an index online allows the table indexed against to be accessed during the index creation process. Creating or regenerating an index for a very large table can consume a considerable period of time (hours, days). Without online index building, creating an index puts a table offline. If that is crucial to the running of a computer system, then you have down time. The result was usually that indexes are not created, or never regenerated.

Even the most versatile BTree indexes can sometimes require rebuilding to increase their performance. Constant data manipulation activity on a table (record insert, update and deletions) can cause a BTree index to deteriorate over time. Online index building is crucial to the constant uptime required by modern databases for popular websites.

1.3 Transact SQL improvements

Transact SQL provides programmable access to SQL Server. Programmable access means that Transact SQL allows you to construct database stored code blocks, such as stored procedures, triggers, and functions. These code blocks have direct access to other database objects—most significantly tables where query and data manipulation commands can be executed directly in the stored code blocks; and code blocks are executed on the database server. New capabilities added to Transact SQL in SQL Server 2005 are as follows:

- Error handling
- Recursive queries
- Better query writing capabilities

There is also something new to SQL Server 2005 called Multiple Active Result Sets (MARS). MARS allows for more than a single set of rows for a single connection. In other words, a second query can be submitted to a SQL Server while the result set of a first query is still being returned from database server to client application.

The overall result of Transact SQL enhancements to SQL Server 2005 is increased performance of code, better written code, and more versatility. Better written code can ultimately make for better performing applications in general.

1.4 Adding the .NET Framework

You can use programming languages other than just Transact SQL and embed code into SQL Server as .NET Framework executables. These programing languages can leverage existing personnel skills. Perhaps more importantly, some tasks can be written in programming languages more appropriate to a task at hand. For example a language like C# can be used, letting a programmer take advantage of the enormous speed advantages of writing executable code using the C programming language.

Overall, you get support for languages not inherently part of SQL Server (Transact SQL). You get faster and easier development. You get to use Web Services and XML (with Native XML capabilities using XML data types). The result is faster development, better development, and hopefully better over database performance in the long run.

The result you get is something called managed code. Managed code is code executed by the .NET Framework. As already stated, managed code can be written using all sorts of programming languages. Different programming languages have different benefits. For example, C is fast and efficient, where Visual Basic is easier to write code with but executes slower. Additionally, the .NET Framework has tremendous built-in functionality. .NET is much, much more versatile and powerful than Transact SQL.

There is much to be said for placing executable into a database, on a database server such as SQL Server. There is also much to be said against this practice. Essentially, the more metadata and logic you add to a database, the more business logic you add to a database. In my experience, adding too much business logic to a database can cause performance problems in the long run. After all, application development languages cater to number crunching and other tasks. Why put intensive, non-data access processing into a database? The database system has enough to do just in keeping your data up to date and available.

Managed code also compiles to native code, or native form in SQL Server, immediately prior to execution. So, it should execute a little faster because it executes in a form which is amenable to best performance in SQL Server.

SQL Server 2005 includes a new management object model called SQL Management Objects (SMO). The SMO has a basis in the .NET Framework. The new graphical, SQL Server Management Studio, is written using the SMO.

1.5 Trace and replay objects

Tracing is the process of producing large amounts of log entry information during the process of normal database operations. However, it might be prudent to not choose tracing as a first option to solving a performance issue. Tracing can hurt performance simply because it generates lots of data. The point of producing trace files is to aid in finding errors or performance bottlenecks, which cannot be deciphered by more readily available means. So, tracing quite literally produces trace information. Replay allows replay of actions that generated those trace events. So, you could replay a sequence of events against a SQL Server, without actually changing any data, and reproduce the unpleasant performance problem. And then you could try to reanalyze the problem, try to decipher it, and try to resolve or improve it.

1.6 Monitoring resource consumption with SQL OS

SQL OS is a new tool for SQL Server 2005, which lives between an SQL Server database and the underlying Windows operating system (OS). The operating system manages, runs, and accesses computer hardware on your database server, such as CPU, memory, disk I/O, and even tasks and scheduling. SQL OS allows a direct picture into the hardware side of SQL Server and how the database is perhaps abusing that hardware and operating system. The idea is to view the hardware and the operating system from within an SQL Server 2005 database.

1.7 Establishing baseline metrics

A baseline is a setting established by a database administrator, either written on paper, but preferably stored in a database (generated by the database). This baseline establishes an acceptable standard of performance. If a baseline is exceeded then the database is deemed to have a performance problem. A metric is essentially a measure of something. The result is many metrics, with established acceptable baseline values. If one or more metric baselines are exceeded then there is deemed to be one or more performance problems. Additionally, each metric can be exceeded for a previously established reason, based on what the metric is. So, if a table, with its indexes, has an established baseline value of 10 bytes per minute of I/O activity, and suddenly that value jumps up to 10 giga bytes per minute—there is probably a performance problem.

An established baseline metric is a measure of normal or acceptable activity.

Metric baselines have more significance (there are more metrics) in SQL Server 2005 than in SQL Server 2000. The overall effect is that an SQL Server 2005 database is now more easily monitored, and the prospect of some automated tuning activities becomes more practical in the long term. SQL Server 2005 has added over 70 additional baseline measures applicable to performance of an SQL Server database. These new baseline metrics cover areas such as memory usage, locking activities, scheduling, network usage, transaction management, and disk I/O activity.

The obvious answer to a situation such as this is that a key index is dropped, corrupt, or deteriorated. Or a query could be doing something unexpected such as reading all rows in a very large table.

Using metrics and their established baseline or expected values, one can perform a certain amount of automated monitoring and detection of performance problems.

Baseline metrics are essentially statistical values collected for a set of metrics.

A metric is a measure of some activity in a database.

The most effective method of gathering those expected metric values is to collect multiple values—and then aggregate and average them. And thus the term statistic applies because a statistic is an aggregate or average value, resulting from a sample of multiple values. So, when some activity veers away from previously established statistics, you know that there could be some kind of performance problem—the larger the variation, the larger the potential problem.

Baseline metrics should be gathered in the following activity sectors:

- **High load**: Peak times (highest database activity)
- **Low load**: Off peak times (lowest database activity)
- **Batch activity**: Batch processing time such as during backup processing and heavy reporting or extraction cycles
- **Downtime**: How long it takes to backup, restore, and recover is something the executive management will always have to detail to clients. This equates to uptime and potential downtime

Some very generalized categories areas of metric baseline measurement are as follows:

- **Applications database access**: The most common performance problems are caused by poorly built queries and locking or hot blocks (conflict caused by too much concurrency on the same data).

  In computer jargon, concurrency means lots of users accessing and changing the same data all at the same time. If there are too many concurrent users, ultimately any relational database has its limitations on what it can manage efficiently.

- **Internalized database activity**: Statistics must not only be present but also kept up to date. When a query reads a table, it uses what's called an optimizer process to make a wild guess at what it should do. If a table has 1 million rows, plus an index, and a query seeks 1 record, the optimizer will tell the query to read the index. The optimizer uses statistics to compare 1 record required, within 1 million rows available. Without the optimizer 1 million rows will be read to find 1 record. Without the statistics the optimizer cannot even hazard a guess and will probably read everything. If statistics are out of date where the optimizer thinks the table has 2 rows, but there are really 1 million, then the optimizer will likely guess very badly.

- **Internalized database structure**: Too much business logic, such as stored procedures or a highly over normalized table structure, can ultimately cause overloading of a database, slowing performance because a database is just a little too *top heavy*.

- **Database configuration**: An OLTP database accesses a few rows at a time. It often uses indexes, depending on table size, and will pass very small amounts of data across network and telephone cables. So, an OLTP database can be specifically configured to use lots of memory—things like caching on client computers and middle tier servers (web and application servers), plus very little I/O. A data warehouse on the other hand produces a small number of very large transactions, with low memory usage, enormous amounts of I/O, and lots of throughput processing. So, a data warehouse doesn't care too much about memory but wants the fastest access to disk possible, plus lots of localized (LAN) network bandwidth. An OLTP database uses all hardware resources and a data warehouse uses mainly I/O.

- **Hardware resource usage**: This is really very similar to the above point under database configuration, expect that hardware can be improved upon. In some circumstances beefing up hardware will solve performance issues. For example, an OLTP database server needs plenty of memory, whereas a data warehouse does well with fast disks, and perhaps multiple CPUs with partitioning for rapid parallel processing. Beefing up hardware doesn't always help. Sometimes increasing CPU speed and number, or increasing onboard memory, can only hide performance problems until a database grows in physical size, or there are more users—the problem still exists. For example, poor query coding and indexing in an OLTP database will always cause performance problems, no matter how much money is spent on hardware. Sometimes hardware solutions are easier and cheaper, but often only a stopgap solution.

- **Network design and configuration**: Network bandwidth and bottlenecks can cause problems sometimes, but this is something rarely seen in commercial environments because the network engineers are usually prepared for potential bandwidth requirements.

The above categories are most often the culprits of the biggest performance issues. There are other possibilities, but they are rare and don't really warrant mentioning at this point. Additionally, the most frequent and exacerbating causes of performance problems are usually the most obvious ones, and more often than not something to do with the people maintaining and using the software, inadequate software, or inadequate hardware. Hardware is usually the easiest problem to fix. Fixing software is more expensive depending on location of errors in database or application software. Persuading users to use your applications and database the way you want is either a matter of expensive training, or developers having built software without enough of a human use (user friendly) perspective in mind.

1.8 Start using the GUI tools

Traditionally, many database administrators will still utilize command line tools because they perceive them as being more grassroots and, thus easier to use. Sometimes these administrators are correct. I am as guilty of this as is anyone else. However, as in any profession, new gadgets are often frowned upon due to simple resistance to change and a desire to deal with tools and methods which are familiar. The new GUI tools appearing in many relational databases these days are just too good to miss.

1.8.1 SQL Server Management Studio

The SQL Server Management Studio is a new tool used to manage all the facets of an SQL Server, including multiple databases, tables, indexes, fields, and data types, anything you can think of. Figure 1.1 shows a sample view of the SQL Server Management Studio tool in SQL Server 2005.

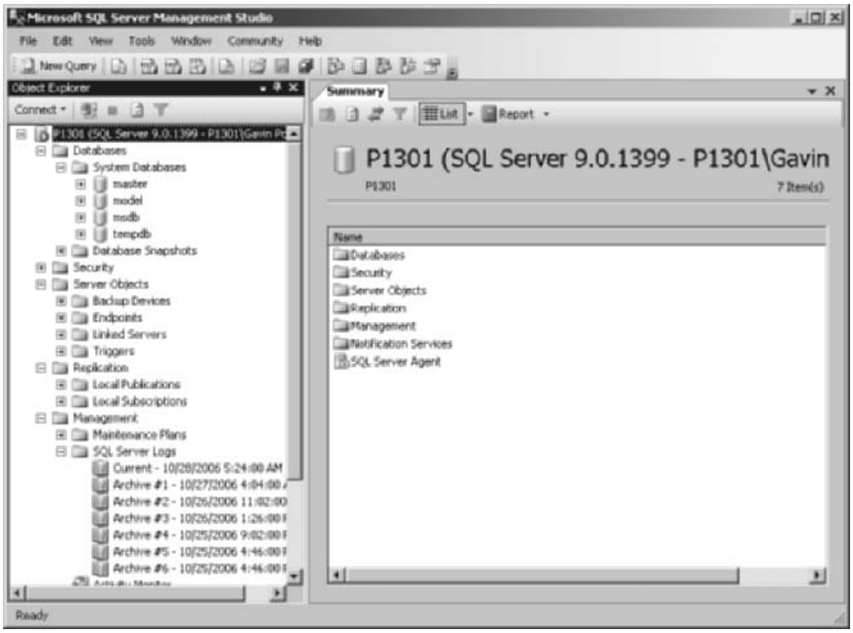

Microsoft SQL Server Management Studio
File Edit View Tools Window Community Help
New Query
Object Explorer
Connect
P1301 (SQL Server 9.0.1399 - P1301\Gavin Pc
Databases
System Databases
master
model
msdb
tempdb
Database Snapshots
Security
Server Objects
Backup Devices
Endpoints
Linked Servers
Triggers
Replication
Local Publications
Local Subscriptions
Management
Maintenance Plans
SQL Server Logs
Current - 10/28/2006 5:24:00 AM
Archive #1 - 10/27/2006 4:04:00
Archive #2 - 10/26/2006 11:02:00
Archive #3 - 10/26/2006 1:26:00
Archive #4 - 10/25/2006 9:02:00
Archive #5 - 10/25/2006 4:46:00
Archive #6 - 10/25/2006 4:46:00
Summary
List
Report
P1301 (SQL Server 9.0.1399 - P1301\Gavin
P1301
7 Item(s)
Name
Databases
Security
Server Objects
Replication
Management
Notification Services
SQL Server Agent
Ready

Figure 1.1 *SQL Server Management Studio*

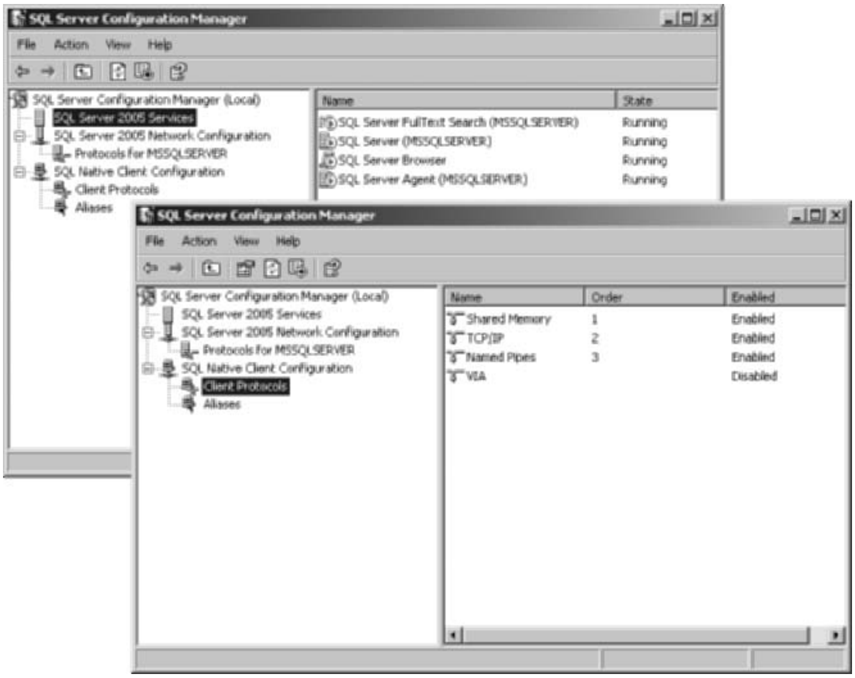
SQL Server Management Studio is a fully integrated, multi-task oriented screen (console) that can be used to manage all aspects of an SQL Server installation, including direct access to metadata and business logic, integration, analysis, reports, notification, scheduling, and XML, among other facets of SQL Server architecture. Additionally, queries and scripting can be constructed, tested, and executed. Scripting also includes versioning control (multiple historical versions of the same piece of code allow for backtracking). It can also be used for very easy general database maintenance.

SQL Server Management Studio is in reality wholly constructed using something called SQL Management Objects (SMO). SMO is essentially a very large group of predefined objects, built-in and reusable, which can be used to access all functionality of a SQL Server database. SMO is written using the object-oriented and highly versatile .NET Framework. Database administrators and programmers can use SMO objects in order to create their own customized procedures, for instance, to automate something like daily backup processing.

SMO is an SQL Server 2005 updated and more reliable version of Distributed Management Objects (DMO), as seen in versions of SQL Server prior to SQL Server 2005.

1.8.2 SQL Server Configuration Manager

The SQL Server 2005 Configuration Manager tool allows access to the operating system level. This includes services such as configuration for client application access to an SQL Server database, as well as access to database server services running on a Windows server. This is all shown in Figure 1.2.


SQL Server Configuration Manager
File
Action
View
Help
SQL Server Configuration Manager
File
Action
View
Help

```
SQL Server Configuration Manager (Local)
    SQL Server 2005 Services
    SQL Server 2005 Network Configuration
        Protocols for MSSQLSERVER
    SQL Native Client Configuration
        Client Protocols
        Aliases

Name                                      State
SQL Server FullText Search (MSSQLSERVER)  Running
SQL Server (MSSQLSERVER)                  Running
SQL Server Browser                        Running
SQL Server Agent (MSSQLSERVER)            Running

Name            Order    Enabled
Shared Memory   1        Enabled
TCP/IP          2        Enabled
Named Pipes     3        Enabled
VIA                      Disabled
```

Figure 1.2 *SQL Server Configuration Manager*

1.8.3 Database Engine Tuning Advisor

The SQL Server 2005 Database Engine Tuning Advisor tool is just that, a tuning advisor used to assess options for tuning the performance of an SQL

Server database. This tool includes both a Graphical User Interface in Windows and a command line tool called **dta.exe**.

This book will focus on the GUI tools as they are becoming more prominent in recent versions of all relational databases,

The SQL Server 2005 Database Engine Tuning Advisor includes other tools from SQL Server 2000, such as the Index Tuning Wizard. However, SQL Server 2005 is very much enhanced to cater to more scenarios and more sensible recommendations. In the past, recommendations have been basic at best, and even wildly incorrect. Also, now included are more object types including differentiating between clustered and non-clustered indexing, plus indexing for view, and of course partitioning and parallel processing.

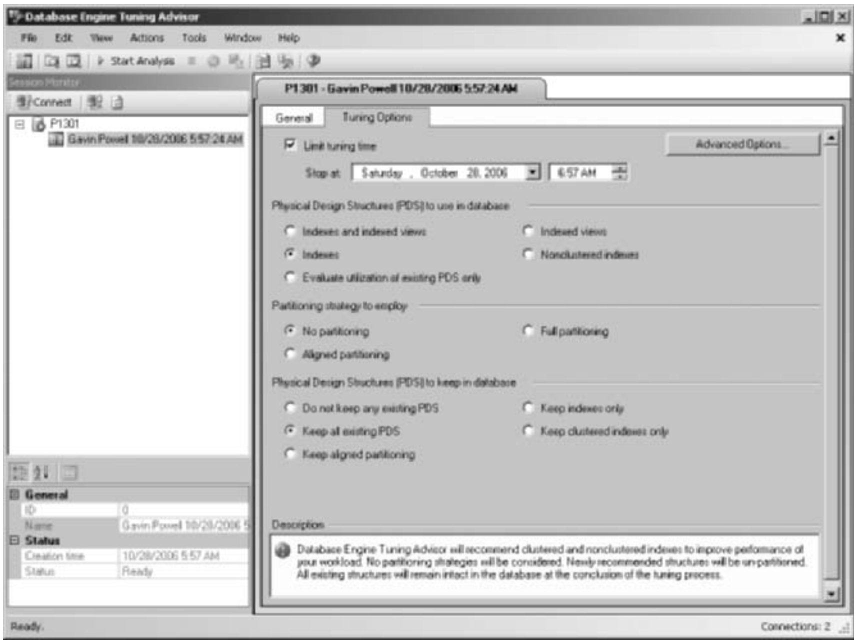
The Database Engine Tuning Advisor is backwardly compatible with previous versions of SQL Server.

New features provided by the SQL Server 2005 Database Engine Tuning Advisor tool are as follows:

- **Multiple databases**: Multiple databases can be accessed at the same time.
- **More objects types**: As already stated, more object types can be tuned. This includes XML, XML data types, and partitioning recommendations. There is also more versatility in choosing what to tune and what to recommend for tuning. Figure 1.3 shows available options for differing object types allowed to be subjected to analysis.

  And there are also some advanced tuning options as shown in Figure 1.4.

- **Time period workload analysis**: Workloads can be analyzed over set time periods, thus isolating peak times, off-peak times, and so on. Figure 1.5 shows analysis, allowance of time period settings, as well as application and evaluation of recommendations made by the tool.
- **Tuning log entries**: A log file containing a record of events which the Database Engine Tuning Advisor cannot tune automatically. This log can be use by a database administrator to attempt manual tuning if appropriate.
- **Negligible size test database copy**: The Database Engine Tuning Advisor can create a duplicate test copy of a production environment,


Database Engine Tuning Advisor
File Edit View Actions Tools Window Help
Start Analysis
Session Monitor
Connect
P1301
Gavin Powell 10/28/2006 5:57:24 AM
General
Status
Ready.
Connections: 2

P1301 - Gavin Powell 10/28/2006 5:57:24 AM

General | Tuning Options

[✓ Limit tuning time]

Stop at: Saturday , October 28, 2006 6:57 AM

Advanced Options...

Physical Design Structures (PDS) to use in database

- Indexes and indexed views
- [✓ Indexes]
- Evaluate utilization of existing PDS only
- Indexed views
- Nonclustered indexes

Partitioning strategy to employ

- [✓ No partitioning]
- Aligned partitioning
- Full partitioning

Physical Design Structures (PDS) to keep in database

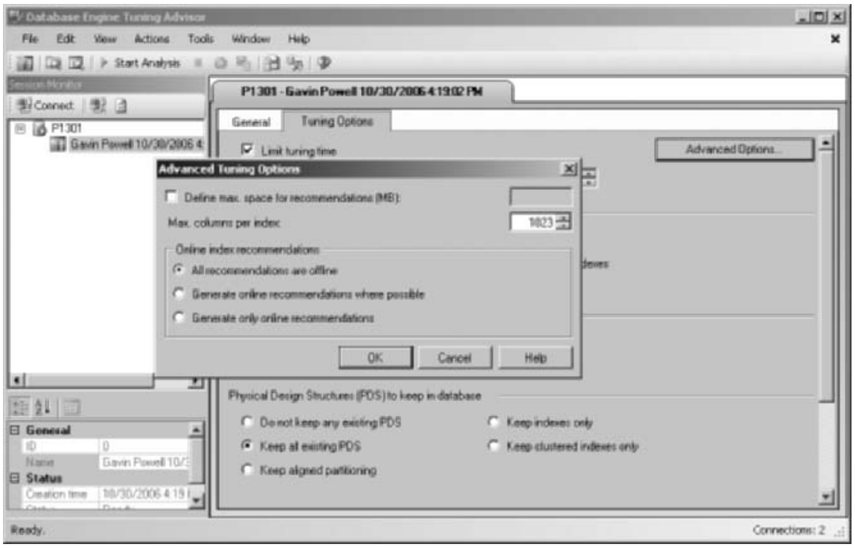
- Do not keep any existing PDS
- [✓ Keep all existing PDS]
- Keep aligned partitioning
- Keep indexes only
- Keep clustered indexes only

Description

Database Engine Tuning Advisor will recommend clustered and nonclustered indexes to improve performance of your workload. No partitioning strategies will be considered. Newly recommended structures will be un-partitioned. All existing structures will remain intact in the database at the conclusion of the tuning process.

ID	0
Name	Gavin Powell 10/28/2006 5
Creation time	10/28/2006 5:57 AM
Status	Ready

Figure 1.3
Object types to tune in the Database Engine Tuning Advisor


Database Engine Tuning Advisor
File Edit View Actions Tools Window Help
Start Analysis
Session Monitor
Connect
P1301
Gavin Powell 10/30/2006 4:
General
Status
OK
Cancel
Help
Ready.
Connections: 2

P1301 - Gavin Powell 10/30/2006 4:19:02 PM

General | Tuning Options

[✓ Limit tuning time]

Advanced Options...

Advanced Tuning Options

- Define max. space for recommendations (MB):
- Max. columns per index: 1023

Online index recommendations

- [✓ All recommendations are offline]
- Generate online recommendations where possible
- Generate only online recommendations

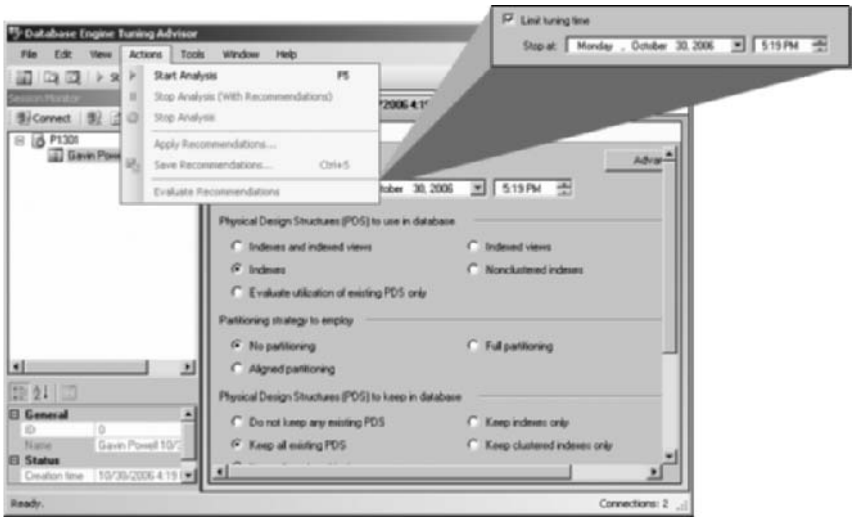
Physical Design Structures (PDS) to keep in database

- Do not keep any existing PDS
- [✓ Keep all existing PDS]
- Keep aligned partitioning
- Keep indexes only
- Keep clustered indexes only

ID	0
Name	Gavin Powell 10/3
Creation time	10/30/2006 4:19

Figure 1.4
Advanced object type tuning options in the Database Engine Tuning Advisor

in order to offload performance tuning testing processing. Most importantly, the test database created does not copy data. The only thing copied is the state of a database without the actual data. This is actually very easy for a relational database like SQL Server. All that is



[✓ Limit tuning time]
Stop at: Monday , October 30, 2006 5:19 PM

Physical Design Structures (PDS) to use in database
- Indexes and indexed views
- [✓ Indexes]
- Evaluate utilization of existing PDS only
- Indexed views
- Nonclustered indexes

Partitioning strategy to employ
- [✓ No partitioning]
- Aligned partitioning
- Full partitioning

Physical Design Structures (PDS) to keep in database
- Do not keep any existing PDS
- [✓ Keep all existing PDS]
- Keep indexes only
- Keep clustered indexes only

ID: 0
Name: Gavin Powell 10/
Creation time: 10/30/2006 4:19

Figure 1.5
Setting analytical time periods in the Database Engine Tuning Advisor

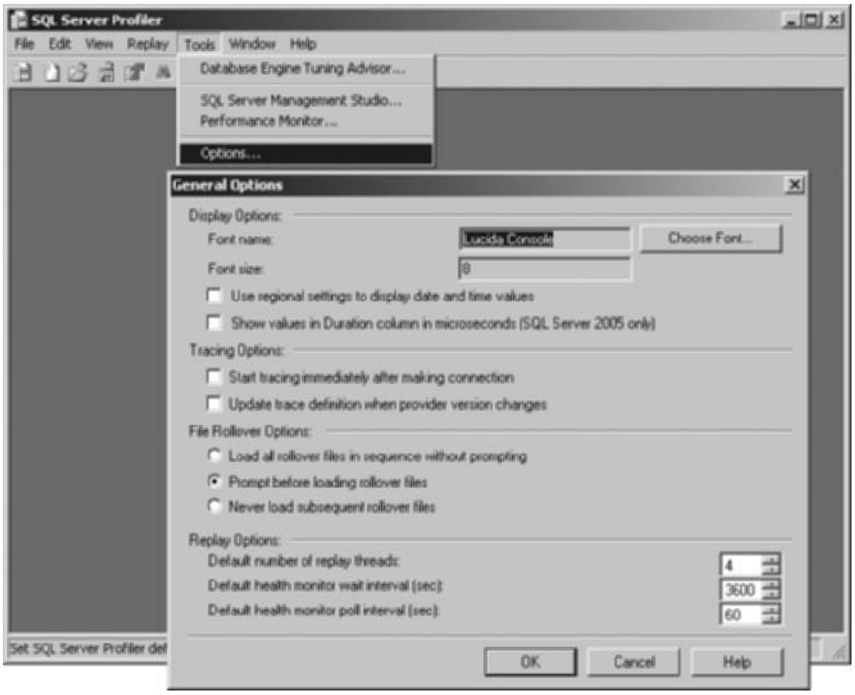
copied are objects, such as tables and indexes, plus statistics of those objects. Typical table statistics include record counts and physical size. This allows a process such as the optimizer to accurately estimate how to execute a query.

- **What-if scenarios**: A database administrator can create a configuration and scenario and subject it to the Database Engine Tuning Advisor. The advisory tool can give a response as to the possible effects of specified configuration changes. In other words, you can experiment with changes, and get an estimation of their impact, without making those changes in a production environment.

1.8.4 SQL Server Profiler

The SQL Server Profiler tool was available in SQL Server 2000 but has some improvements in SQL Server 2005. Improvements apply to the recording of things or events, which have happened in the database, and the ability to replay those recordings. The replay feature allows repetition of problematic scenarios which are difficult to resolve.

Essentially, the SQL Server Profiler is a direct window into trace files. Trace for any relational database contain a record of some, most, or even all activities in a database. Trace files can also include general table and indexing statistics as well. Performance issues related to trace files themselves is that tracing can be relatively intensive, depending on how tracing is configured. Sometimes too much tracing can affect overall database performance,


SQL Server Profiler
File Edit View Replay Tools Window Help
Database Engine Tuning Advisor...
SQL Server Management Studio...
Performance Monitor...
Options...
General Options
Choose Font...
OK Cancel Help
Set SQL Server Profiler def

Display Options:
Font name: Lucida Console
Font size: 8
☐ Use regional settings to display date and time values
☐ Show values in Duration column in microseconds (SQL Server 2005 only)
Tracing Options:
☐ Start tracing immediately after making connection
☐ Update trace definition when provider version changes
File Rollover Options:
○ Load all rollover files in sequence without prompting
● Prompt before loading rollover files
○ Never load subsequent rollover files
Replay Options:
Default number of replay threads: 4
Default health monitor wait interval (sec): 3600
Default health monitor poll interval (sec): 60

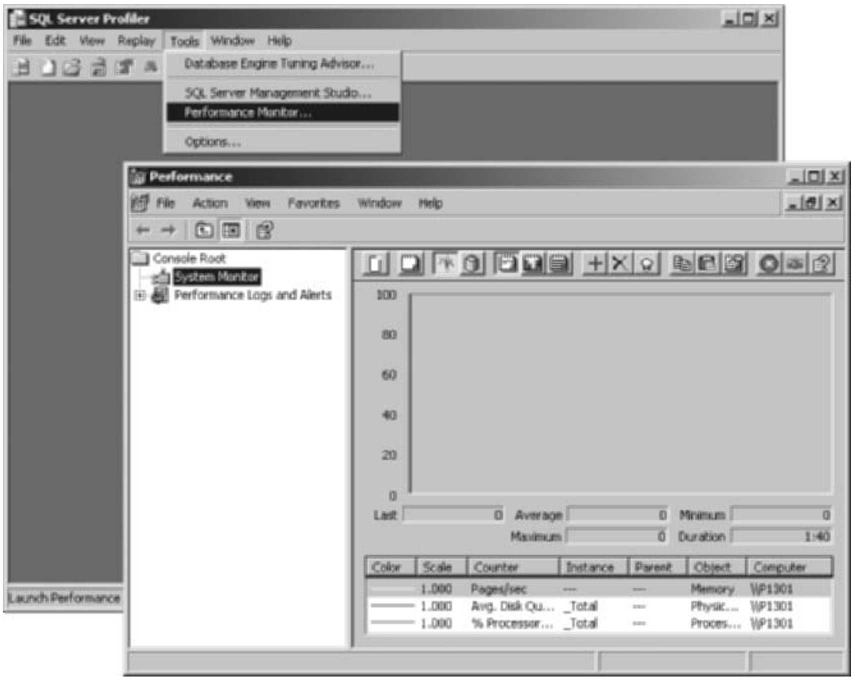
Figure 1.6
SQL Server Profiler options

and sometimes even quite drastically. Tracing is usually a last resort but also a very powerful option when it comes to tracking down the reason for performance problems and bottlenecks.

There are a number of things new to SQL Server 2005 for SQL Server Profiler:

- **Trace file replays**: Rollover trace files can be replayed. Figure 1.6 shows various options that can be set for tracing, rollover, and subsequent tracing entry replay.
- **XML**: The profiler tool has more flexibility by allowing for various definitions using XML.
- **Query plans in XML**: Query plans can be stored as XML allowing for viewing without database access.
- **Trace entries as XML**: Trace file entries can be stored as XML allowing for viewing without database access.

- **Analysis Services**: SQL Server Profiler now allows for tracing of Analysis Services (SQL Server data warehousing) and Integration Services.
- **Various other things**: Aggregate views of trace results and Performance Monitor counters matched with SQL Server database events. The Windows Performance Monitor tool is shown in Figure 1.7.


SQL Server Profiler
File Edit View Replay Tools Window Help
Database Engine Tuning Advisor...
SQL Server Management Studio...
Performance Monitor...
Options...
Launch Performance
Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
80
60
40
20
0

Last	0	Average	0	Minimum	0
		Maximum	0	Duration	1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
	1.000	Pages/sec	---	---	Memory	\\P1301
	1.000	Avg. Disk Qu...	_Total	---	Physic...	\\P1301
	1.000	% Processor...	_Total	---	Proces...	\\P1301

Figure 1.7
The Windows Performance Monitor tool from SQL Server Profiler

1.8.5 Business Intelligence Development Studio

This tool is used to build something called Business Intelligence (BI) objects. The BI Development Studio is a new SQL Server 2005 tool used to manage projects for development. This tool allows for integration of various aspects of SQL Server databases, including analysis, integration, and reporting services. This tool doesn't really do much for database performance in general, but moreover can help to speed up development, and make development a cleaner and better coded process. In the long term, better built applications perform better.

1.9 Availability and scalability

Availability means that an SQL Server database will have less down time and is less likely to irritate your source of income—your customers. Scalability means you can now service more customers with SQL Server 2005. Availability and scalability are improved in SQL Server 2005 by the addition and enhancement of the following:

- **Data mirroring**: Addition hot standby databases. This is called database mirroring in SQL Server.
- **Clustering**: Clustering is introduced which is not the same thing as a hot standby. A hot standby takes over from a primary database, in the event that the primary database fails. Standby is purely failover ability. A clustered environment provides more capacity and up-time by allowing connections and requests to be serviced by more than one computer in a cluster of computers. Many computers work in a cluster, in concert with each other. A cluster of multiple SQL Server databases effectively becomes a single database spread across multiple locally located computers—just a much larger and more powerful database. Essentially, clustering provides higher capacity, speed through mirrored and parallel databases access, in addition to just failover potential. When failover occurs in a clustered environment the failed node is simply no longer servicing the needs of the entire cluster, whereas a hot standby is a switch from one server to another.
- **Replication**: Replication is enhanced in SQL Server 2005. Workload can be spread across multiple, distributed, replicated databases. Also the addition of a graphical Replication Monitor tool eases management of replication and distributed databases.
- **Snapshot flashbacks**: A snapshot is a picture of a database, frozen at a specific point in time. This allows users to go back in time, and look at data at that point in time in the past. The performance benefit is that the availability of old data sets, in static databases, allows queries to be executed against multiple sets of the same data.
- **Backup and restore**: Improved restoration using snapshots to enhance restore after crash recovery, and giving partial, general online access, during the recovery process.
- **Bulk imports**: This is improved in SQL Server 2005.

1.10 Other useful stuff

Other useful stuff introduced in SQL Server 2005 offers the potential of improving the speed of development and software quality. Improvements include the following:

- **Native XML data types**: Native XML allows the storage of XML documents in their entirety inside an SQL Server database. The term Native implies that that stored XML document is not only stored as the textual data of the XML document, but it also includes the browser interpretive XML structure and metadata meaning. In other words, XML data types are directly accessible from the database as fully executable XML documents. The result is the inclusion of all the power, versatility, and performance of XML in general—ALL OF IT! Essentially, XML data types allow direct access to XQuery, SOAP, XML data manipulation languages, XSD—anything and everything XML. Also included are specifics of XML exclusive to SQL Server [1].
- **XML is the eXtensible Markup Language**. There is a whole host of stuff added to SQL Server 2005 with the introduction of XML data types. You can even create specific XML indexes, indexing stored XML data type and XML documents.
- **Service broker notification**: This helps performance enormously because multiple applications are often tied together in eCommerce architectures. The Server Broker part essentially organizes messages between different things. The notification part knows what to send, and where. For example, a user purchases a book online at the Amazon website. What happens? Transactions are placed into multiple different types of databases:
  - stock inventory databases
  - shipping detail databases
  - payment processing such as credit cards or a provider like Paypal
  - data warehouse archives
  - accounting databases

  The different targets for data messages are really dependent on the size of the online operation. The larger the retailer, the more distributed their architecture becomes. This stuff is just too big to manage all in one place.
- **New data modeling techniques**: A Unified Dimensional Model (UDM) used for OLAP and analysis is data warehouse environments.

This type of feature helps performance in terms of overall end-user productivity, rather than SQL Server database performance specifically.

1.11 Where to begin?

Essentially, when looking at database performance, it is best to start at the beginning. Where is the beginning of a relational database? The data model logical design is the where a relational database is first created. The next chapter will examine relational data model design, as applied to improving overall SQL Server database performance.

1.12 Endnotes

1. Beginning XML Databases, Gavin Powell, Nov 2006, ISBN: 0471791202, Wiley.

2
Logical Database Design for Performance

In database parlance, logical design is the human perceivable organization of the slots into which data is put. These are the tables, the fields, data types, and the relationships between tables. Physical design is the underlying file structure, within the operating system, out of which a database is built as a whole. This chapter covers logical database design. Physical design is covered in the next chapter.

This book is about performance. Some knowledge of logical relational database design, plus general underlying operating system and file system structure, and file system functioning is assumed. Let's begin with logical database design, with database performance foremost in mind.

2.1 Introducing logical database design for performance

Logical database design for relational databases can be divided into a number of distinct areas:

- **Normalization**: A sequence of steps by which a relational database model is both created and improved upon. The sequence of steps involved in the normalization process is called *normal forms*. Each normal form improves the previous one. The objective is to remove redundancy and duplication, plus reduce the chance of inconsistencies in data and increase the precision of relationships between different data sets within a database.
- **Denormalization**: Does the opposite of normalization by undoing normal forms. It thus reintroduces some redundancy and duplication, plus increases the potential for inconsistencies in data, and so on. Denormalization is typically implemented to help increase per-

formance. The extreme in denormalization helps to create specialized data models in data warehouses.

- **Object Design**: The advent of object databases was first expected to introduce a new competitor to relational database technology. This did not happen. What did happen was that relational databases have absorbed some aspects of object modeling, in many cases helping to enhance the relational database model into what is now known as an object-relational database model.

Objects stored in object-relational databases typically do not enhance relational database performance, but rather enhance functionality and versatility.

To find out more about normalization [1] and object design [2] you will have to read other books as these topics are both very comprehensive all by themselves. There simply isn't enough space in this book. This book deals with performance tuning. So, let's begin with the topic of normalization, and how it can be both used (or not used) to help enhance the performance of a relational database in general.

So, how do we go about tuning a relational database model? What does normalization have to do with tuning? There are a few simple guidelines to follow and some things to watch out for:

- Normalization optimizes data modification at the possible expense of data retrieval. Denormalization is just the opposite, optimizing data retrieval at the expense of data modification.
- Too little normalization can lead to too much duplication. The result could be a database that is bigger than it should be, resulting in more disk I/O. Then again, disk space is cheap compared with processor and memory power.
- Incorrect normalization is often made obvious by convoluted and complex application code.
- Too much normalization leads to overcomplex SQL code which can be difficult, if not impossible, to tune. Be very careful implementing beyond 3rd normal form in a commercial environment.
- Too many tables results in bigger joins, which makes for slower queries.
- Quite often databases are designed without forehand knowledge of applications. The data model could be built on a purely theoretical

basis. Later in the development cycle, applications may have difficulty mating to a highly granular data model (highly normalized). One possible answer is that both development and administration people should be involved in data modeling. Busy commercial development projects rarely have spare time to ensure that absolutely everything is taken into account. It's just too expensive. It should be acceptable to alter the data model at least during the development process, possibly substantially. Most of the problems with relational database model tuning are normalization related.

- Normalization should be simple because it is simple! Don't overcomplicate it. Normalization is somewhat based on mathematical set theory, which is very simple mathematics.
- Watch out for excessive use of outer joins in SQL code. This could mean that your data model is too granular. You could have overused the higher normal forms. Higher normal forms are rarely needed in the name of efficiency, but rather preferred in the name of perfection, and possibly overapplication of business rules into database definitions. Sometimes excessive use of outer joins might be akin to: *Go and get this. Oh! Go and get that too because this doesn't quite cover it*.

The other side of normalization is of course denormalization. In many cases, denormalization is the undoing of normalization. Normalization is performed by the application of normal form transformations. In other cases, denormalization is performed through the application of numerous specialized tricks. Let's begin with some basic rules for normalization in a modern commercial environment.

2.2 Commercial normalization techniques

The terms *modern* and *commercial* imply a lack of traditional normalization. This also means that techniques used in a commercial environment are likely to bear only a vague resemblance to what you were taught about normalization in college or university. In busy commercial environments, relational database models tend to contradict the mathematical purity of a highly normalized table structure. Purity is often sacrificed for the sake of performance, particularly with respect to queries. This is often because commercial implementations tend to do things that an academic would never dream of, such as mix small transactions of an OLTP database with large transactions of a data warehouse. Some academics may not think too highly of a data warehouse dimensional model, which is essentially *denormalized up the gazoo*! Each approach has its role to play in the constant dance of trying to *get things right and turn a profit.*

2.2.1 Referential integrity

How referential integrity and tuning are related is twofold:

- **Implement Referential Integrity?** Yes. Too many problems and issues can arise if not.
- **How to Implement Referential Integrity?** Use built-in database constraints if possible. Do not use triggers or application coding. Triggers can especially hurt performance. Application coding can cause much duplication when distributed. Triggers are event driven, and thus by their very definition cannot contain transaction termination commands (COMMIT and ROLLBACK commands). The result is that their over use can result in a huge mess, with no transaction termination commands.

Some things to remember:

- **Always Index Foreign Keys**: This helps to avoid locking contention on small tables when referential integrity is validated. Why? Without indexes on foreign key fields, every referential integrity check against a foreign key will read an entire table without the index to read. Unfavorable results can be hot blocking on small tables and too much I/O activity for large tables.

Note: A hot block is a section of physical disk or memory with excessive activity—more than the software or hardware can handle.

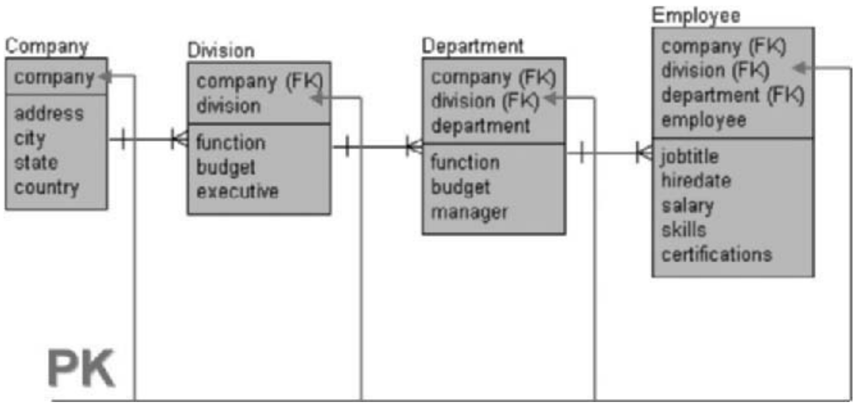
- **Avoid Generic Tables**: A table within a table. In some older database models, a single centralized table was used to store system information; for example, sequence numbers for surrogate keys, or system codes. This is a very bad idea. Hot blocking on tables like this can completely kill performance in even a low concurrency multi-user environment.

2.2.2 Primary and foreign keys

Using surrogates for primary and foreign keys can help to improve performance. A surrogate key is a field added to a table, usually an integer sequence counter, giving a unique value for a record in a table. It is also totally unrelated to the content of a record, other than just uniqueness for that record.

Primary and foreign keys can use natural values, effectively names or codes for values. For example, in Figure 2.1, primary keys and foreign keys are created on the names of companies, divisions, departments, and employees. These values are easily identifiable to the human eye but are lengthy and complex string values as far as a computer is concerned. People do not check referential integrity of records, the relational database model is supposed to do that.

Figure 2.1
Natural value keys



The data model in Figure 2.1 could use coded values for names, making values shorter. However, years ago, coded values were often used for names in order to facilitate easier typing and selection of static values—not for the purpose of efficiency in a data model. For example, it is much easier to type *USA* (or select it from a pick list), rather than *United States of America,* when typing in a client address.

The primary and foreign keys, denoted in Figure 2.1 as PK and FK respectively, are what apply referential integrity. In Figure 2.1, in order for a division to exist it must be part of a company. Additionally, a company cannot be removed if it has an existing division. What referential integrity does is verify that these conditions exist whenever changes are attempted to any of these tables. If a violation occurs an error will be returned. It is also possible to cascade or pass changes down through the hierarchy. In other words, when cascade deleting a company all divisions, departments, and employees for that company will be removed from the database, as well as the company itself.

In a purist's or traditional relational data model, keys are created on actual values, such as those shown in Figure 2.1. The primary key for the Company table is created on the name of the company, a variable length string. Try to create keys on integer values. Integer values make for more efficient keys than alphanumeric values. This is because both range and exact searches, using numbers, are mathematically much easier than using alphanumeric values. There are only 10 possible digits to compare (0 to 9), but many more alphanumeric characters than just 10. Sorting and particularly hashing are much more efficient with numbers.

Avoid creating keys on large fixed or variable length strings. Dates can also cause problems due to implicit conversion requirements, and differences between dates and timestamps. Numbers require less storage and thus shorter byte lengths. The only negative aspect is that as the numbers grow, predicting space occupied becomes more difficult.

A possibly more efficient key structure for the data model in Figure 2.1 would be as shown in Figure 2.2. In Figure 2.2, all variable length strings are relegated as details of the table, by the introduction of integer primary keys. Integer primary keys are known as surrogate keys, the term *surrogate* meaning substitute value or key.

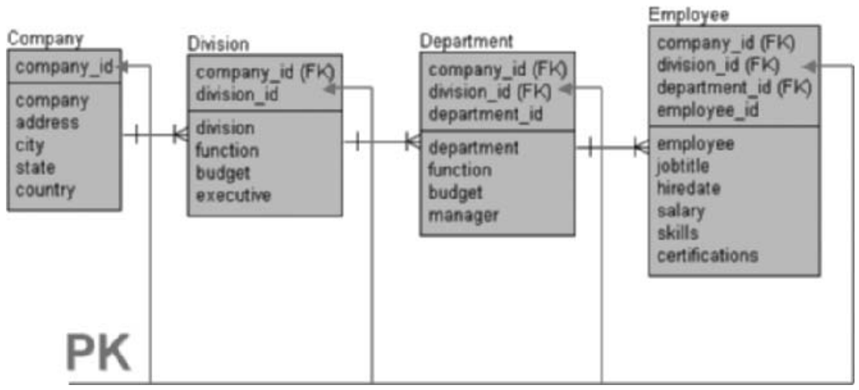


Figure 2.2 *Surrogate value keys*

A more effective form of the data model shown in Figure 2.1 and Figure 2.2 would be the data model as shown in Figure 2.3. In Figure 2.3, the relationships become non-identifying.

An identifying relationship exists where a parent table primary key is part of a child table primary key (the parent identifies each child directly). A parent table primary key, which is not part of the primary key in a child table, makes the relationship non-identifying. This is because each record in the child is not directly and exclusively identified by the primary key in the parent table.

In Figure 2.3, all foreign keys have been removed from the composite primary keys, resulting in a single unique primary key surrogate key field. This is often the most effective structure for an OLTP database. However, where there is heavy reporting, it can sometimes be beneficial to retain composite indexing structures like this, but perhaps as alternate keys and not as part of the primary key.

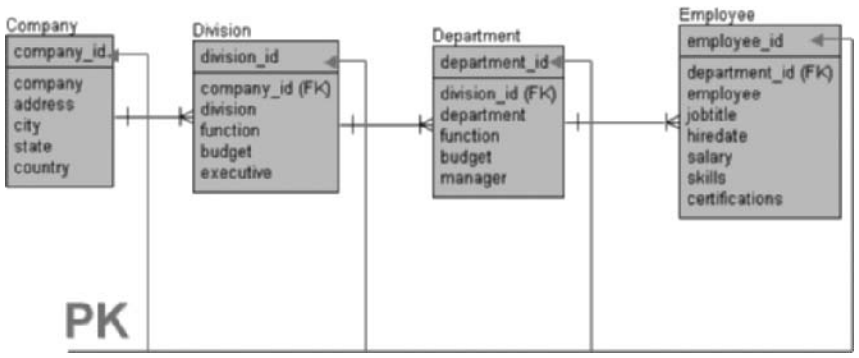


Figure 2.3
Single field, unique surrogate keys

In most relational databases, surrogate keys are most efficiently generated using autonomous sequence counter objects, or auto counters.

2.2.3 Business rules in a relational database model

Business rules are the rules defining both database metadata structure, and some of the data manipulation coding (commonly seen in application coding—not in the database model). The idea of modern relational databases, such as SQL Server, is that you do have the option of implementing all business rules pertaining to a set of data, and large parts of the noncustomer facing part of applications, within a database model. This does not mean that you should do so.

Business rules can be implemented in a relational database model using the combination of tables, fields, data types, relationships between tables, procedures, triggers, specialized objects, and so on.

What are business rules? This term is often used vaguely where it could quite possibly have different meanings. Some technical people will insist on placing all business rules into a database. Others will insist on the opposite. Either approach is often dependent on available skills of personnel working on a particular project.

There are two important areas with respect to business rule implementation inside a relational database model:

- **Referential Integrity**: Referential integrity can be implemented in the database using constraints, triggers, or application coding. Usually the best option is use of constraints because they are centralized and simplistic.
- **SQL code**: Using stored procedures, functions, and sometimes triggers.

You can put as much as you want of the business rules of an application, into a database model. Sometimes this can work well for performance. In a practical situation it makes the database model just too complex. It is better to divide up structural and processing requirements between database and application coding. There is a very simple reason for this: application coding is much more proficient at processing data than any database is.

A database is primarily built to both store data and to read and write data. A database is not built to process data.

The term process means to change data from one thing into another. This function is often called number-crunching in the case of complex numerical calculations.

An application coding tool is built to process data, not store it. In short, use the appropriate tools in the appropriate places. This is fundamental to building applications and databases with acceptable performance.

2.2.4 Alternate indexes

Alternate indexes are sometimes known as secondary indexes. The meaning of the terms alternate and secondary is somewhat confusing because they mean different things in English. In the context of a relational database, these terms are one and the same. Alternate implies *another option*, and secondary implies *in addition to*. So now complete confusion reigns supreme. The precise meaning of the terms in this context is not really too important.

Alternate indexes are usually created because the primary and foreign key indexes in a data model do not cater to everything required by applications. That *everything* would be any kind of filtering and sorting, particularly in SQL statements, such as queries. Filtering and sorting SQL statements is what uses indexes. Those indexes help applications to access the database quickly, scanning over much smaller amounts of physical disk space, than when reading an entire table.

A database with too many alternate indexes will have more to maintain. For example, whenever a record in a table is changed, every index created against that table must be changed at the same time. So, inserting a single record into a table with four indexes in reality requires four index additions and one table addition. That is a total of five additions, not a single addition. That's a lot of work for a database to do. That can hurt performance, sometimes very badly. On the contrary, if a database has a large number of alternate indexes, there could be a number of reasons, but not necessarily all performance friendly scenarios:

- **Reporting in an OLTP database**: More traditional relational database structures, such as those shown in Figure 2.1 and Figure 2.2, include composite primary keys. Composite primary keys get a little too precise and are more compatible with reporting efficiency and perhaps only a very few specific reports. Structures such as that shown in Figure 2.3 do not have composite primary keys. Imposition of reporting requirements on the structure in Figure 2.3 would probably require composite alternate indexes, somewhat negating the usefulness of surrogate keys.
- **Matching query filtering and sorting with existing keys**: When existing composite primary keys do not cater to requirements then perhaps either further normalization or denormalization is a possibility. Or perhaps current structures simply do not match application requirements. However, changing the data model at a late stage (post-development) is difficult. Also, further normalization can cause other problems, particularly with recoding of application and stored procedure code.
- **Direct copying between development and production databases**: Developers have a habit of creating a lot of indexes while coding, as they should do. They are building and testing after all. Quite often indexes will not be cleaned up. And they may not be removed when no longer in use.

- **Surrogate keys do not negate uniqueness requirements**: When using surrogate keys, such as in Figure 2.3, items such as names of people or departments may be required to be unique. These unique keys are not part of referential integrity but they can be important and are most easily maintained at the database level using unique alternate indexes.

There is no cut-and-dry answers to which alternate keys should be allowed, and how many should be created. The only sensible approach is to keep control of the creation of what are essentially *extra* keys. These keys were not thought about in the first place. Or perhaps an application designed for one thing has been expanded to include new functionality, such as reporting.

Alternate indexing is not part of the normalization process. It should be somewhat included at the data model design stage, if only to avoid difficulties when coding. Programmers may swamp database administrators with requests for new indexes, if those indexes were not added when the data model was first created. Typically, alternate keys added in the data modeling stage are those most obvious as composites, where foreign keys are inherited from parent tables, such as the composite primary keys shown in Figure 2.1 and Figure 2.2.

Composite field keys use up a lot of physical space, somewhat negating the space saving I/O advantage of creating an index in the first place. Also, a composite index will create a more complicated index. A more complicated index makes the job of specialized index searching algorithms a more difficult task.

The sheer scale and size of OLTP Internet databases can sometimes be horrifying to even the most seasoned of database administrators. Many hundreds of gigabytes are common. Some modern OLTP databases are even on the terabyte scale. Some installations even mix OLTP and reporting processing into the same database, even to the point of mixing intense OLTP concurrency and heavy I/O data warehousing activity. Enormous amounts of money can be spent on very expensive hardware to maintain performance at a level acceptable to customer satisfaction. And, developers cannot possibly test reliably against databases of such magnitude. The result is often applications coded to small-scale databases, and unexpected, if not disappointing, performance in much larger scale production environments. Extensive tuning is often required. The larger the database, the more likely that a dual database architecture has to be adopted. That dual database architecture would be OLTP plus data warehouse architecture in separate databases, on different computers. Unless of course you prefer to spend a small fortune on fancy computer hardware.

2.3 Denormalization for performance

Before we get to data warehouse data modeling, you need to have a brief understanding of what denormalization is. A large part of denormalization is a process of undoing normalization. This is a process of removal of too much data model granularity. Too much granularity may be preventing acceptable performance of data retrieval from a relational data model. This is true not only for reporting and data warehouse databases, but also quite significantly even for OLTP databases.

In SQL Server the SQL Analysis Server is effectively a data warehousing query analysis tool.

So, as already stated, denormalization is largely a process of undoing normalization. However, that does not cover all practical possibilities. Here are some areas of interest when it comes to undoing of granularity, in addition to that of undoing of normalization:

- **Denormalization**: Undoing various normal forms, some more than others.
- **Context**: Various tricks to speed up data access based on the context of data or application function.
- **Denormalization using unusual database objects**: Creation of special types of database objects used to cluster, pre-sort, and pre-construct data. These specialized objects help to avoid excessive amounts of complex repetitive SQL.

The reason for removal or degradation of normalization is for the purpose of improving performance. As already stated, it is very important to note that removal of normalization granularity is usually a necessity in data warehousing, and sometimes even smaller scale reporting environments. It has often been the case, in my experience, that even OLTP transactional data models have required reduction in granularity produced by overnormalization. Any application retrieving data from an overnormalized database can potentially have performance problems. A very deeply normalized database is only most effective for single record insert, update, and deletion transactions; changing single data items.

In the past I have even seen some very poorly performing OLTP applications, even those with small transactional units. The reason for this poor performance is often the result of severely granular data models. Some of these systems did not contain large numbers of records or occupy a lot of physical space. Even so, brief data selection listings into one page of a browser, more often than not performed join queries comprising many tables.

These types of applications sometimes fail, their creators fail or both. In these cases, there is a very strong case to be said for the necessity of designing a data model with the functionality of the application in mind, and not just the beauty of the granularity of an enormously overnormalized relational data model structure.

One particular project I have in mind had some queries with joins containing in excess of 15 tables, a database under 10 megabytes in size, and some query return times of over 30 seconds on initial testing. These time tests were reduced to mere seconds after extensive application code changes of embedded SQL code (to get around an impractically over normalized table structure). However, it was too late to change the data model without extensive application recoding, and thus some very highly tuned and convoluted embedded SQL code remained. Maintenance for this particular application will probably be a nightmare, if it is even possible. In a situation such as this, it might even be more cost effective to rewrite.

In conclusion, normalization granularity can often be degraded in order to speed up SQL data retrieval performance. How do we do this? The easiest and often the most effective method, from the perspective of the data model, is a drastic reduction to the number tables in joins. It is possible to tune SQL statements joining 15 tables but it is extremely difficult—not only to write SQL code but also for the database optimizer to perform at its best. If the data model can possibly be changed then begin with that data model. Changing the data model obviously affects any SQL code, be it in stored procedures or embedded in applications.

Some basic strategies when attempting to tune a logical database (the data model), could include the following approaches:

- **Application tasks versus database tables**: Compare the number of tasks (or application interface screens) in an application and compare with the number of tables. If there are many more tables than application task units, then you might want to denormalize or simply remove some of the tables. There may well be some redundant tables in the data model. For example, a database originally designed to store aircraft parts is later changed to a seating reservation system. This is probably an unrealistic scenario but retaining tables describing tolerances for aircraft parts is irrelevant when reserving seats for passengers on flights of those aircraft.

- **Focus on heavily used functionality**: Focus on the heavily active tables in a data model. The simple reason is that attacking the busiest parts of any system is likely to gain the best results, the quickest.

2.3.1 What is denormalization?

In most cases, denormalization is the opposite of normalization. Normalization is an increase in granularity by removing duplication. Denormalization is an attempt to remove granularity by reintroducing duplication, previously removed by normalization.

Denormalization is usually required in order to assist performance because a highly granular structure is only useful for retrieving very precise, small amounts of information, rather than large amounts of information. Denormalization is used to analyze and report, not to facilitate changing specific data items. In simple terms denormalize to decrease the number of tables in joins. Joins are slow! Simple SQL statements are fast and they are easy to tune, being the order of the day wherever possible.

It is sometimes the case that table structure is much too granular, or possibly even incompatible with structure imposed by applications. This particular situation can occur when the data model is designed with the perfection of normalization in mind, without knowledge of, or perhaps even consideration for, realistic application requirements. It is a lucky development team that understands application requirements completely, when the data model is being built. Denormalization is one possible solution in this type of situation.

Denormalization is not rocket science. Denormalization is antonymic with normalization. In other words, the two are often completely opposite. Both are common sense.

2.3.2 Denormalizing the already normalized

From my own past experience and that of others, there are numerous things you can look for when considering what to denormalize in a data model.

2.3.2.1 Multiple table joins (more than two tables)

A join is a query that merges records from two tables based on matching fields. When more than two tables are in a single join, it gets more difficult for a relational database to find all the records efficiently. If these types of joins are executed frequently, the result is usually a general decrease in performance. Queries can sometimes be speeded up by denormalizing multiple tables into fewer tables. The objective would be to reduce the number of tables in join queries overall. Query filtering (WHERE clause) and sorting (ORDER BY clause) only add to the complexity of queries. The following query is an ANSI format query showing a horribly complex join of 8 tables:

```
SELECT cu.customer_id, o.order_id, ol.seq#, ca.category_id
FROM customer cu JOIN orders o ON (cu.customer_id = o.customer_id)
 JOIN transactions t ON (o.order_id = t.order_id)
  JOIN transactionsline tl ON (t.transaction_id = tl.transaction_id)
   JOIN ordersline ol ON (o.order_id = ol.order_id)
    JOIN stockmovement sm ON (tl.stockmovement_id = sm.stockmovement_id
    AND ol.stockmovement_id = sm.stockmovement_id)
     JOIN stock s ON  (s.stock_id = sm.stock_id)
      JOIN category ca ON (ca.category_id = s.category_id)
WHERE ca.text = 'Software';
```

The above query is completely ridiculous but it is the sort of complexity that you might want to search for. The data model for the above query is shown in Figure 2.4.

2.3.2.2 Multiple table joins finding a few fields

Sometimes a join gets at many tables in a data model and only retrieves as little as one field, or even just a few fields, from a single table in a multiple table join. A join could be passing through one or more tables, from which no fields are retrieved. This can be inefficient because every table passed through adds another table to the join. This problem can be resolved in two ways:

- By denormalizing tables to fewer tables. However, this may be difficult to implement because so much denormalization could be required, that it may affect functionality in a lot of application code
- Maintain copies of the offending field values in more than a single table.

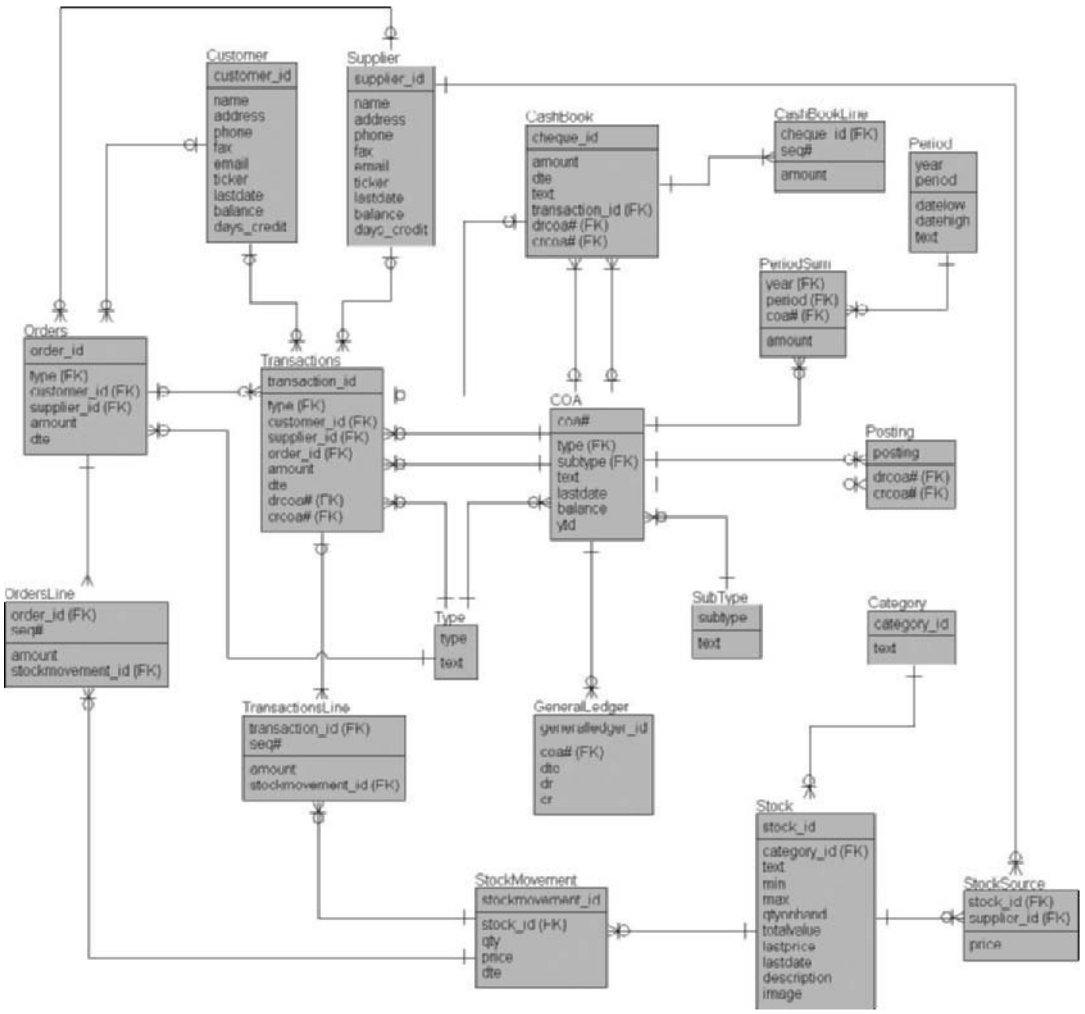


Figure 2.4 *A relational data model for a simple accounting database*

In the next script is the same ANSI format query as above, but with a small change. In the join query below, only the customer_id and category_id fields are retrieved. Of course, the join is still ridiculous. It probably would never be a requirement for an application but this is the only way that the stock category can be linked to a customer. This is definitely a problem. In this example, denormalization would be nonsensical but some type of a relationship could possibly be established between the customer and stock tables. On the other hand, new table relationships would only serve to complicate the data model further:

```
SELECT cu.customer_id, ca.category_id
FROM customer cu JOIN orders o ON (cu.customer_id = o.customer_id)
 JOIN transactions t ON (o.order_id = t.order_id)
  JOIN transactionsline tl ON (t.transaction_id = tl.transaction_id)
   JOIN ordersline ol ON (o.order_id = ol.order_id)
    JOIN stockmovement sm ON (tl.stockmovement_id = sm.stockmovement_id
    AND ol.stockmovement_id = sm.stockmovement_id)
     JOIN stock s ON  (s.stock_id = sm.stock_id)
      JOIN category ca ON (ca.category_id = s.category_id)
WHERE ca.text = 'Software';
```

Again, the data model for the above query is shown in Figure 2.4.

2.3.2.3 The presence of composite keys

Do tables in your data model have composite keys? It is possible to partially denormalize by adding composite key elements to the primary keys of subset tables. Composite keys are totally contrary to object structure and more compatible with reporting. Applications written using object-oriented programming languages, such as Java, may perform poorly when data is accessed using composite keys. Object applications perform best when accessing precise data objects, containing collections of other precisely defined objects.

This is what an object design is and is common for front-end application development tools such as Java, JavaScript, .NET objects, and even XML. In other words, modern object-oriented applications are more compatible with surrogate keys in the data model. This is especially the case when applications control database access, and customers do not have direct access to the tables in your database. Now something like Analysis Services provides an overlay structure to tables in a database, and typically does not provide direct access to underlying tables. If end-users and power users do have direct access to your normalized table structure, then you probably don't need to read this book. If not, then be warned that database performance tuning is a highly complex topic, with many variables that even an experienced database performance tuner can't predict. Hire some help if you need to, and keep your customers.

Composite indexes that are primary keys fall into the realm of alternate indexing.

The use of composite keys has already been discussed between Figure 2.1, Figure 2.2, and Figure 2.3. Figure 2.3 represents the most efficient and compatible form of unique primary key structures, in that Figure 2.3 uses a surrogate key (or replacement key) in each table. In other words, the surrogate acts as a replacement for any possible combination of fields, which uniquely identify each record, within each table.

2.3.2.4 One-to-one relationships

One-to-one relationships are often the result of overapplication of BCNF (Boyce-Codd Normal Form), or 4th normal form. Look for them! They are usually used to preserve disk space and provide integrity for potentially NULL values. They may be unnecessary if the required removal of NULL values causes costly joins. Disk space is cheap! An example of this is shown in Figure 2.5.

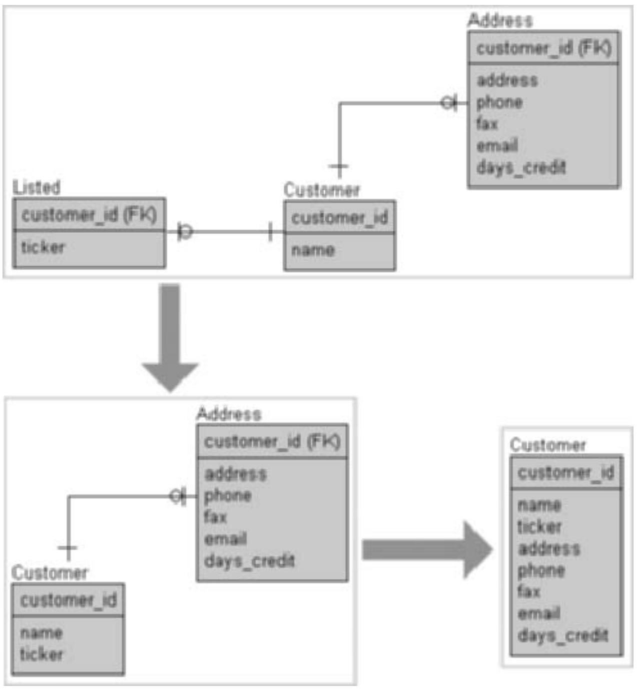


Figure 2.5 *Denormalizing one-to-one NULL separation relationships*

In Figure 2.5, the fully denormalized version is shown bottom-right, in the form of the customer table, containing both address and listing information. This is based on listing information potentially not being present, and address information potentially not being present. Splitting the customer table into customer, listed, and address tables is correct. And it is proper normalization. However, if 99% of all queries in your database find both the name of the customer and their address, then you are simply over stressing your applications in general with constant repetition of a join between customer and address tables. On the contrary, if 99% of queries find only a customer's name without their address, then this level of normalization is warranted. Applications should help to influence the normalization process. So, database design is usually most effective with as much knowledge of future applications as possible.

Finding address information without the customer name is unlikely in an OLTP database. It is more likely for a data warehouse. OLTP databases require precisely identifiable, small amounts of information (a single customer). Data warehouses seek to analyze large quantities of data by one or more criteria, such as one or more regions in a country.

Removal of one-to-one relationships can also be pictured mentally as disallowing related static data to exist in multiple tables.

Application of BCNF (Boyce-Codd Normal Form) creates unnecessary one-to-one relationships, within static data tables.

BCNF is intended to separate candidate keys into separate tables. In other words, any field in a table, which has potential as a primary key (is capable of representing a record in a table uniquely), should not be allowed to be dependant on the primary key. It should be its own primary key, in its own table, and link back to the parent table through a one-to-one relationship. An extreme application of BCNF is shown in Figure 2.6, where I have created a rather absurdly broken up picture of BCNF, separating out all unique values (candidate keys) into separate tables.

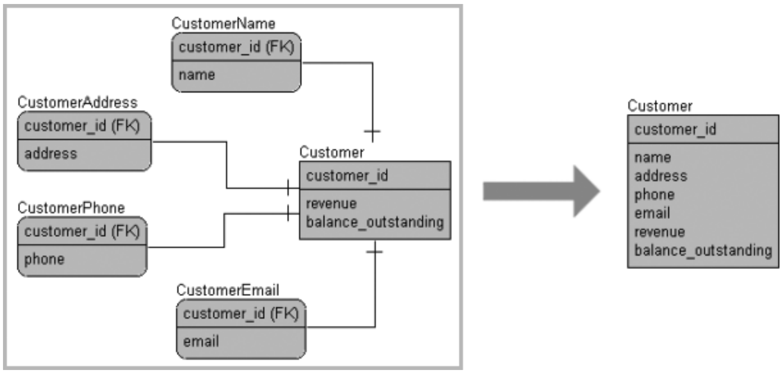


Figure 2.6
Denormalize absurd applications of BCNF

2.3.2.5 Denormalize static tables

Similar to that shown in Figure 2.5, but not identical because the relationships are not all one-to-one, are queries against multiple static tables performing a lot of joins. Querying a single table is much more efficient than multiple tables. Static data does not need to be normalized into separate tables unless said separate tables are accessed individually, from separate queries. Even so, static tables are often small and generally will be full table scanned by the optimizer anyway, regardless of any indexing. A good example of this is represented by the listing (Listed) and exchange (Exchange) tables in Figure 2.7. The listing and exchange tables do not need to be separate tables.

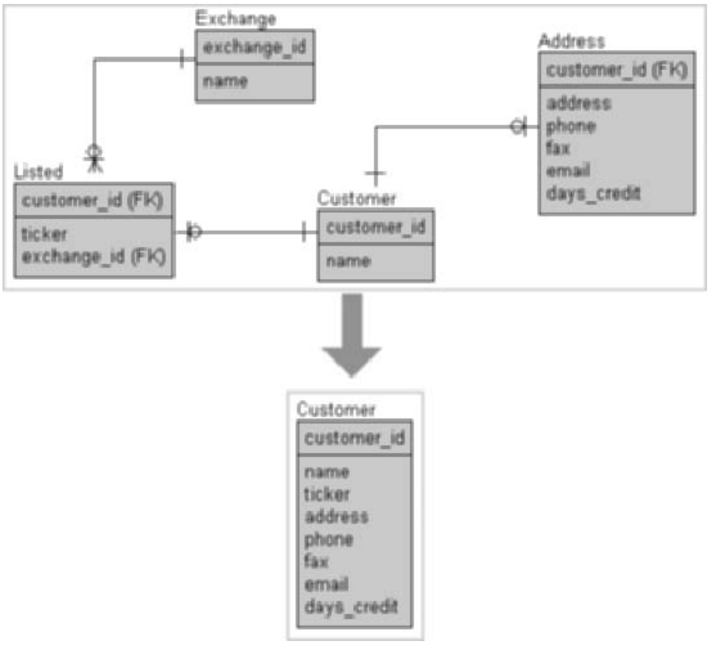


Figure 2.7 *Static tables should be fully denormalized*

Just to be more precise, and prevent any confusion, in Figure 2.7 the denormalized customers at the bottom of the diagram contains a ticker symbol but no exchange. Ticker symbols are in actuality unique regardless of the exchange. For example, the company International Business Machines is listed on the NYSE with the ticker symbol IBM. It is also listed on the London Stock Exchange as IBM, because it's the same company. If the ticker symbol is different, you would be looking at a different company with respect to stock trading. The Exchange table was added for the purposes of demonstrating a point.

2.3.2.6 Reconstructing collection lists

An object collection inside a table implies a multi-valued list, or a comma-delimited list. This comma-delimited list has zero or more elements, with same data type values repeated in a single field of a table. For example, 1,2,3,4,5 is a comma-delimited list consisting of five numbers. Rome, London, Hamburg, Paris, Nice is a comma-delimited list of five European cities.

A multi-valued list is directly dependent on the primary key as a whole, but not as individual elements within the list. Figure 2.8 shows normalization (from left to right) of two multi-valued lists. Employee skill and certification collections are removed into separate tables. An employee could have skills, or certifications, or both. So, there is no connection between the fields of the employees table, other than the employee number, and the details of skills or certifications for each employee.

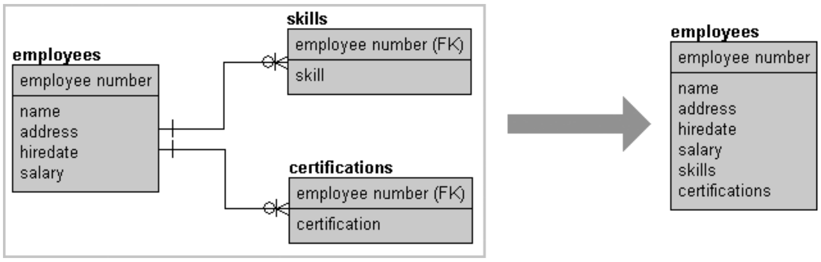


Figure 2.8 *Normalizing contained object collections*

Figure 2.8 shows that one table has been divided up into three tables. In the interests of good join query performance, Figure 2.8 is exactly what you don't want to do. Application programming languages are highly effective at managing comma-delimited lists, and all sorts of other lists for that matter. There is no reason on earth why this kind of deep business logic needs be buried within a relational database.

This type of normalization is usually 4th normal form normalization, which strives to eliminate multi-valued dependencies. In modern commercial applications, this kind of depth is unnecessary and can be detrimental to performance. Let your application programmers code for this type of data in something like Java. Perhaps just try to use the best tools for each particular job.

2.3.2.7 Removing tables with common fields

3rd normal form can be used to remove fields, shared on two or more tables, making a single copy of the shared fields in a new table. The result is shown on the left side of Figure 2.9, where common fields are shared between multiple tables, using an extra table. Once again, the denormalized version of this transformation, as shown on the right side of Figure 2.9, is possibly a more efficient form.

And yet again, the simple reason is reducing the number of tables in join queries to be constructed to read records from these tables. It is sometimes better just to duplicate the fields in their parent tables. Of course, not duplicating and normalizing may be determined by, in how many tables the fields are repeated. Additionally, the context of the data may be important as well.

In the case of the customers and suppliers shown in Figure 2.9, it can possibly improve to denormalize something the way it is shown. Relative to invoices and transactions in an accounting system, customers and suppliers are relatively static data. So, normalizing in this context is splitting static data unnecessarily.

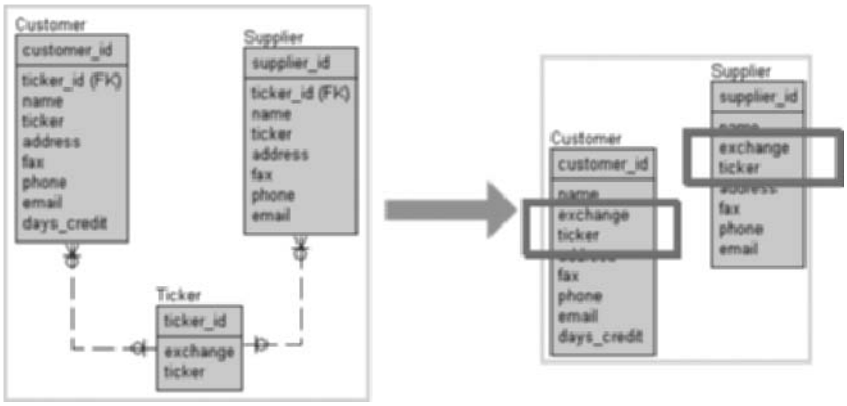


Figure 2.9 *Denormalize common fields by duplication*

2.3.2.8 Reincorporating transitive dependencies

Another type of 3rd normal form transformation can remove a transitive dependency, as shown on the left side of Figure 2.10. A transitive dependency is when a third field is dependent on a second field. Also, the second field is dependent on the primary key but the third field is not fully dependent on the primary key. So, the third field can be removed to a new table, as shown on the left side of Figure 2.10. The result is that the second field becomes the primary key of the new table. In other words, on the left side of Figure 2.10, each department depends on division (it's in a division), which in turn depends on being in a company. However, each department does not absolutely depend on being part of any particular company (different companies can have the same department names).

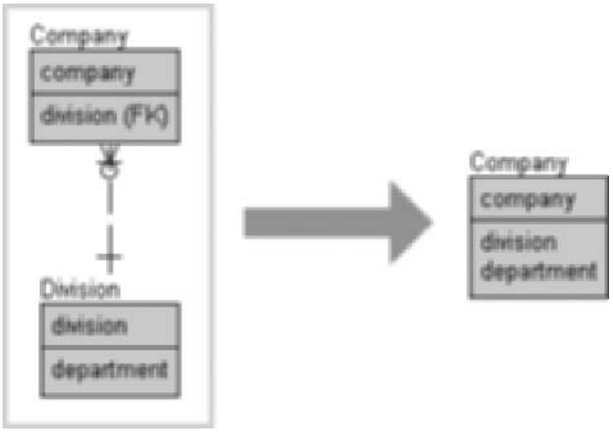


Figure 2.10 *Reincorporate transitive dependencies*

Once again, in the interests of efficient join queries, the denormalized form, as shown on the right side of Figure 2.10, might be the most prudent option.

In the interest of simplicity, I have not labeled primary key column names in my standard format for surrogate keys, as column_id. However, the company column in the Company table on the right side of Figure 2.10 is a surrogate key. Otherwise, the division column should be part of a composite primary key of both division and company.

2.3.3 Denormalizing by context

There are many tricks to denormalize data, which are not necessarily reversals of any of the steps of normalization. These we examine in this section. There could be similarities between *denormalization by context*, and that of the reversal of normal forms. However, the fitting of scenarios, to a specific normal form rule, rather defeats the purpose of creating the rules in the first place. Why qualify something when it is already done? Application of theory by way of proof of that theory is used to create the environment, which that theory has described. Normal forms create the environment of perfectly structured relational data. The theory is normalization. Normalization works properly. Then again, in commercial environments the critical factors are speed and customer satisfaction, not perfection—usually far from it!

2.3.3.1 Copies of single fields across tables

The sheer depth of some data models can necessitate searching through multiple tables just to find sometimes even as little as one or two fields, and even from a single table. Again, the performance objective is to avoid too

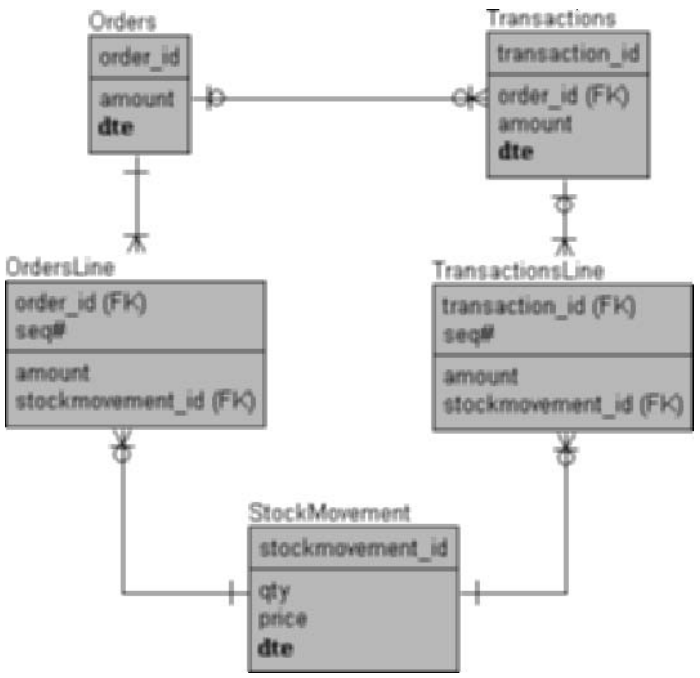


Figure 2.11 *Duplicate fields across distantly related tables*

many tables in join queries, thereby increasing performance. Figure 2.11 is a small section of the data model shown in Figure 2.4.

In Figure 2.11, the field called DTE (containing a date value) is present in three different tables. Realistically an order will be placed completely, or a transaction will be executed such as sending an invoice, all at the same time. This is regardless of how many stock items are moved in or out of stock. So, it makes perfect sense to include the date with each order and transaction. However, this particular data model combines the activities of accounting data, stock tracking (movements in and out of stock), plus an ordering-invoicing system. So, when taking stock, if dates are required by stock movement records, that date must be retrieved from orders or transactions. Both orders and transactions are irrelevant to deciding which stock items need to be ordered from suppliers. The date is relevant to stock movements because when an order is placed, those items are not necessarily from stock until invoiced. Or they may also be sent to a customer before invoicing. Or perhaps even removed when payment is received – which is after invoicing. So, in this case dates are copied into the stock movements history file.

2.3.3.2 Summary fields in parent tables

Summary fields can be used to add up or aggregate a repetitive value, from a child table into a parent table. These aggregates can be placed into parent tables to summarize information held in child tables. So, in the case of the data model section (from Figure 2.4), shown in Figure 2.12, every transaction (a transaction is an invoice) contains lines (items on each transaction).

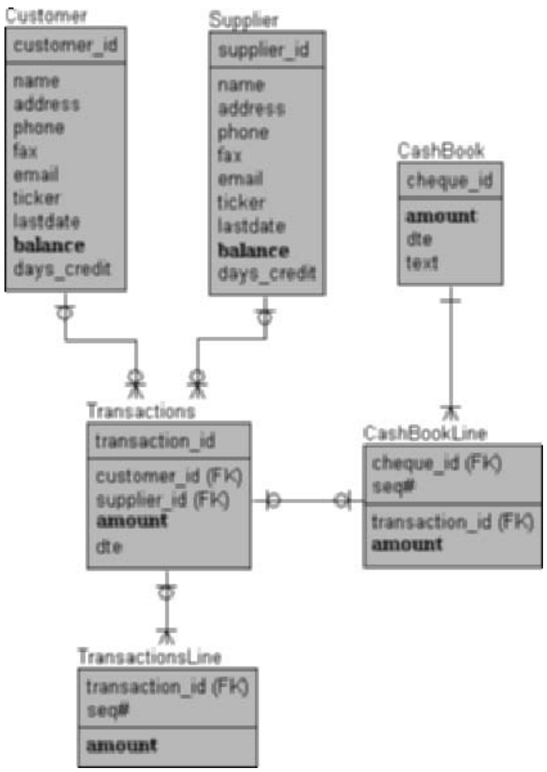


Figure 2.12
Summary fields in parent tables

When you go to a store and buy three items there are three different items on your receipt. The receipt is a little like an invoice. Each line item has an amount (how much it cost you). Also, your total transaction has an amount, which is actually the sum of all the items you purchased, plus sales tax of course. Now if you were using a credit card then your credit card will send you a statement at the end of the month showing all purchases (transactions) for the entire month. So, your credit card has a total balance owed by you to your credit card company.

The point to make in this situation is that all amounts and balances can be recalculated by adding up all individual line items, Calculations can be executed again, every time some kind of summary is required. In many situations, it is more efficient to store summary values in each parent table. Bear in mind that this data model sits in a single database, and is thus an enclosed system. In other words, the scenario of the store, your receipt, and your credit card company is quite inappropriate from a practical perspective because the store and the credit card company—are different companies—they do not share the same database, or even the same building.

The only potential problems with maintaining summary amounts is what is called hot block locking. Consider that thousands or even millions of transaction lines are generated by credit card companies every second. In this case, maintaining summary amounts could cause conflict because lots of credit card users are generating summary amount updates all at once. This wouldn't happen at the customer level, but when understanding how a database is physically structured, many customer records could be stored in a single database block (because each customer occupies a small byte chunk of physical data space). So, lots of people are trying to update summaries all at once. Again, for a credit card company maintaining a balance for each customer, locking is not likely to be an issue. This type of denormalization is context sensitive. For example, if a new parent table summarized all amounts owed, by a company like American Express, for all states across the United States, then the context would be quite different indeed. In this situation, there could be millions of customers updating a summary of states table, throughout the table, and all at once—24 hours per day. Not a good idea!

2.3.3.3 Separating data by activity and application requirements

The classic example of data separation, of active and inactive data, is that of a data warehouse. Data warehouses can be used to maintain historical data over many years. Those data warehouses can grow to be thousands of times larger than the customer facing OLTP databases. It is much easier to find a customer's transactions for a credit card statement, for one customer, when searching transactions in a single year's transactions. If the data contained the last 20 years of credit card transactions, every statement would take 20 times longer to find. When considering that American Express perhaps prints millions of statements per month, then multiplying quantities of data by 20 can be quite a staggering thought.

Essentially OLTP data is active and data warehouse data is inactive data. Part of the reason for separating data into data warehouses is because the OLTP data model requires fast action for customers—with everything divided into small pieces (normalized). Data warehouses on the other hand like to put things back together into meaningful reports (not small pieces but great big chunks of data). Not only does this cause serious conflict with OLTP requirements, but reports that continually stitch data back into joins from OLTP tables are just too slow.

Recent years have seen the power of computer hardware and software grow, such that OLTP and data warehouse functionality is often placed onto the same computer, or even in the same database, but unlikely in the same table structures.

Even with using a data warehouse to store historical data, quite often older data is constantly phased out or just deleted. And sometimes older data is stored in separate tables to that of the customer facing OLTP database tables. Even copies of tables can be used. This is done for the same reasons mentioned above, when creating a separate data warehouse. That reason is separation of active and inactive data to prevent performance issues, as a result of conflicting requirements placed on active versus inactive data.

Similar to separating active and inactive data is the separation of heavily and lightly accessed fields within the same table. Tables can be split based on varying degrees of access to fields. This can help to avoid unnecessary frequent access to rarely used data.

2.3.3.4 Local application caching

Caching means to temporarily store data, from a database, somewhere in memory. Obviously, you would want to cache data that is frequently used, changed as little as possible, and is small in size. The perfect type of data for caching is static data. Static data is generally physically small, it is much in demand, and it is rarely changed.

Also, you can store static data in memory on a database server computer or on a middle tier computer such as an application or web server. You can even send copies of static data to client computers, even those on the other side of the world, connected to your database over an Internet connection.

2.3.4 Denormalizing and special purpose objects

Some relational databases allow creation of specialized objects, allowing storage of things like summaries, and copies of data. The result is reduction in conflicting requirements of customer facing OLTP applications, and reporting such as in a data warehouse. The ultimate objective is increasing overall database performance. The simplest forms of copies are copies of tables, views, or temporary tables. Let's examine different types of database objects in use in relational databases:

- **Views**: A view does not store data but stores a query. That query can be executed at will, by querying the view whenever you want. This is not a performance issue in itself. One of the problems with views is that they are often used to filter and sort data from tables. When a view is read, someone writing the query should be aware of how the underlying view query works, in order to avoid extra work in the view and extra work in the query against the view. The worst scenario is major reduction filtering or sorting against a view, which finds all records in a large table anyway. The equivalent filter against a table would read only the filtered portion. The view will always read the entire table. Where the underlying table is extremely large, 1 record read from the view could still retrieve millions of records from the underlying table because only the view uses no filtering. Another problem with views is the tendency for developers to sometimes overuse them, embedding layer upon layer of views within each other. This can simply become too complex for any database to handle. This is an example of a view:

```
CREATE VIEW VIEW_ChartOfAccounts AS
 SELECT coa.coa#, coa.type, coa.subtype, coa.text,
  sum(gl.dr), sum(gl.cr)
 FROM coa, generalledger gl
 WHERE coa.coa# = gl.coa#
GROUP BY coa.coa#, coa.type, coa.subtype, coa.text;
```

The above script is pseudocode and creates a view, which joins two tables shown in the data model of Figure 2.4.

Note: Pseudocode is a type of coding used to explain a concept. Pseudocode will not function 100% in any particular environment, database, programming language, or on any specific hardware platform. The intention of presenting a concept using pseudocode is to demonstrate—not to provide bulletproof example coding.

- **Temporary tables**: These can be used to form intermediary functions per session and are manually or automatically emptied and dropped after each use.

  A session is a connection to a database, between a user and a database.

Temporary tables can sometimes help to eliminate intermediary functionality and eliminate duplication. The result is less I/O activity on primary tables because temporary tables are read after they have first been filled. Temporary tables are superseded in modern relational databases by other more exotic and more versatile database objects, such as materialized views. This is a simple pseudocode example of a temporary table created to drop all its records when the session creating it terminates:

```
CREATE TABLE TEMP_ChartOfAccounts AS
 SELECT coa.coa#, coa.type, coa.subtype, coa.text,
  sum(gl.dr), sum(gl.cr)
 FROM coa, generalledger gl
 WHERE coa.coa# = gl.coa#
GROUP BY coa.coa#, coa.type, coa.subtype, coa.text
DELETE ON DISCONNECT;
```

- **Intermediary tables**: A table with more permanent data content than a temporary table. That data is retained for different sessions to make use of and would like be deleted manually when so required. These tables could also be used to contain summary information. Summary records consist of groups of many other records.

  Materialized views and clusters can also be used, which are regenerated periodically and automatically. These types of objects are commonly used in data warehouses but can sometimes be useful in OLTP databases.

  Intermediary tables are useful as long as less than real time response is acceptable for the summarized information.

- **Cluster**: Places most commonly accessed indexed fields, sometimes including some data fields, together into the same physical space. They would also be commonly ordered in a specified order. This is a method of splitting active and inactive data by fields, making more frequent use fields more readily available, in pre-sorted, indexed, and clustered form. So, the most active data is copied to a presorted, preorganized cluster. Less active data remains only in underlying table (or tables), from which the cluster has been created. Clusters help performance by organizing in desired order, and allowing for less I/O time on the most highly active fields in a table.

- **Clustered and non-clustered indexes**: A clustered index contains actual data values in leaf nodes. A non-clustered index contains addresses to data values, in leaf nodes, not the actual data values in an index. Both these types of indexes use a BTree structure. So, a clustered index creates an index out of the fields in a table, sorting table data in the order of the index, and physically stored in the order of a specified group of one or more fields (in the same table). In other words, it turns a table into an index, but retaining all the fields in the table. The result is a table logically sorted based on the index. This can help with performance only when reading the data in the order of the index. Updating can be relatively efficient, depending on how flexible the index structure is. BTree indexes are generally amenable to frequent small changes. Bitmap, hash key, and ISAM indexes cause changes to be made outside of the indexed structure—this is known as overflow. Subsequent reads into overflow space cause *bouncing around the disk*, which is highly inefficient.

  A clustered single table index is also known in some relational databases as an index organized table (IOT).

- **Indexed view**: This allows creation of either a clustered or non-clustered index against a view. The view still consists of a query, which will be executed against underlying tables, every time the view is accessed.

  Some relational databases allow creation of what is called a materialized view. A materialized view does not contain a query, but is a copy of data. The data copy in the materialized view can be updated automatically, periodically, or even in real-time. Additionally, materialized views usually allow automated changes to queries, as they are submitted to a database, effectively rewriting the query (query rewrite) during the optimization process. This can help to improve performance if a materialized view contains a summary (much fewer records) of data in underlying tables.

  Materialized views are mostly applicable in data warehouses but can be utilized in OLTP databases under certain circumstances. As with views, materialized views, and index views—these objects will not help performance if over-used or used to cater to overlaying a new table design on top of a poorly designed data model.

  SQL Server does not allow explicit creation of, or even direct access to materialized views. In SQL Server, materialized views are all internalized within Analysis Services and cannot be used outside of the context of the analytical processing engine. In other words, you can't explicitly build a materialized view, you can't explicitly access a materialized view (using a query), and you can't force the optimizer to use or ignore query rewrite. In essence, materialized views are utilized in SQL Server 2005 but completely contained within Analysis Services.

- **Other factors**: This includes more architectural aspects, such as partitioning at the table and index level, parallel processing, clustering, replication, duplication, and mirrored (or standby) databases.

Most of these performance solutions are more physical rather than logical forms of denormalization, as they require creation of objects not part of the standard normalization process. These specialized objects do not reduce logical complexity, but can very well do the opposite by increasing complexity. This is simply because more database objects are being created. What specialized objects do accomplish is to place data sets into purpose built constructs. These constructs can be used to access data in the required order, without jeopardizing access speed to the underlying tables, which may be very busy doing other things like servicing a busy website. So, in addition to allowing access to data in required order, extra objects such as materialized views can help to reduce multi-user conflict issues.

2.4 Extreme denormalization in data warehouses

Data warehouses deal with such enormous quantities of data that they can sometimes require specialized data modeling techniques. Some data warehouses use specialized modeling technique called the dimensional data model.

The dimensional data model is sometimes also called the dimensional-fact model.

The dimensional data model contains tables built to form a star (a star schema) or a snowflake (a snowflake schema). Some data warehouses may use 3rd normal form schemas (just like an OLTP data model relational schema).

Many data warehouses contain hybrid schemas using a combination of two or even all three data model types including star, snowflake, and 3rd normal form schemas. In general, for large amounts of data, and the intensity of data warehouse analytical reporting, the most efficient data model for a data warehouse is a star schema. A snowflake is a slightly more granular structural form of a star schema, containing more tables, and is thus less efficient than a star schema. A 3rd normal form schema is a highly granular and broken down schema, requiring many tables in join queries—and consequently provides very poor performance.

So, let's briefly describe the data warehouse dimensional model for representing data. The dimensional model consists of dimensions and facts. A fact is some historical or archived record, about some type of activity. A dimension is literally a dimension of a fact, or something describing a fact.

OLTP transactional databases use a normalized relational data model, which requires a large number of small operations. Data warehouses on the other hand require small numbers of large transactions, for data loading and reporting. The requirements are thus completely different with respect to performance—a data warehouse does lots of joins, on lots of tables, and reads lots of data in each transaction. Performing data warehouse type reports on a normalized OLTP database would likely cause the customer facing OLTP applications to perform very poorly.

A data mart is a term often bandied around in data warehouse terminology. A data mart is simply a subsection of a data warehouse.

Let's take a quick look at an OLTP relational data model for a container shipping company. This company ships containers of luxury goods. The company owns a number of small container ships. They ship goods between a European port and a number of West African ports. Figure 2.13 shows this OLTP database relational data model.

The meanings of tables shown in Figure 2.13 are as follows:

- **CONSIGNEE**. The party receiving the container contents.
- **SENDER**. The party shipping the container contents.
- **VESSEL**. The ship on which the container was transported.
- **VOYAGE**. Ships making voyages can call at multiple ports. Individual containers or groups of containers are transported on all or some legs of the entire voyage.
- **SHIPMENT**. Contents of part of a container, or even one or more containers, are shipped from one port to another. For the sake of simplicity we assume a shipment as being an individual container shipped from one port to another.
- **CONTENTS**. The contents of a container or the Bill of Lading.

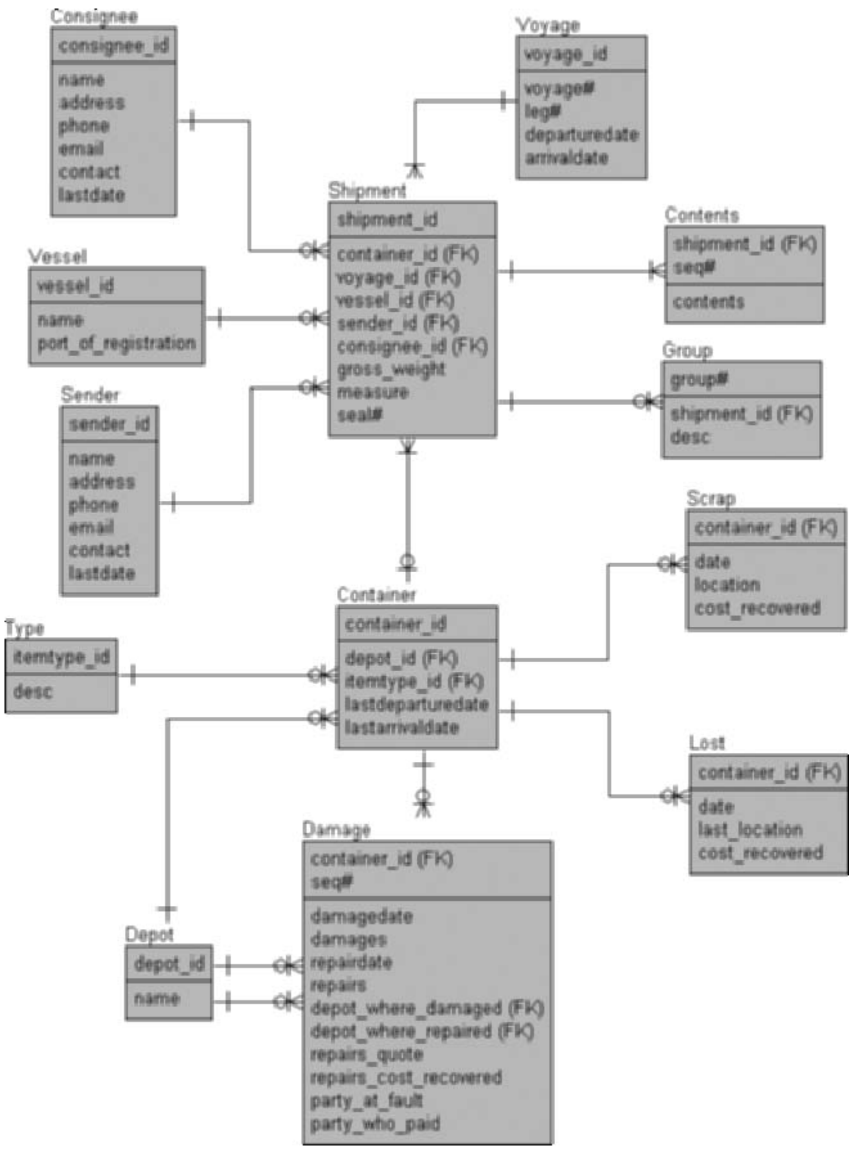


Figure 2.13
OLTP relational model for tracking container shipments

- **GROUP**. A group of containers are transported from one port to another as a single shipment.
- **CONTAINER**. An individual container.
- **TYPE**. A container can be refrigerated, open-topped, or a flatbed, amongst numerous other types.
- **DEPOT**. A container depot or container port.

- **DAMAGE**. Containers can sustain damage.
- **SCRAP**. Damaged containers can become irreparably damaged and have a scrap value.
- **LOST**. Containers can be stolen or sometimes even lost at sea. In fact loss of containers at sea happens often. Additionally these containers being sealed can float just below the water, sometimes doing really nasty things to smaller craft.

Now let's convert the relational model to a data warehouse dimensional-fact model.

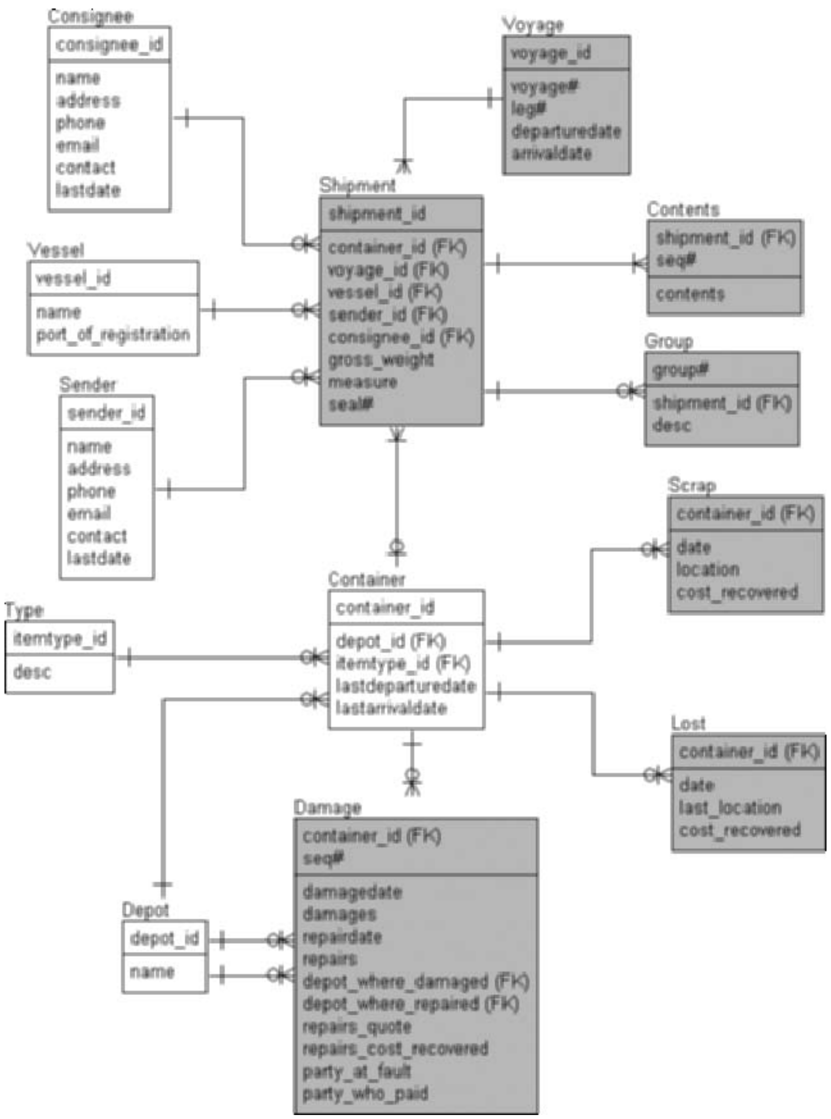
2.4.1 The dimensional data model

A table relationship model is inappropriate to the requirements of a data warehouse, even a denormalized one. Another modeling technique used for data warehouses is called dimensional modeling. In layman's terms a dimensional model consists of facts and dimensions. What does that mean? What is a fact and what is a dimension? A fact is a single iteration in a historical record. A dimension is something used to dig into, divide, and collate those facts into something useful. That isn't really layman's terms now is it? Let's try to explain this a little more easily by example.

Let's explain dimensional modeling in small steps. Figure 2.14 shows the same table relationship as that shown in Figure 2.13, but with a slight difference. Vaguely, facts are the equivalent of transactional tables and dimensions are the equivalent of static data. Therefore in Figure 2.14 the fact tables are colored gray and the dimensions tables are not. Note how the facts represent historical or archived data and dimensions represent smaller static data tables. It follows that dimension tables will generally be small and fact tables can become frighteningly huge. What does this tell us? Fact tables will always be appended to and dimension tables can be changed, preferably not as often as the fact tables are appended to. The result is many very small tables related to data in groups from very large tables.

The most desirable result when modeling for a data warehouse using dimensions and facts is called a star schema. Figure 2.15 and Figure 2.16 show slightly modified, pseudo-type star schema versions of the normalized table relationship diagrams in Figure 2.13 and Figure 2.14. In Figure 2.15 we can see that all dimensions would be contained within a single fact table, containing shipping history records of containers. Each record in the fact table would have foreign key values to all related dimension tables.

Figure 2.14 *Highlighting dimensions and facts for container shipments*



Every star or snowflake schema always has a fact table. A single data warehouse can consist of multiple fact tables and thus multiple star and or snowflake schemas.

Figure 2.15 simply contains another fact table or another subset of the data contained in the normalized table relationship structure in Figure 2.13 and Figure 2.14. It is quite conceivable that the two fact table sets in Figure

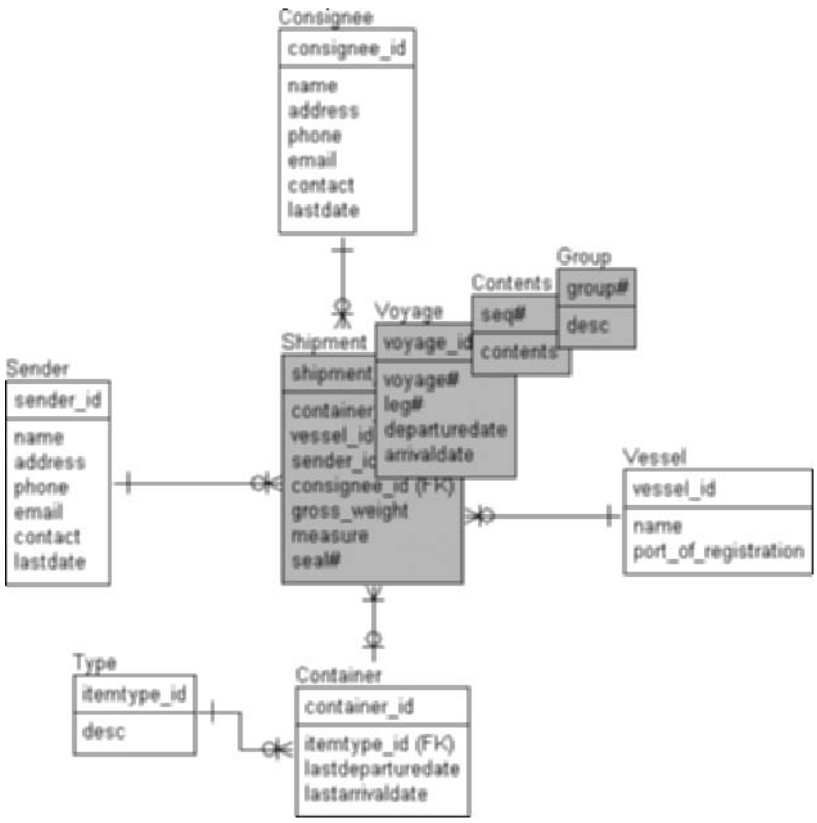


Figure 2.15 *Star schema for containers currently at sea*

2.14 and Figure 2.15 should be merged into a single table, separating used, damaged, scrapped, and lost containers by an appropriate type field.

There could be a small problem with the fact table shown in Figure 2.15. Damaged, scrapped, and lost containers could either be a fact table or part of the container dimension. This decision would depend on exactly how often containers are damaged, scrapped or lost. It is more than likely that this type of thing occurs frequently in relation to other dimensions, but not necessarily in relation to the high frequency of container shipments.

The star schemas shown in Figure 1.8 and Figure 1.9 show that there are two potential fact tables for the relational schema shown in Figure 2.13 and Figure 2.14.

2.4.1.1 What is a star schema?

A star schema contains one or at least very few, very large fact tables, plus a large number of small dimensional tables. As already stated, effectively fact

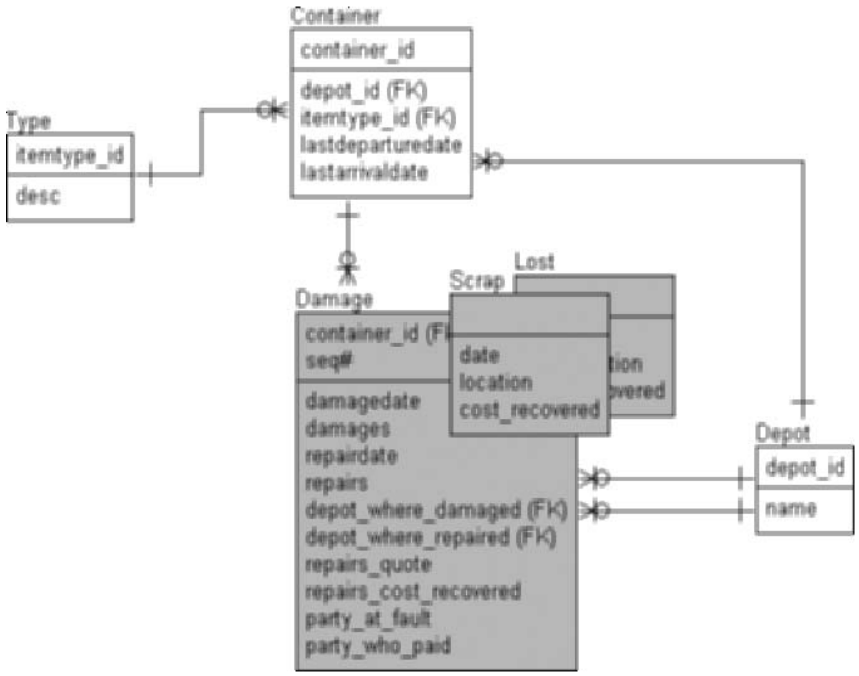


Figure 2.16
Damaged, scrapped and lost containers star schema

tables contain transactional histories and dimension tables contain static data describing the fact table archive entries. The objective for performance is to obtain joins on a single join level where one fact table is joined to multiple small dimension tables, or perhaps even a single dimension table. Figure 2.17 shows a snowflake schema for the used container portion of the original normalized structure in Figure 2.13 and Figure 2.14, assuming that damaged, scrapped, and lost containers represented in Figure 2.16 are rolled into the container dimension.

2.4.1.2 What is a snowflake schema?

A snowflake schema is a normalized star schema such that dimension tables are normalized.

Dimension objects can be used in Oracle Database to represent multiple layers between dimensions creating dimensional hierarchies. Oracle Database dimensional hierarchies can assist optimizer efficiency and materialized view query rewrite selection.

The schema shown in Figure 2.17 is actually a snowflake schema because the type dimension has been normalized out from the container dimension. Figure 2.18 shows a star schema representation of the snowflake

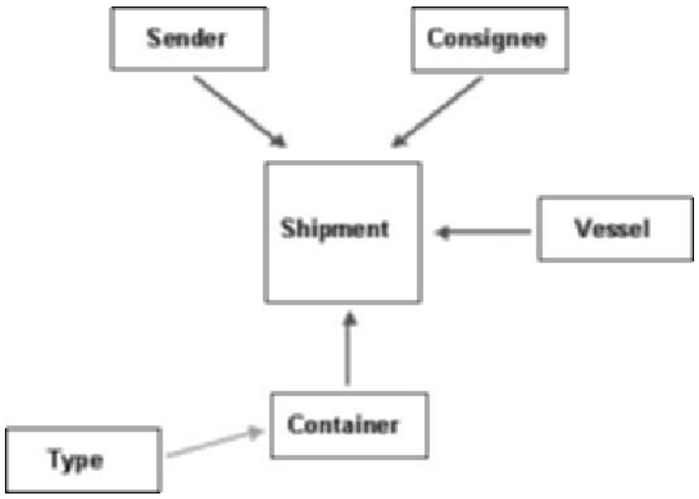


Figure 2.17
A snowflake schema

schema in Figure 2.17. Figure 2.18 has the type dimension included into the container dimension. There is no dimensional normalization in a star schema.

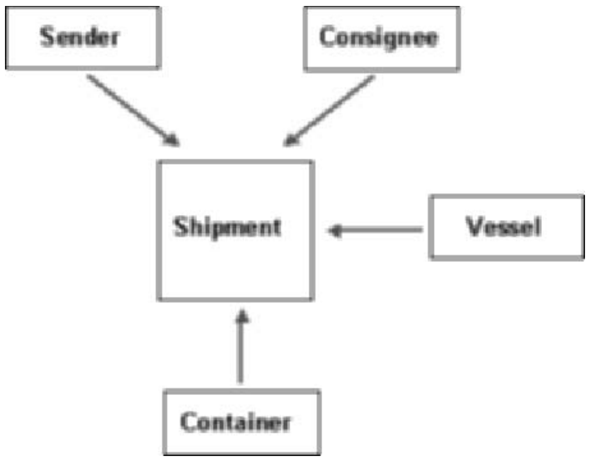


Figure 2.18
A star schema

Once again let's reinforce the point previously made: the schema representation shown in Figure 2.18 is a star schema of the snowflake schema shown in Figure 2.17. The reason why is because the type table has been rolled into or contained within the container table. Effectively the container and type tables have been denormalized such that type.name field values, the name of the type itself excluding the type.id field, are now included with the container table. Denormalization is effective for a data warehouse schema for a number of reasons:

- Non-technical users such as salespeople, forecasters, and sometimes even executives often access data warehouse tables. A star schema is very simple to understand without bombarding the user with all the complex tables and intra-table relationships of a relational data model and multiple dimensional hierarchies. Containers are meaningful to end-users. Types of containers may not be as meaningful in a users mind, perhaps being understood better as *refrigerated* or *flatbed*. If users do not know what something is it could potentially render the entire structure useless to some people.
- As you can see by the simplicity of the star schema shown in Figure 2.18 a star schema can easily be augmented by adding new dimensions as long as they fit in with the fact table.
- From a purely performance tuning perspective a star schema rolls subset dimensions into single tables from a multiple dimensional hierarchy of a snowflake schema. The number of joins in queries will be reduced. Therefore queries should execute faster.

A single data warehouse database can contain multiple fact tables and therefore multiple star schemas. Additionally individual dimension tables can point to multiple fact tables. Perhaps more importantly bear in mind that dimension tables will occupy a small fraction of the storage space that fact tables do. Fact tables in a data warehouse can have billions of rows whereas dimensions are in the range of tens, hundreds. or perhaps thousands. Any larger than thousands and those dimensions could possibly be facts.

As a final note, snowflake schemas help to organize dimensions a little better from a mathematical perspective by saving disk space. Disk space is cheap. Increasing dimension number increases joins in queries. The more joins in queries the worse a data warehouse will perform in general.

2.4.2 Data warehouse data model design basics

What is the objective of a data warehouse? When asking this question the primary consideration is for the end-users, the people who read reporting produced from a data warehouse, whether they build those reports or not.

So think in terms of end-users searching for patterns and trying to forecast trends from masses of historical information.

How does one go about designing a data warehouse data model?

- **Business Processes**. What are the business processes? The business processes will result in the source of facts about the business, the fact tables.
- **Granularity**. What is the level of granularity required? Fact tables are appended to at a specific level of granularity or grain. The finer the granularity the more of your historical data is stored. The lowest level of granularity excludes any summaries of historical data, even though specialized objects such as materialized views can create summaries at a later stage. When you do not know the precise requirements for future use of your data warehouse, to be on the safe side, it is best to store all levels of detail. If you miss a level of detail in any specific area and it is later requested you are likely to be up the creek without a paddle!
- **Identify Dimensions**. What are your dimensions? Dimensions generally consist of static data such as codes and descriptions, the kind of data that goes into front-end application pick lists. Make sure that the dimensional data conforms to fact data granularity levels.
- **Build Facts**. The last step is to build the fact tables containing all the non-static transactional information generated by applications on a daily basis.

When building a data model for a data warehouse database there are various factors to consider. Some of these design factors to consider in design can be vaguely divided up by dimension and fact tables. Let's begin with dimension tables.

2.4.2.1 Dimension tables

Data warehouse dimensions can be loosely classified into various groups. The more generic types are dimensions such as time, product, or location dimensions. Obviously your classifications may vary depending on the data warehouse.

A time dimension determines how long archived or historical data is to be retained. Figure 2.19 shows an example.

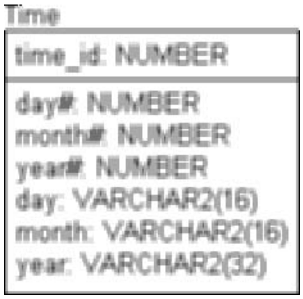


Figure 2.19
A time dimension table

A product dimension would be used to contain details or products a company produces or deals with in some way, as shown in Figure 2.20.

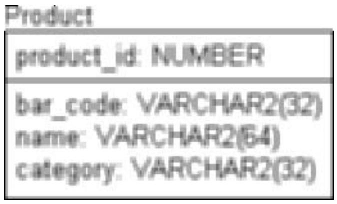


Figure 2.20
A product dimension table

Figure 2.21 shows what could be an original relational table set of product and category tables, with a many-to-one relationship between the product and category tables. The product dimensional table in Figure 2.21 has the name of the category rolled into the product table, minimizing on joins and ultimately providing for faster performing code in a data warehouse.

Transactional databases usually have some form of demographic information attached to transactions, either directly or through relations with other tables. The equivalent for a data warehouse is sometimes a location dimension such as that shown in Figure 2.22.

Figure 2.23 shows a transformation between a relational table location structure design, and that of a location dimension table on the right side of the diagram. Figure 2.23 serves to show the stark difference in complexity between a relational structure and its equivalent denormalized dimensional structure.

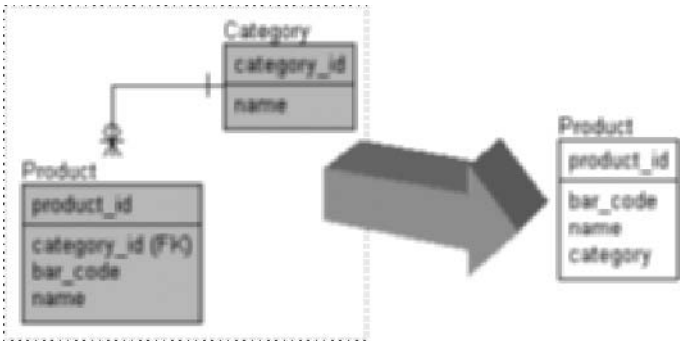


Figure 2.21
Relational to dimensional star schema transformation for Figure 2.20

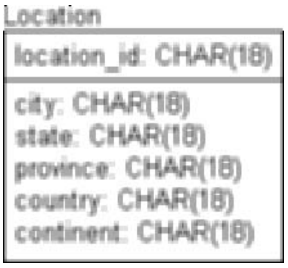


Figure 2.22
A location dimension table

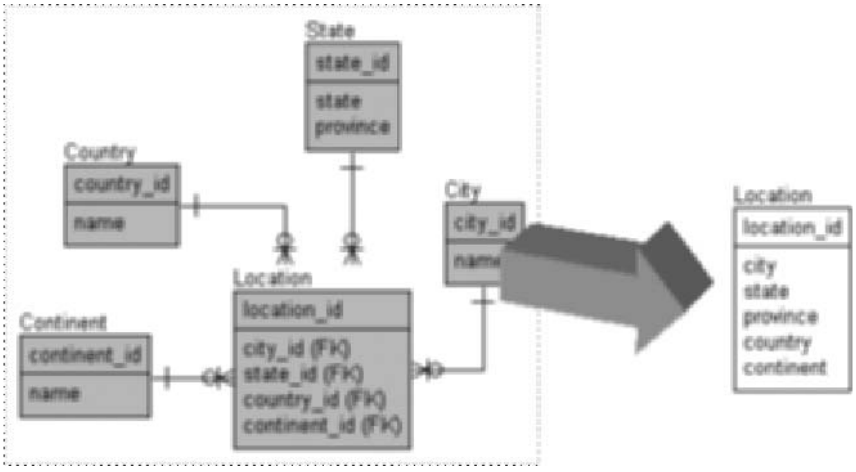


Figure 2.23
Relational to dimensional star schema transformation for Figure 2.22

Using generic dimensions containing information such as time, product, and location it is likely that dimensions in Figure 2.14 to Figure 2.18 could very well be absorbed or denormalized into the existing shipment fact table.

Commonly used data warehouse dimensions can also be tables such as customers, stock items, and suppliers. The star schema in Figure 2.18 shows more specific rather than generic dimensions in the form of senders, consignees, vessels, and containers.

Now let's take a look at factors affecting data warehouse data model design with respect to fact tables.

2.4.2.2 Fact tables

In general fact tables contain two field types, namely numeric fact values and foreign key reference attributes to dimension tables. Figure 2.24 shows a more detailed version of the snowflake schema shown in Figure 2.15. The shipment, group, contents, and voyage specific table fields are named as such simply to indicate their relational table origins. All of these attributes are measurements of one form or another. The remaining attributes for the shipment table shown in Figure 2.24 are all dimensional table foreign keys.

The two voyage departure and arrival dates could be removed to a time-based table and replaced with the appropriate foreign keys.

Data warehouse fact tables can be loosely classified into various areas. Once again classifications may vary depending on the data warehouse. Fact table types are as follows:

- Fact tables can contain detailed fact record entries.
- Fact tables can be summary tables, which are aggregations of details. Detail rows are not stored in the data warehouse, probably having been summarized in transportation from transactional databases.
- Fact tables can be simply added to or changed, but are preferably only partially changed, or better still static. The least intrusive method of adding new data is during low usage times and appended to the end of existing data.

Fact table attribute types are another significant point to note with respect to fact tables. Facts are often known as additive, semi-additive, or non-additive. For example, accounting amounts can be added together (additive) but averages cannot (non-additive). Semi-additive facts can be accumulated for some but not all facts.

In Figure 2.24 the shipment.shipment_gross_weight is additive across all dimensions. In other words there is a gross weight for a sender (the per-

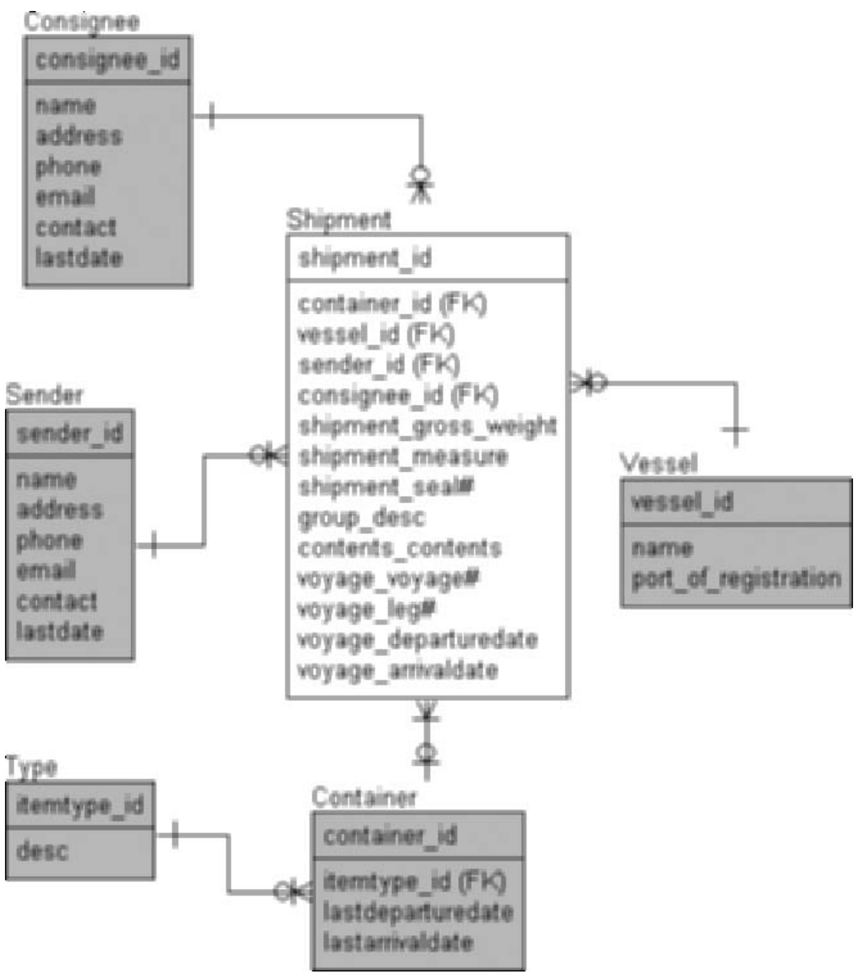


Figure 2.24 *A detailed fact table*

son or company shipping the container), the consignee (the person or company receiving the container), a vessel, and a container. The shipment.contents_contents attribute is semi-additive because contents are only specific to each container and thus meaningless to sender, consignee, or vessel. Multiple senders and consignees can have their individual shipments grouped in a single container or even spread across multiple containers. The shipment.voyage# and shipment.voyage_leg# attributes are nonadditive since these values simply do not make sense to be accumulated. However, an average number of legs per voyage, per vessel might make some sense as an accumulation.

It is always preferable that fact tables are only added to, and also best appended to. Less desirable situations are fact tables with updatable cumulative fields such as summaries or even fact tables requiring deletions. Even worse are fact tables containing directly updatable numerical values. For example, if a table contains a history of stock movements and all prices have increased, it is possible that a large portion of those stock movement entries must be updated. It might be best in a situation such as this to maintain stock item prices outside of the stock movements table, perhaps as part of a dimension.

Granularity, Granularity, and Granularity! The most significant factor with respect to fact tables is granularity—how much data to keep, to what level of detail. Do you store every transaction or do you summarize transactions and only store totals for each day, month, per client, and so on?

Granularity is deciding how much detail the data warehouse will need in the future. This is all about requirements. Firstly, do the end-users know how much detail they will need? Do you understand and trust that they know what they need? Will they change their minds? Probably. If there is any doubt, and if possible, keep everything! Now there's a simple of rule of thumb. Disk space is cheap unless your data warehouse is truly humungous. If that is the case you have plenty of other problems to deal with as well, You also have a good reason why particular details might be missing say one year down the line when some really irritated executive comes to ask you why the data he wants isn't there. Too much data is your answer.

So the most important design issue with fact tables is the level of granularity. Simply put this means, does one save all the data or summarize it? Storing all data can lead to very large fact tables and thus very large databases. However, after data has been deleted from your transactional data sources it might be costly to discover that all the archived dimensional-fact combinations are required for reports at a later date. From a planning perspective it might be best to begin by retaining all facts down to the smallest detail if at all possible. Data warehouses are expected to be large and disk space is cheap.

The time factor and how long should data be retained? How long do you keep data in the data warehouse for? Some data warehouses retain data in perpetuity and others discard data over a few years old. Expired data removed from a data warehouse can always be copied to backup copies. However, remember that if a data warehouse is extremely large, removing older data may cause serious performance problems for appending of new data, and most especially for end users. Many modern databases operate on a global twenty-four hour time scale; there simply is no room for down time or even *slow time* for that matter.

2.4.2.3 Other factors to consider during design

On the topic of surrogate keys: when designing tables for data warehouses use surrogate keys or unique sequential identifiers for table primary keys. The reason for this is possible multiple data sources from different databases and perhaps even different database vendor software. Additionally data can originate from flat text files and perhaps from outside purchased data sets. Keys in different data sources could be named differently containing the same values or named the same with different values. The data warehouse needs its own unique key values specific to itself and each of its tables, as a distinct data table set.

On the subject of duplication of surrogate keys and associated names: when creating composite field keys for child tables, parent key fields could be copied into child tables; not only copy the key values into the child tables but also the unique naming values they represent, as shown in Figure 2.25. This will allow access to all details of a multiple hierarchical dimension in all forms.

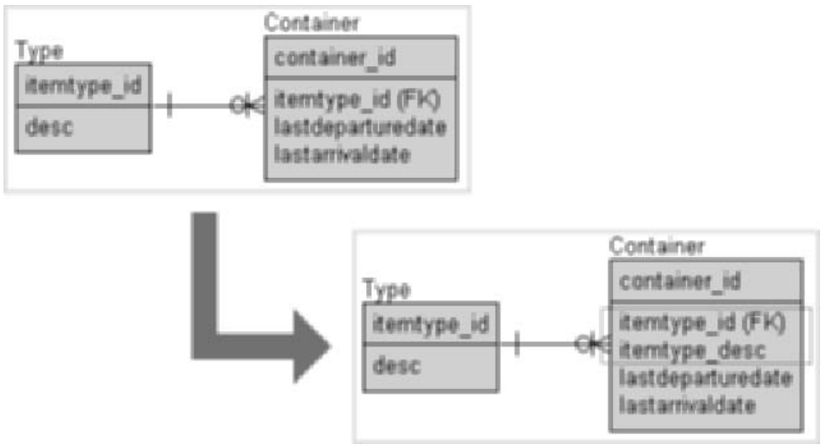


Figure 2.25 *Duplicate all composite key fields*

This chapter has shown you the essentials for logical database design. In this case, relational database design. This chapter is deliberately non-vendor database specific. This is because logical relational database design is not fussy about the database vendor, only that you are using a relational database. The relational design features are still more or less identical. Except that perhaps there are a few small quirky differences here and there. The next chapter will cover physical database design, including information about underlying operating system file structure.

2.5 Endnotes

1. Beginning Database Design (Gavin Powell, Wiley, Late 2005, ISBN: 0764574906).
2. The Object Database Standard (Cattell, Morgan Kaufmann, Jun 1997, ISBN: 1558604634) and The Object Database Handbook (Barry, Wiley, pre-2000, ISBN: 0471147184).

3

Physical Database Design

Physical design is the underlying file structure, within the operating system, out of which a database is built as a whole. Physical database design takes into account the performance speed of hardware in use. And thus factors such as I/O, memory, and CPU processing speed are important. Just as important though is the matching of hardware parameters with requirements. For example, a data warehouse is generally very I/O intensive. So, a data warehouse will likely perform best if you spent relatively more time and money of the disk subsystem, than on the CPU and memory.

3.1 Introducing physical database design

Once the database logical design has been satisfactorily completed on paper, it can be turned into a database physical design.

Note: Logical database design, as pertaining to performance, was covered in Chapter 2.

In the physical design process the database designer will be considering such issues as the placement of data and the choice of indexes and, as such, the resulting physical design will be crucial to good database performance. The following important points should be made here:

- A bad logical design means that a good physical design cannot be implemented. Good logical design is crucial to good database performance. A bad logical design will result in a physical design that attempts to cover up weaknesses in a logical design. A bad logical design is hard to change, and once the system is implemented it will be almost impossible to do so.

- The physical design process is a key phase in the overall design process. It is too often ignored until the last minute in the vain hope that performance will be satisfactory. Without a good physical design, performance is rarely satisfactory. Throwing hardware at the problem is rarely completely effective. There is no substitute for a good physical design. The time and effort spent in the physical design process will be rewarded with an efficient and well-tuned database, not to mention happy users!

Before embarking on the physical design of the database, it is worth stepping back and considering a number of points, as follows:

- What kind of system are we trying to design? Is it a fast online transaction processing (OLTP) system comprised of perhaps hundreds of users with a throughput of hundreds of transactions per second (TPS) and an average transaction response time that must not exceed two seconds? Is it a multi-gigabyte or even a multi-terabyte data warehouse, which must support few online users but must be able to process very complex ad hoc queries in a reasonable time. Is your database perhaps a combination of both OLTP and data warehouse databases?

Note: The type of system will strongly influence the physical database design decisions that must be made. If the system is to support OLTP and complex decision support, then maybe more than one database should be considered—one for the operational OLTP system and one, fed by extracts from the operational OLTP system, to support complex decision support (the data warehouse).

Even some of the very largest of modern databases are combinations of both OLTP and data warehouses. However, achieving good performance with combining OLTP and data warehouse structure creates conflicting requirements. An OLTP environment needs to get at small pieces of data, rapidly. A data warehouse needs to find lots of data all at once, and it is not expected to happen in milliseconds. In other words, the two requirements are completely contradictory. It is possible to join the two architectures into a single database, if your database is either very small, or you can afford incredibly expensive hardware.

- What are our hardware and budget constraints? The most efficient physical database design will still have a maximum performance capability on any given hardware platform. It is no use spending weeks trying to squeeze the last few CPU cycles out of a CPU bound database when, for a small outlay, another processor can be purchased. Similarly, there is little point purchasing another CPU for a system that is disk I/O bound.
- Has the database design been approached from a textbook normalization standpoint? Normalizing the database design is the correct approach and has many benefits, but there may be areas where some denormalization might be a good idea. This might upset a few purists, but if a very short response time is needed for a specific query it might be the best approach. This is not an excuse for not creating a normalized design. A normalized design should be the starting point for any effort made at denormalization.
- How important is data consistency? For example, is it important that if a query rereads a piece of data within a transaction it is guaranteed that it will not have changed? Data consistency and performance are enemies of one another, and, therefore, if consistency requirements can be relaxed, performance may be improved as a result.

How does a database designer move from the logical design phase to a good physical database design? There is no single correct method. However, certain information should be captured and used as input to the physical design process. Such information includes data volumes, data growth, and transaction profiles.

3.2 Data volume analysis

It is very important to capture information on current data volumes and expected data volumes. Without this information it is not even possible to estimate the number and size of the disk drives that will be required by the database. Recording the information is often a case of using a simple spreadsheet, as shown in Table 3.1.

The activity shown in Table 3.1 may appear to be a trivial operation, but it is surprising how few database designers do it. It is also interesting to find the different views from business users on what the figures should be! Another column that could be added might represent how volatile the data is in a particular table. The percentage annual growth of a table might be

Table 3.1 *Capturing simple data volume information*

Table Name	# of Rows	Row Size	Space Needed	% Annual Growth	Space Needed in 12 Months
Accounts	25,000	100	2,500,000	10	2,750,000
Branches	100	200	20,000	5	21,000
Customers	10,000	200	2,000,000	20	2,400,000
Transactions	400,000	50	20,000,000	25	25,000,000

zero, but this may be because a large amount of data is continually being removed as well as being added.

Simple addition of these figures in Table 3.1 gives the data size requirements, but this is only part of the calculation. The database designer must take into account the space required by indexes, the transaction log, and the backup devices. An experienced database designer would not ask for the disk space that came out of the sum in Table 3.1. They would, of course, add on a percentage for safety. Users typically do not phone you to complain that you oversized the database by 20 percent. They will certainly phone you to complain that the system just stopped because the database was full!

So how are the sizes of indexes calculated? The *Creating and Maintaining Databases* online book gives sample calculations to assist in the sizing of tables, as well as clustered and non-clustered indexes with both fixed, and variable-length columns.

Note: The *Creating and Maintaining Databases* online book can be found in Microsoft documentation. Search http://www.microsoft.com for *Creating and Maintaining Databases*.

It is highly recommended that these calculations are performed, and it is worth using a spreadsheet such as Microsoft Excel to perform the calculations in order to save time and effort. Watch the newsgroups for stored procedures in circulation that do these calculations. When sizing a table, a general rule of thumb is to double the size of the user data to estimate the size of the database. Crude though this appears, by the time indexes and some space for expansion is added—double the size is not far off!

What about the size of the transaction log? This is difficult to size because it depends on write activity to the database, frequency of transaction backups, and transaction profiles. Microsoft suggests that about 10 percent to 25 percent of the database size should be chosen. This is not a bad start, but once the system testing phase of development has started, then the database designer can start monitoring the space use in the transaction log with dbcc sqlperf (logspace).The transaction log space is a critical resource and running out of it should be avoided.

Unfortunately, many factors contribute to transaction log growth. Factors include the rate per second of transactions that change database data, and the amount of data these transactions change. Remember that in an operational system, if a transaction log backup fails for some reason, then the transaction log will continue to fill until the next successful transaction log backup. It may be desirable to have a transaction log large enough so that it can accommodate the failure of one transaction log backup. Replication failures will impact the effectiveness of transaction log backups, and, of course, there is always the user who runs a job that updates a million-row table without warning you—or by mistake.

For all these reasons, do not be tight with transaction log space. Disk space is cheap. So, a transaction log can be created with a large amount of contingency space.

Finally, do not forget that as a database designer/administrator, you will need lots of disk space to hold at least one copy of the production database for performance tuning testing. Not having a copy of the production database can really hinder you. And if you are doing performance testing on a production database—then don't.

So, we now have documented information on data volumes and growth. This in itself will determine a minimum disk configuration. However, it is only a minimum because transaction analysis may determine that the minimum disk configuration will not provide enough disk I/O bandwidth.

If data volume analysis is concerned with the amount of data in the database and the space it needs, then transaction analysis is concerned with the way in which data is manipulated—and at what frequency.

3.3 Transaction analysis

Data in the database may be manipulated by code, such as Visual Basic, or a tool like Microsoft Access, or even a third-party product accessing SQL Server. Whichever way the data is accessed, it will presumably be as a result of a business transaction of some kind. Transaction analysis is about capturing information on these business transactions and investigating how they access data in the database, and in which mode. Table 3.2 shows some attributes of a business transaction it might be useful to record.

Table 3.2 *Capturing transaction attributes*

Attribute	**Explanation**
Name	A name assigned to the transaction
Average frequency	Average number of times executed per hour
Peak frequency	Peak number of times executed per hour
Priority	A relative priority assigned to each transaction
Mode	Whether the transaction only reads the database or writes to it also
Tables accessed	Tables accessed by the transaction and in which mode
Table keys	Keys used to access the table

Clearly, by its very nature, it is not possible to capture the information shown in Table 3.2 for ad hoc transactions. Nor is it practical to capture this information for every business transaction in anything other than a very simple system. However, this information should be captured for at least the most important business transactions. By most important we mean those transactions that must provide the fastest response times and/or are frequently executed. A business transaction that runs every three months and can be run over a weekend is unlikely to appear on the list of most important transactions!

It is important to prioritize transactions because it is virtually impossible to be able to optimize every transaction in the system. Indexes that will speed up queries will almost certainly slow down insert operations.

An example of the attributes captured for a transaction are shown in Table 3.3.

There are various ways to document the transaction analysis process. Some modeling tools will automate part of this documentation. The secret is to document the important transactions and their attributes. This allows the database designer to decide which indexes should be defined for which tables.

Table 3.3 *Example transaction attributes*

Attribute	Value
Name	Order Creation
Average frequency	10,000 per hour
Peak frequency	15,000 per hour
Priority	1 (high)
Mode	Write
Tables accessed	Orders (w), Order Items (w), Customers (r), Parts (r)
Table keys	Orders (order_number), Order Items (order_number), Customers (cust_number), Parts (parts_number)

Again, it is often a case of using simple spreadsheets, as shown in Table 3.4.

Table 3.4 *Capturing simple transaction analysis information*

Transactions/Tables	Orders	Order_items	Parts	Customers
Customer inquiry				R
Order inquiry	R	R		
Order entry	I	I	R	R
Customer inquiry				cust_number
Order inquiry	order_number	order_number		
Order entry	order_number	order_number	Parts_number	cust_number

The first spreadsheet maps the transactions to the mode in which they access tables. Modes are I for insert, R for read, U for update, and D for delete. The second spreadsheet maps the transactions to the key with which they access tables. Again, there is nothing complex about this but it really pays to go through these steps. Depending on how the system has been implemented, a business transaction may be modeled as a number of stored procedures. Also, one may wish to use these instead of transaction names if so desired.

It is also important when considering the key business transactions not to forget triggers. A trigger accesses tables in various modes, just as application code does.

Data integrity enforcement using declarative referential integrity should also be included. Foreign key constraints will access other tables in the database, and there is nothing magical about them. If an appropriate index is not present, they will scan the whole table like any other query.

Note: This is because foreign keys do not have indexes automatically created for them by SQL Server, as in the case of primary keys. A primary key must have a unique index otherwise every time new record is added to a table, SQL Server will scan the entire table to ensure that the new primary key is unique. A primary key must be unique. Foreign keys are not required to be unique, so, a unique key is not required. In fact, a foreign is not a requirement, and can be null under certain circumstances. If a foreign key index is required then one must be created manually.

Once a transaction analysis has been performed, the database designer should have a good understanding of the tables that are accessed frequently, in which mode, and with which key. From this information once a beginning can be made at deriving the following:

- Which tables are accessed the most and therefore, experience the most disk I/O?
- Which tables are written to frequently by many transactions and, therefore, might experience the most locking contention?
- For a given table, which columns are used to access the required rows? In other words, which common column combinations form the search arguments in the queries?

In short, where are the hot spots in the database?

The database designer, armed with this information, should now be able to make informed decisions. Those decisions will cover estimated disk I/O rates to tables, the type of indexes required on those tables, and the columns that should be used in indexes.

Most relational databases make it very easy to prototype where a prototype allows you to create something, which can be examined visually for correctness, and easily changed. SQL Server is as any other relational database at allowing for prototyping. So, there is no excuse for not testing the physical design you are considering. Load data into your tables, add your indexes, and put your database under stress using some appropriately representative Transact-SQL. See how many transactions a second you can perform on a given server. Or, to look at it another way, how much disk I/O does a named transaction generate? Which resource—CPU or disk—do you run out of first?

Start stress testing with simple experiments. Jumping in at the deep end with many users, and testing complex functionality, is likely to only complicate and confuse the issue. Begin with simple transactions issued by a single user, and then try some more complex transactions.

Do not forget multiuser testing! Lock contention cannot be tested unless some kind of multiuser testing is performed. In its simplest form, this might involve persuading a number of potential users to use the test system concurrently. Perhaps have them follow a set of scripts while performance statistics are monitored. In its more sophisticated form, this might involve the use of a multiuser testing product, which can simulate many users, while running automated scripts.

Transaction analysis and performance testing can be approached in a much more sophisticated way than has been described above. The important point is that it should be done—the level of sophistication being determined by the available resource, be it time or money.

Again, note that physical design and performance testing are ongoing activities. Systems are usually in a constant state of flux. This is because business requirements are usually in a constant state of flux. Therefore, performance should be regularly monitored and, if necessary, the database tuned.

3.4 Hardware environment considerations

The previous section described preproduction performance testing. This should have given the database designer a feel for the hardware requirements of the production system. Obviously, there is a hardware budget for any project, but it is clearly critical to have sufficient hardware to support the workload of the system. It is also critical to have the correct balance and correct type of hardware.

For example, there is no point in spending a small fortune on CPU power if only a small amount of money is spent on the disk subsystem. Similarly, there is no point in spending a small fortune on the disk subsystem if only a small amount of money is spent on memory. Would the application benefit from a multiprocessor configuration or a single powerful processor?

If the application's main component is a single report that runs through the night but must be finished before 9:00 A.M., then a single powerful processor might be the best choice. On the other hand, if the application consists of a large number of users using an OLTP system, a more cost-effective solution would probably be a multiprocessor configuration.

Take a step back and look at the application and its hardware as a whole. Make sure the system resource is not unbalanced and do not forget the network!

Once we have performed our data volume and transaction analysis we can start to consider our physical design. We will need to decide what transactions need to be supported by indexes and what type of index we should use. Chapter 5 discusses indexes in detail, but before we look at indexes we need a more general view of the storage structures used in SQL Server, and these are covered in the next chapter.

4

SQL Server Storage Structures

A developer of application code is probably quite content to consider an SQL Server as being a collection of databases containing tables, indexes, triggers, stored procedures, views, and so on. As a database designer and a person who will be responsible for the performance of those databases, it is useful to be able to look a little deeper at the storage structures in SQL Server. A lot of the internals of SQL Server are hidden and undocumented. However, one can still learn a fair amount about the way the product works. This chapter investigates the storage structures that SQL Server uses and the methods available to view them.

4.1 Databases and files

A database contains all the tables, views, indexes, triggers, stored procedures, and user data that make up an application. An SQL Server will typically host many databases.

The term SQL Server refers to a computer containing an installation of the SQL Server software. An SQL Server database is a single database, created within a SQL Server installation.

Usually individual databases are backed up, restored, and checked for integrity—database by database. So, a database can also be thought of as a unit of administration. We will need to spend some time here looking at how databases are structured and managed because a database is the container for our objects. We will then drill down into the database files and investigate database pages and other structures.

A database resides in one or more operating system files, which may reside on FAT, FAT32, or NTFS partitions; depending on the operating system in use. These operating system files are known, in SQL Server terminology, as database files. These database files may be used to hold user and system tables (data files), or to track changes made to these tables (transaction log files). An SQL Server 2005 database can contain a ridiculous number of files and in the multiple terabyte (TB) range in physical size. Even a single data file, by itself, can be many TB in size; this includes the transaction log as well. Of course, most sites will never get anywhere remotely close to these numbers, but it is nice to know that there is plenty of headroom!

The files used by an SQL Server database belong exclusively to a particular database. In other words, a file cannot be shared by more than one database, even if those databases share the same SQL Server installation. Clustering is the exception to this rule.

Clustering allows for multiple CPUs to share the same set of underlying data files. Clustering is an advanced architectural structure and will be covered briefly, later on in this book.

Also, a file cannot be used to hold both data and transaction log information. This means that a database must consist of a minimum of two files. This is a much cleaner model than used in previous versions (prior to SQL Server 7.0).

There are three file types associated with an SQL Server database:

- The primary data file is the starting point of the database and contains the pointers to the other files in the database. All databases have a single primary data file. The recommended file extension for a primary data file is an .mdf extension.
- Secondary data files hold data that does not fit into the primary data file. Some databases may not have any secondary data files. Other databases may have multiple secondary data files. The recommended file extension for secondary data files is an .ndf extension.
- Log files hold all of the log information used to recover the database. There is at least one log file for each database. The recommended file extension for log files is an .ldf extension.

The primary data file will hold the system tables and may hold user tables. For most users, placing all their database tables in this file, and placing the file on a suitable RAID configuration will be sufficient. For some users, their user tables may be too large to place in a single file, because this would mean that the file would be too large to place on one of the storage devices. In this case, multiple data files—primary and multiple secondary files—may be used. User tables would then be created and populated. SQL Server would allocate space from each file to each table, such that tables were effectively spread across the files and physical storage devices.

Figure 4.1 shows a simple database topology. It uses a single file to hold the system tables and user tables, plus a single file for the transaction log. The files reside on separate physical storage devices, which may be single disks or RAID configurations. RAID configurations are discussed in Chapter 10.

Figure 4.1
A simple database topology

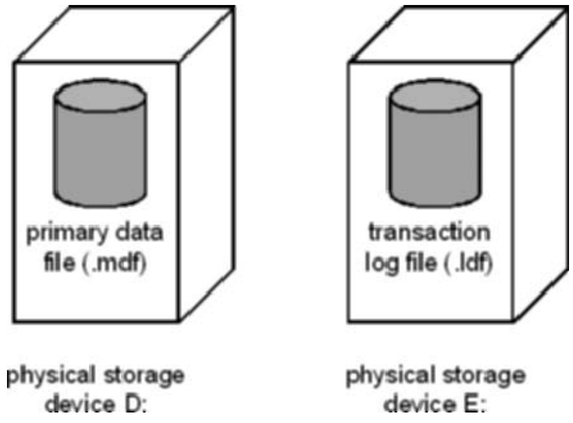

primary data file (.mdf)
transaction log file (.ldf)
physical storage device D:
physical storage device E:

Figure 4.2 shows a more complex database topology, using multiple files to hold the system tables and user tables, as well as multiple files for the transaction log. The files reside on separate physical storage devices, which may be single disks or RAID configurations.

For those users with even greater database performance and size requirements, filegroups may be used. The role of a filegroup is to gather data files together into collections of files into which database tables, indexes, and text/image data can be explicitly placed. This gives the database administrator a lot of flexibility and control over the placement of these database objects. For example, if two database tables are very heavily accessed, then they can be separated into two filegroups. Those two filegroups would consist of two sets of data files, residing on two sets of physical storage devices. The tables could also be separated from their non-clustered indexes in a similar fashion. Non-clustered indexes are described in Chapter 5. From an administration perspective, individual filegroups can be backed up, allowing for a large database to be backed up in parts.

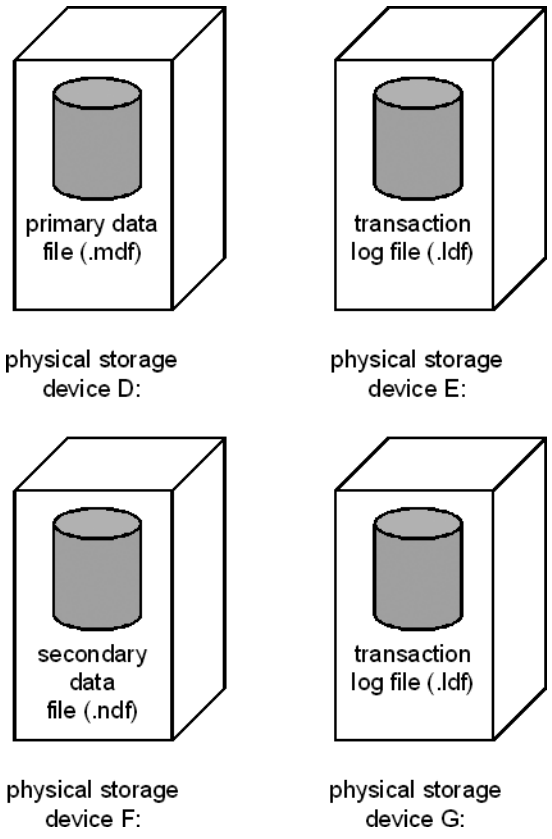

primary data file (.mdf)
physical storage device D:
transaction log file (.ldf)
physical storage device E:
secondary data file (.ndf)
physical storage device F:
transaction log file (.ldf)
physical storage device G:

Figure 4.2
A more complex database topology

Some rules govern the use of filegroups. Transaction logs are never members of filegroups—only data files are. Also, data files can only be a member of one filegroup.

For most users, though, the use of filegroups and multiple data and transaction log files will not be necessary to support their performance and administration requirements. They will use one data file and one transaction log file. Though they will not use user-defined filegroups, even in this simple case the database will contain a filegroup known as the primary filegroup. This will contain the system tables and user tables. It will also be the default filegroup. The default filegroup is the filegroup into which tables, indexes, and text or image data is placed when no filegroup is specified as part of their definition. Any filegroup can be made the default filegroup, and there is a school of thought that advocates always create a single user-defined filegroup, and making this the default filegroup when the database is first created. This ensures that the system tables alone reside in the primary filegroup and all user data resides in the user-defined filegroup in a separate, secondary data file.

4.2 Creating databases

The easiest way to create a database in SQL Server 2005 is to use the SQL Server Management Studio, which incidentally can also be used to generate a script for creating a database at a later date. Obviously scripts can be manually changed as well for fine tuning. There is also the Transact-SQL CREATE DATABASE statement. What used to be the Create Database Wizard prior to SQL Server 2005 is now part and parcel of the SQL Server Management Studio

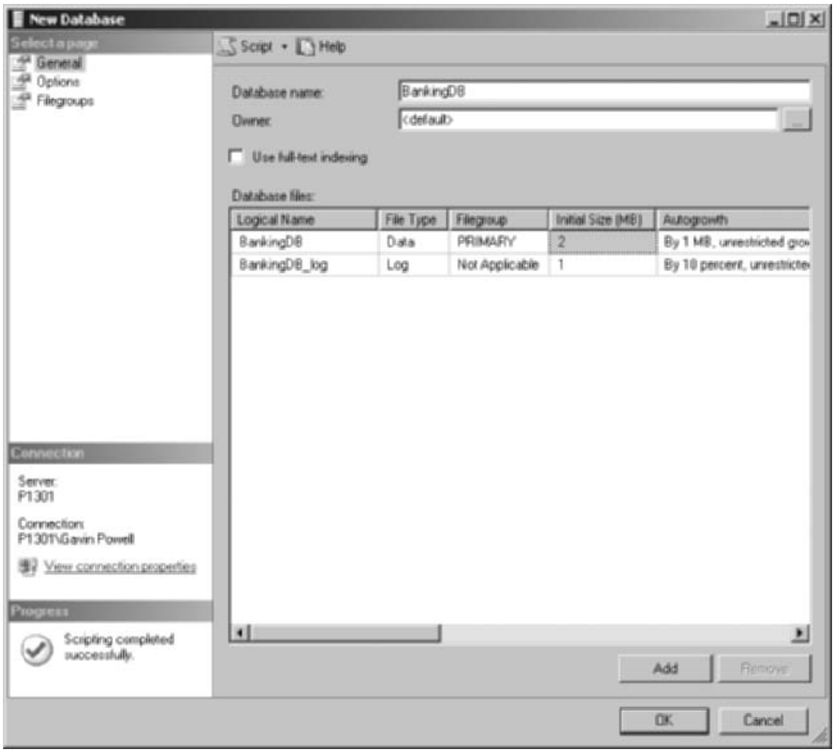
Let's use the SQL Server Management Studio to create a sample database:

1. Start up the SQL Server Management Studio.
2. Right-click Databases in the Object Explorer, then click New Database.
3. Enter the name of the database.
4. Leave everything else set to the default settings.
5. Click OK.

Depending on how large the database will be, this may take a considerable length of time. If your database is very large, using a Transact-SQL script running in the background may be a better bet. The SQL Server Management Studio New Database dialog box is shown in Figure 4.3.

As can be seen in Figure 4.3, various properties can be set for each data and transaction log file. The logical name of the file is the name by which it is referred to within SQL Server. For example, by various system-stored procedures, such as sp_helpfile. The file type is general data or a log file. A filegroup may also be entered for data files other than the primary at this point, in which case a secondary data file will be placed in that filegroup. Other attributes of the file relate to size and growth, which will be discussed shortly.

An example of creating a database using the Transact-SQL CREATE DATABASE statement is as follows:



Database name: BankingDB
Owner: <default>
Use full-text indexing

Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth
BankingDB	Data	PRIMARY	2	By 1 MB, unrestricted grow
BankingDB_log	Log	Not Applicable	1	By 10 percent, unrestricte

Server:
P1301

Connection:
P1301\Gavin Powell

View connection properties

Scripting completed successfully.

Figure 4.3
Creating databases

```
CREATE DATABASE BankingDB ON PRIMARY
( NAME = BankingData, FILENAME = 'd:\data\BankingData.mdf',
   SIZE = 2MB, MAXSIZE = 100MB, FILEGROWTH = 1MB )
LOG ON ( NAME = 'BankingLog',
   FILENAME = 'e:\data\BankingLog.ldf',
   SIZE = 1MB, MAXSIZE = 100MB, FILEGROWTH = 10% )
```

The script above, and many following, uses very small default values set in SQL Server Management Studio. Small sizes for files and growth are generally unrealistic for a busy database.

As with SQL Server Management Studio, a name is specified for the file—this time with the NAME option—and a physical location is specified with the FILENAME option. The ON keyword introduces a list containing one or more data file definitions, and the LOG ON keyword introduces a list containing one or more transaction log file definitions.

The PRIMARY keyword identifies the list of files following it as files that belong to the primary filegroup. The first file definition in the primary filegroup becomes the primary file, which is the file containing the database system tables. The PRIMARY keyword can be omitted, in which case the first file specified in the CREATE DATABASE statement is the primary file.

Regardless of the mechanism by which a database is created, size and growth information may be specified. The Initial size (MB) in the SQL Server Enterprise Manager and the SIZE keyword in the CREATE DATABASE statement specify the initial size of the file. In Transact-SQL, the units are, by default, megabytes, although this can be specified explicitly by using the suffix MB. If desired, the file size can be specified in kilobytes using the KB suffix, gigabytes using the GB suffix, and terabytes using the TB suffix.

When a data file or transaction log file fills it can automatically grow. In the SQL Server Management Studio, a file is allowed to automatically grow by default. In Transact-SQL, the file, by default, will be allowed to grow unless the FILEGROWTH keyword is set to 0. When a file grows, the size of the growth increment is controlled by the file growth property in the SQL Server Enterprise Manager and the FILEGROWTH keyword in Transact-SQL. In Transact-SQL, the growth increment can be specified as a fixed value, such as 10 megabytes, or as a percentage. This is the percentage of the size of the file at the time the increment takes place. Therefore, the size increment will increase over time. In Transact-SQL, the FILEGROWTH value can be specified using the suffix MB, KB, GB, TB, or %, with MB being the default. If the FILEGROWTH keyword is not specified in Transact-SQL, the default is 10 percent.

The file may be allowed to grow until it takes up all the available space in the physical storage device on which it resides, at which point an error will be returned when it tries to grow again. Alternatively, a limit can be set using the Restrict file growth (MB) text box in the SQL Server Enterprise Manager or the MAXSIZE keyword in Transact-SQL. The MAXSIZE value can be specified using the suffix MB, which is the default, KB, GB, or TB. The keyword UNLIMITED can also be specified—this is the default.

Every time a file extends, the applications using the database during the file extension operation may experience performance degradation. Also, extending a file multiple times may result in fragmented disk space, most especially where file growth additions are not all of the same size. It is advisable, therefore, to try to create the file with an initial size estimated to be close to the size that will ultimately be required by the file.

The following example shows a CREATE DATABASE statement, which will create a database consisting of multiple data and transaction log files:

```
CREATE DATABASE BankingDB ON PRIMARY
( NAME = BankingData1, FILENAME = 'd:\data\BankingData1.mdf',
 SIZE = 2MB, MAXSIZE = 100MB, FILEGROWTH = 1MB),
( NAME = BankingData2, FILENAME = 'e:\data\BankingData2.ndf',
 SIZE = 2MB, MAXSIZE = 100MB, FILEGROWTH = 1MB)
LOG ON
( NAME = BankingLog1, FILENAME = 'f:\data\BankingLog1.ldf',
 SIZE = 1MB, MAXSIZE = 100MB, FILEGROWTH = 10%),
( NAME = BankingLog2, FILENAME = 'g:\data\BankingLog2.ldf',
  SIZE = 1MB, MAXSIZE = 100MB, FILEGROWTH = 10%)
```

The following example re-creates the multiple file BankingDB database created in the previous example, but this time a user-defined filegroup, Filegroup1, is created. The file named BankingData2 is placed into Filegroup1 because it follows the file-group definition. This means that tables, indexes, and text or image data can be explicitly placed in this filegroup, if so required. If no filegroup is specified on the object definition, the object will be created in the DEFAULT filegroup, which is the primary filegroup (unless deliberately changed):

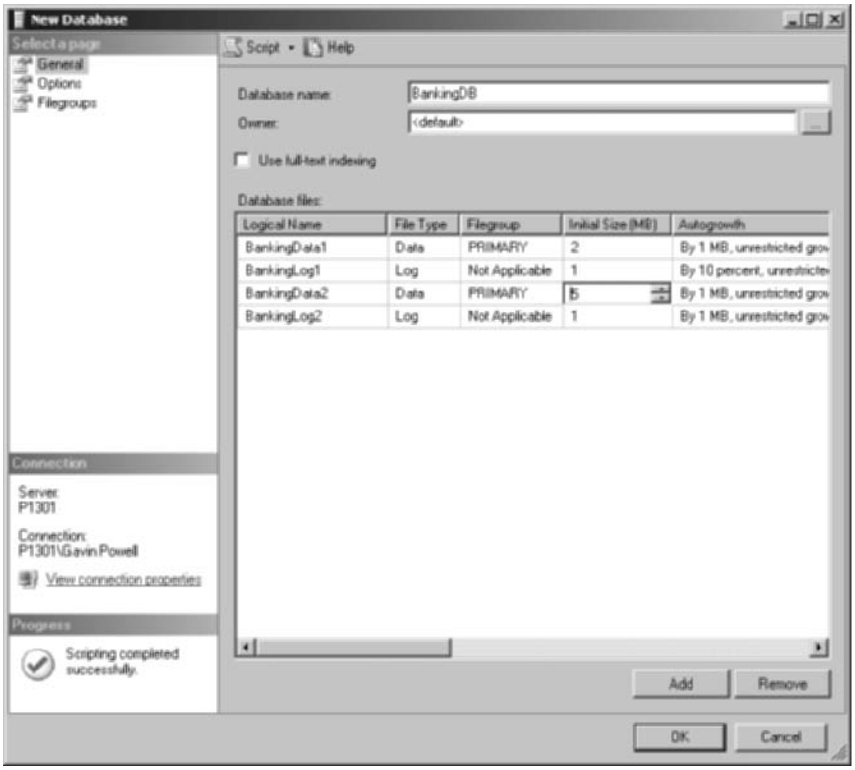
```
CREATE DATABASE BankingDB ON PRIMARY
( NAME = BankingData1, FILENAME = 'd:\data\BankingData1.mdf',
  SIZE = 2MB, MAXSIZE = 100MB, FILEGROWTH = 1MB),
FILEGROUP Filegroup1
( NAME = BankingData2, FILENAME = 'e:\data\BankingData2.ndf',
  SIZE = 2MB, MAXSIZE = 100MB, FILEGROWTH = 1MB)
LOG ON
( NAME = BankingLog1, FILENAME = 'f:\data\BankingLog1.ldf',
  SIZE = 1MB, MAXSIZE = 100MB, FILEGROWTH = 10%),
( NAME = BankingLog2, FILENAME = 'g:\data\BankingLog2.ldf',
  SIZE = 1MB, MAXSIZE = 100MB, FILEGROWTH = 10%)
```

Various attributes of a database can be modified after it has been created. These include increasing and reducing the size of data and transaction log files, adding and removing database and transaction log files, creating filegroups, changing the DEFAULT filegroup, and changing database options.

These operations are achieved by using the ALTER DATABASE statement, DBCC SHRINKFILE, and DBCC SHRINKDATABASE. These operations can also be changed through the SQL Server Management Studio. Let us first look at increasing the size of a database.

4.3 Increasing the size of a database

To increase the size of a database, data and transaction log files may be expanded by using the SQL Server Management Studio or the Transact-SQL ALTER DATABASE statement. Increasing the size of a file in the SQL Server Management Studio is merely a case of entering a new value in the Space allocated (MB) text box, as shown in Figure 4.4.



Database name: BankingDB
Owner: <default>
Use full-text indexing
Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth
BankingData1	Data	PRIMARY	2	By 1 MB, unrestricted grow
BankingLog1	Log	Not Applicable	1	By 10 percent, unrestricte
BankingData2	Data	PRIMARY	5	By 1 MB, unrestricted grow
BankingLog2	Log	Not Applicable	1	By 1 MB, unrestricted grow

Server:
P1301
Connection:
P1301\Gavin Powell
View connection properties
Scripting completed successfully.

Figure 4.4
Increasing the size of a database file

In Transact-SQL, the ALTER DATABASE statement is used, as follows:

```
ALTER DATABASE BankingDB MODIFY FILE
 (NAME = BankingData2, SIZE = 5MB)
```

File attributes such as MAXSIZE and FILEGROWTH may also be modified with an ALTER DATABASE statement.

Another way of increasing the size of a database is to add data and transaction log files, as follows:

```
ALTER DATABASE BankingDB ADD FILE
(NAME = BankingData3, FILENAME = 'h:\data\BankingData3.ndf',
   SIZE = 2MB, MAXSIZE = 100MB, FILEGROWTH = 1MB)
```

The ADD LOG clause is used to add a transaction log file.

To add a file to an existing user-defined filegroup, the ADD FILE ... TO FILEGROUP syntax is used, as follows:

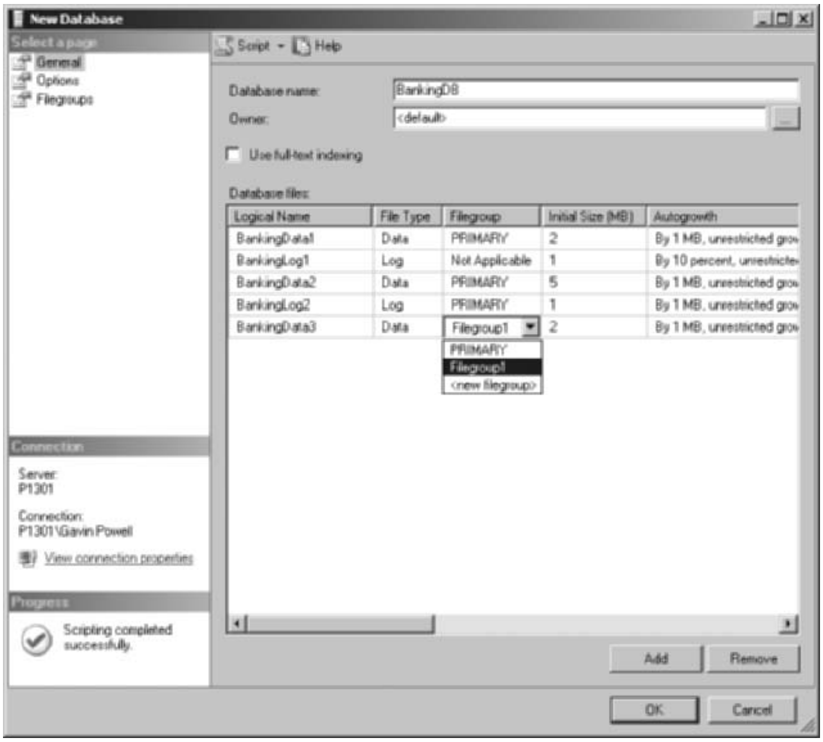
```
ALTER DATABASE BankingDB ADD FILE
  (NAME = BankingData3, FILENAME = 'd:\data\
BankingData3.ndf',
   SIZE = 2MB, MAXSIZE = 100MB, FILEGROWTH = 1MB)
   TO FILEGROUP FileGroup1
```

In the SQL Server Management Studio, adding a new file to an existing filegroup is achieved by selecting the appropriate filegroup from the drop-down File group list, as shown in Figure 4.5.

A file that already exists in the database cannot be subsequently added to another filegroup.

4.4 Decreasing the size of a database

There are a number of mechanisms that can be used to decrease the size of a database. On one hand, a database can be flagged to allow automatic database shrinkage to occur at periodic intervals. This requires no effort on the part of the database administrator, but it also allows no control. On the other hand, DBCC statements can be used to manually shrink a database or individual database files. These DBCC statements provide the database administrator with the greatest control over how the shrinkage takes place. The SQL Server Management Studio also provides a means to shrink a database or file, and this operation can be scheduled under the control of the database administrator.



Database name: BankingDB

Owner: <default>

Use full-text indexing

Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth
BankingData1	Data	PRIMARY	2	By 1 MB, unrestricted grow
BankingLog1	Log	Not Applicable	1	By 10 percent, unrestricte
BankingData2	Data	PRIMARY	5	By 1 MB, unrestricted grow
BankingLog2	Log	PRIMARY	1	By 1 MB, unrestricted grow
BankingData3	Data	Filegroup1	2	By 1 MB, unrestricted grow

PRIMARY
Filegroup1
<new filegroup>

Server:
P1301

Connection:
P1301\Gavin Powell

View connection properties

Scripting completed successfully.

Figure 4.5
Adding a new file to an existing file group

Before we look at shrinking a database, it is worth considering why we might want to do so. Obviously, shrinking a database in a way that physically releases space back to the operating system is an attractive proposition if space is limited on the server. Also disk space must be shared among applications. However, if space is taken away from a database and used by another application, it is no longer available for use by that database. If the database is likely to grow and needs the space in the short term, it is pointless releasing the space. Also, the process of expanding the database files in increments, as previously discussed, is not necessarily efficient. The act of extending a file may impact the performance of applications, and the file extents may end up being fragmented around the disk drive.

However, if a database has grown in an uncharacteristic fashion because a large amount of data has been added and then removed, it makes sense to release the space that is not likely to be needed again. With these thoughts in mind, let us look at how a database and its files can be shrunk.

4.4.1 The autoshrink database option

A database option can be set that makes a database a candidate for automatically being shrunk. Database options and how to set them will be discussed shortly. At periodic intervals a database with this option set may be shrunk if there is sufficient free space in the database to warrant it. Note that the database administrator has no control over exactly what happens and when.

4.4.2 Shrinking a database in the SQL Server Management Studio

A database can be shrunk using SQL Server Management Studio as follows:

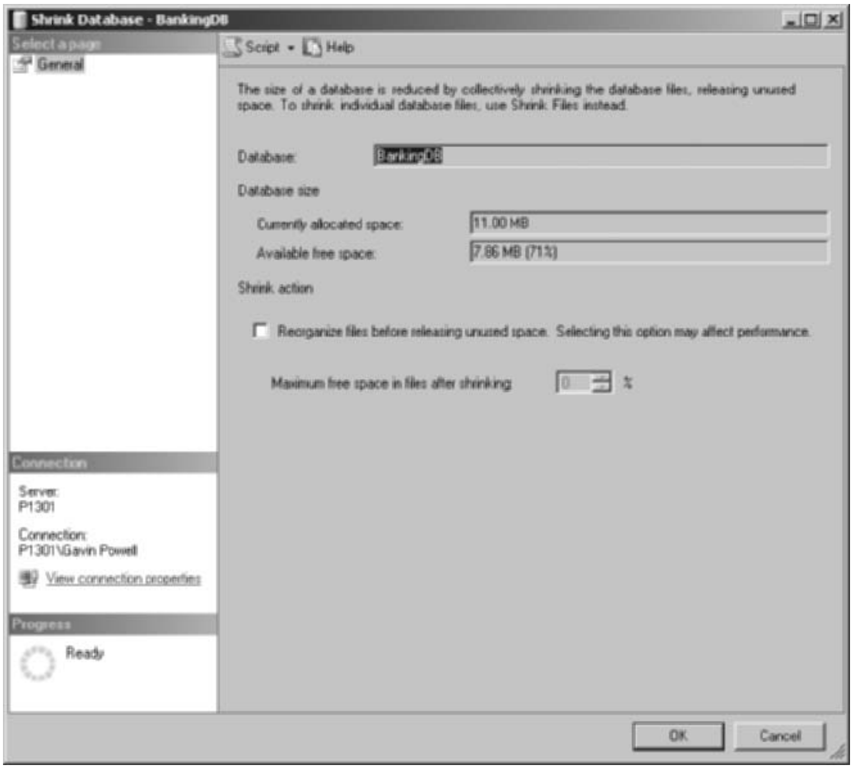
1. Expand the server group and expand the server.
2. Expand Databases, then right-click the database to be shrunk.
3. Select Tasks, Shrink, and Database (select Files to shrink an individual file).
4. Select the desired options.
5. Click OK.

The SQL Server Enterprise Manager Shrink Database dialog box is shown in Figure 4.6.

The dialog box offers the database administrator some options concerning database shrinkage. Selecting the option to reorganize before releasing unused space is effectively a defragmentation process. This will typically result in empty pages at the end of the file. Whether this option is chosen or not, SQL Server will truncate the files, releasing the free space at the end of the files back to the operating system. How much free space is not released but kept at the end of the file can be controlled by the option to specify the maximum free space after shrinking.

This dialog box pretty much maps onto the DBCC SHRINKDATABASE statement, which will be described shortly. There are two restrictions to bear in mind when using a shrink database operation:

- A database cannot be shrunk in such a way that user data is lost.



The size of a database is reduced by collectively shrinking the database files, releasing unused space. To shrink individual database files, use Shrink Files instead.

Database: BankingDB

Database size

Currently allocated space: 11.00 MB

Available free space: 7.86 MB (71%)

Shrink action

☐ Reorganize files before releasing unused space. Selecting this option may affect performance.

Maximum free space in files after shrinking: 0 %

Server:
P1301

Connection:
P1301\Gavin Powell

View connection properties

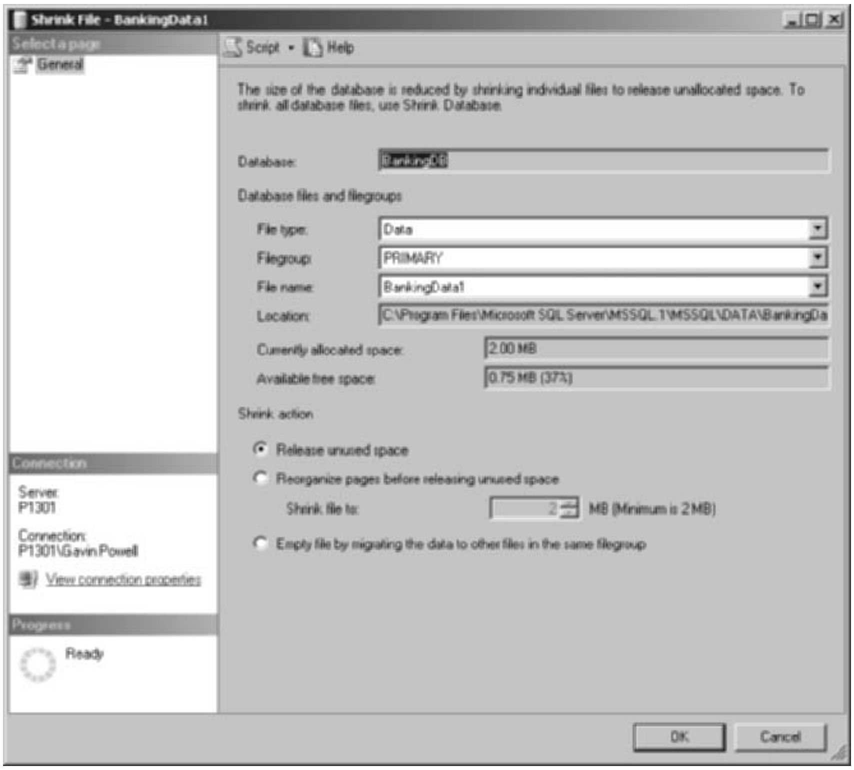
Ready

Figure 4.6
Shrinking a database using the SQL Server Management Studio

- The files that comprise the database cannot be shrunk past their initial size (the size at which they were initially created).

For greater control, individual files can be shrunk (see Figure 4.6), as opposed to a database as a whole. The Shrink File dialog box is displayed in Figure 4.7.

When a file is shrunk using this dialog box, it can be shrunk below its initial creation size as long as user data would not be lost. Various options allow a finer level of control. The file can be reorganized (compressed), and the free space truncated from the end. The target file size can be set using the shrink file to option. This option will compress and truncate. There is also an option to migrate data from the file to other files in its filegroup so it can be emptied and then removed from the database. This dialog box pretty much maps onto the DBCC SHRINKFILE statement described in the following section.



The size of the database is reduced by shrinking individual files to release unallocated space. To shrink all database files, use Shrink Database.

Database: BankingDB

Database files and filegroups

File type: Data
Filegroup: PRIMARY
File name: BankingData1
Location: C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\DATA\BankingDa
Currently allocated space: 2.00 MB
Available free space: 0.75 MB (37%)

Shrink action

(●) Release unused space
( ) Reorganize pages before releasing unused space
Shrink file to: 2 MB (Minimum is 2 MB)
( ) Empty file by migrating the data to other files in the same filegroup

Server: P1301
Connection: P1301\Gavin Powell
View connection properties

Ready

Figure 4.7 *Decreasing the size of a database*

4.4.3 Shrinking a database using DBCC statements

The greatest control over database shrinkage is provided by two DBCC statements—DBCC SHRINKDATABASE and DBCC SHRINKFILE. The first statement considers all the files in the database when attempting to shrink it. The second statement only considers the named file.

The SQL Server Enterprise Manager actually executes a DBCC SHRINKDATABASE statement when it is used to shrink a database and a DBCC SHRINKFILE statement when it is used to shrink a database file.

Let us first consider DBCC SHRINKDATABASE. The syntax diagram for this statement is as follows:

```
DBCC SHRINKDATABASE
( database_name [, target_percent]
[, {NOTRUNCATE | TRUNCATEONLY}]
)
```

The target percent parameter is the desired percentage of free space left in the database file after the database has been shrunk. If this parameter is omitted, SQL Server will attempt to shrink the database as much as possible.

The NOTRUNCATE option ensures that any free file space produced by relocating data is kept within the database files and not given back to the operating system. If the database files were examined with Windows Explorer before and after the shrink operation, no change in file size would be observed.

The TRUNCATEONLY option ensures that any free space at the end of the data files is returned to the operating system but no data is relocated within the files. If the database files were examined with Windows Explorer before and after the shrink operation, a change in file size may be observed. The target_percent parameter is disregarded when the TRUNCATEONLY option is used.

If neither of these is specified, data is relocated in the files, and the free space at the end of the files is released to the operating system.

The operation of shrinking a database is not quite as straightforward as it first appears. Various restrictions come into play, and you may not always see shrinkage as large as you may expect. For example, as we have said, a database file cannot be shrunk, using DBCC SHRINKDATABASE, smaller than the size at which it was first created. Also, a database cannot be shrunk smaller than the model database (a DBCC SHRINKFILE can shrink a file smaller than its initial size). Data files and transaction log files are also treated differently. In the case of data files, each file is considered individually. In the case of transaction log files, all the files are treated as if they were one contiguous lump of transaction log.

Of course, a database can never be shrunk smaller than the amount of data it currently holds.

Let us now consider DBCC SHRINKFILE. The syntax diagram for this statement is as follows:

```
DBCC SHRINKFILE
( {file_name | file_id }
{ [, target_size]
| [, {EMPTYFILE | NOTRUNCATE | TRUNCATEONLY}]
}
)
```

The target size parameter is the desired size, to which the database file should be shrunk. If this parameter is omitted, SQL Server will attempt to shrink the file as much as possible.

The NOTRUNCATE and TRUNCATEONLY options have the same meaning as DBCC SHRINKDATABASE. The EMPTYFILE option moves the data contained in the file to other files that reside in the same filegroup and stops the file being used to hold new data. This option is most often used to prepare a file for removal from the database. It could not otherwise be removed if it contained data.

Files can be removed from the database by using the ALTER DATBASE statement. Neither data files nor transaction log files can be removed from a database if they contain data or transaction log records. In the case of data files, the DBCC SHRINKFILE statement with the EMPTYFILE option can be used to move data out of the file that is to be removed to other files in the same filegroup. This is not possible in the case of transaction log files. The transaction log will have to be truncated to remove transaction log records before the removal of a transaction log file is possible.

The following example removes a file from the BankingDB database created earlier:

```
ALTER DATABASE BankingDB REMOVE FILE BankingData2
```

Removing a file using the SQL Server Enterprise Manager is merely a case of selecting the file to remove and mouse-clicking the Delete button, as shown in Figure 4.8.

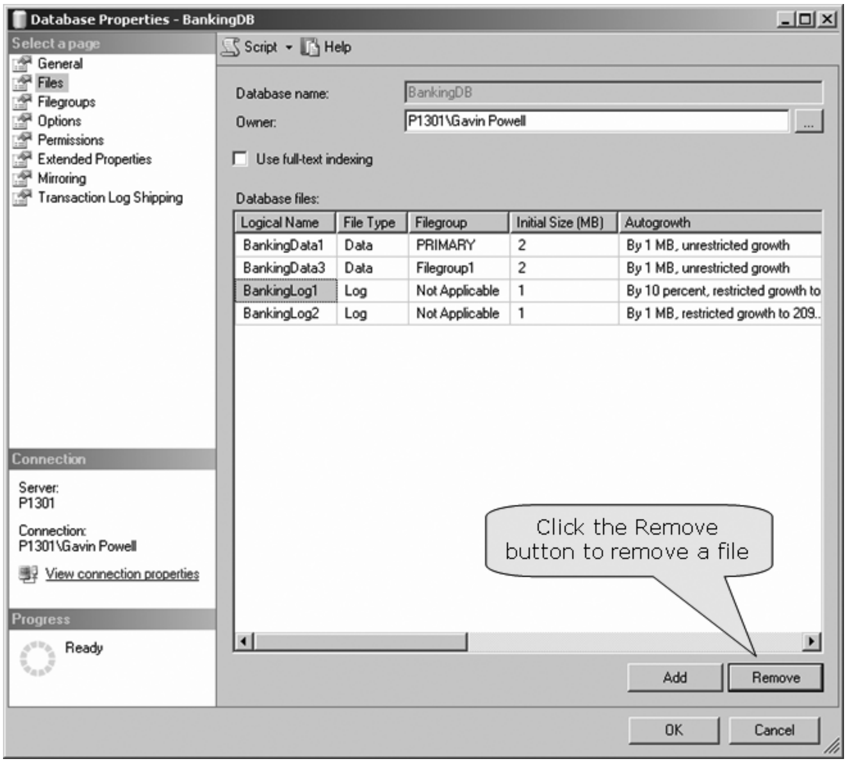
A filegroup can also be removed, as follows:

```
ALTER DATABASE BankingDB REMOVE FILEGROUP FileGroup1
```

However, a filegroup cannot be removed if it contains files.

4.5 Modifying filegroup properties

The properties of a filegroup can be changed. Filegroup properties can be READWRITE, READONLY, and DEFAULT. The READWRITE property is typically the property that is set for most filegroups. This means that objects such as tables and indexes in the filegroup can be both retrieved and changed. The READONLY property is the opposite of the READWRITE



Database name: BankingDB
Owner: P1301\Gavin Powell
Use full-text indexing
Server: P1301
Connection: P1301\Gavin Powell

Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth
BankingData1	Data	PRIMARY	2	By 1 MB, unrestricted growth
BankingData3	Data	Filegroup1	2	By 1 MB, unrestricted growth
BankingLog1	Log	Not Applicable	1	By 10 percent, restricted growth to
BankingLog2	Log	Not Applicable	1	By 1 MB, restricted growth to 209...

Figure 4.8 *Removing a file with the SQL Server Management Studio*

property in that those objects in a filegroup with the READONLY property set cannot be changed; they can only be retrieved. The primary file-group cannot have this property set.

The DEFAULT property is by default set on the primary filegroup. A filegroup with this property set is used to store objects whose definition does not include a target filegroup specification. The DEFAULT property can be set on a filegroup other than the primary filegroup, but only one file-group in a database can have this property set. The following example sets the READONLY attribute on the filegroup FileGroup1:

```
ALTER DATABASE BankingDB MODIFY FILEGROUP FileGroup1 READONLY
```

Setting the properties READONLY or READWRITE requires exclusive use of the database.

4.6 Setting database options

Database options are the attributes of a database and controlling its behavior and capabilities. The database options are listed in Table 4.1.

Table 4.1 *Database options*

Settable Database Options	Meaning
`ANSI null default`	This option controls the database default nullability. If a table column is created without specifying NULL or NOT NULL, the default behavior is to create the column with NOT NULL. However, the ANSI standard specifies that the column should be created with NULL. Set this option to follow the ANSI standard. It is recommended that NULL or NOT NULL always be explicitly specified to avoid confusion.
`ANSI nulls`	This option controls the result of comparing NULL values. If it is set, comparisons with a NULL value evaluate to NULL, not TRUE or FALSE. When not set, comparisons of non-Unicode values with a NULL value evaluate to TRUE if both values are NULL.
`ANSI padding`	If ON, strings are padded to the same length before comparison or insert. If OFF, strings are not padded.
`ANSI warnings`	This option controls whether warnings are issued if, for example, NULL values appear in aggregate functions.
`arithabort`	If ON, a query is terminated when an overflow or divide-by-zero error occurs during the execution of the query.
`auto create statistics`	This option controls whether statistics are automatically created on columns used in the search conditions in WHERE clauses.
`auto update statistics`	This option controls whether existing statistics are automatically updated when the statistics become inaccurate because the data in the tables have changed.
`autoclose`	This option controls whether a database is shut down and its resources released when the last user finishes using it.
`autoshrink`	This option controls whether a database is a candidate for automatic shrinking.
`concat null yields null`	This option controls whether NULL is the result of a concatenation if either operand is NULL.

Table 4.1 *Database options (continued)*

`cursor close on commit`	This option controls whether cursors are closed when a transaction commits.
`default to global cursor`	This option controls whether cursors are created locally or globally when this is not explicitly specified.
`numeric roundabort`	If ON, an error is generated when loss of precision occurs in an expression.
`offline`	This option ensures that the database is closed and shut down cleanly and marked offline.
`quoted identifier`	This option controls whether identifiers can be delimited by double quotation marks.
`read only`	This option controls whether a database can be modified.
`recursive triggers`	This option controls whether triggers can fire recursively.
`single user`	This option limits database access to a single user connection.
`auto update statistics asynchronization`	Reduces dependence between query optimization and the existence of up to date statistics. However, executing a query without up to date statistics can cause the query to possibly run slower.
`date correlation optimization`	Can help performance of join queries, with joins on date and time fields
`simple parameterization`	Allows for better SQL optimization, by matching previously compiled SQL statements with submitted SQL statements. In other words, reduces hard parsing on queries.
`full recovery`	Allows for a full database recovery.
`page verification checksum`	Used for high availability and scalability in tandem with torn page detection. (This option allows incomplete I/O operations to be detected.)

The following is a simple database creation script generated by SQL Server Management Studio, showing all the default options:

```
CREATE DATABASE test ON  PRIMARY
( NAME = 'test', FILENAME = 'C:\data\test.mdf',
  SIZE = 2MB , FILEGROWTH = 1MB )
LOG ON
```

```
( NAME = 'test_log', FILENAME = 'C:\data\test_log.ldf',
  SIZE = 1MB , FILEGROWTH = 10%)
GO
ALTER DATABASE test SET ANSI_NULL_DEFAULT OFF
GO
ALTER DATABASE test SET ANSI_NULLS OFF
GO
ALTER DATABASE test SET ANSI_PADDING OFF
GO
ALTER DATABASE test SET ANSI_WARNINGS OFF
GO
ALTER DATABASE test SET ARITHABORT OFF
GO
ALTER DATABASE test SET AUTO_CLOSE OFF
GO
ALTER DATABASE test SET AUTO_CREATE_STATISTICS ON
GO
ALTER DATABASE test SET AUTO_SHRINK OFF
GO
ALTER DATABASE test SET AUTO_UPDATE_STATISTICS ON
GO
ALTER DATABASE test SET CURSOR_CLOSE_ON_COMMIT OFF
GO
ALTER DATABASE test SET CURSOR_DEFAULT GLOBAL
GO
ALTER DATABASE test SET CONCAT_NULL_YIELDS_NULL OFF
GO
ALTER DATABASE test SET NUMERIC_ROUNDABORT OFF
GO
ALTER DATABASE test SET QUOTED_IDENTIFIER OFF
GO
ALTER DATABASE test SET RECURSIVE_TRIGGERS OFF
GO
ALTER DATABASE test SET AUTO_UPDATE_STATISTICS_ASYNC OFF
GO
ALTER DATABASE test SET DATE_CORRELATION_OPTIMIZATION OFF
GO
ALTER DATABASE test SET PARAMETERIZATION SIMPLE
GO
ALTER DATABASE test SET  READ_WRITE
GO
```

```
ALTER DATABASE test SET RECOVERY FULL
GO
ALTER DATABASE test SET  MULTI_USER
GO
ALTER DATABASE test SET PAGE_VERIFY CHECKSUM
GO
```

Both SQL Server Management Studio and the ALTER DATABASE statement can be used to set a database option. To use the SQL Server Enterprise Manager, do the following:

1. Expand the server group and expand the server.
2. Expand Databases, then right-click the database whose options are to be set.
3. Select Properties.
4. Select the Options tab and the required options.
5. Click OK. The SQL Server Enterprise Manager Options tab is shown in Figure 4.9.

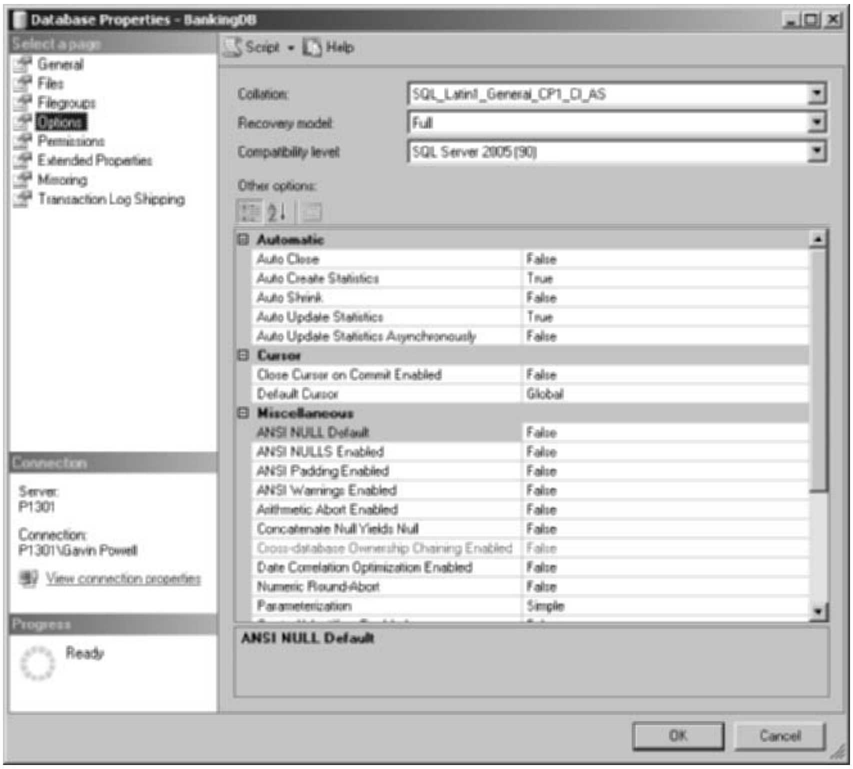
Because some options—for example, replication options—are set by other parts of the SQL Server Management Server, the options displayed in the Options tab are a subset of the available database options.

The following example sets a database option using Transact-SQL:

```
ALTER DATABASE BankingDB SET AUTO_SHRINK ON
```

4.7 Displaying information about databases

Information about databases can be obtained through the SQL Server Management Studio, or various Transact-SQL statements. We have already seen the properties page that is displayed when a database is right-clicked and Properties selected. This shows us quite a lot of information, including the files that comprise the database. An example of this is shown in Figure 4.4. Right-click a database in the SQL Server Management Studio, and select the Properties option. Figure 4.10 shows general options under database properties.



Collation: SQL_Latin1_General_CP1_CI_AS
Recovery model: Full
Compatibility level: SQL Server 2005 (90)
Other options:

Automatic	
Auto Close	False
Auto Create Statistics	True
Auto Shrink	False
Auto Update Statistics	True
Auto Update Statistics Asynchronously	False
Cursor	
Close Cursor on Commit Enabled	False
Default Cursor	Global
Miscellaneous	
ANSI NULL Default	False
ANSI NULLS Enabled	False
ANSI Padding Enabled	False
ANSI Warnings Enabled	False
Arithmetic Abort Enabled	False
Concatenate Null Yields Null	False
Cross-database Ownership Chaining Enabled	False
Date Correlation Optimization Enabled	False
Numeric Round-Abort	False
Parameterization	Simple

ANSI NULL Default

Server:
P1301

Connection:
P1301\Gavin Powell

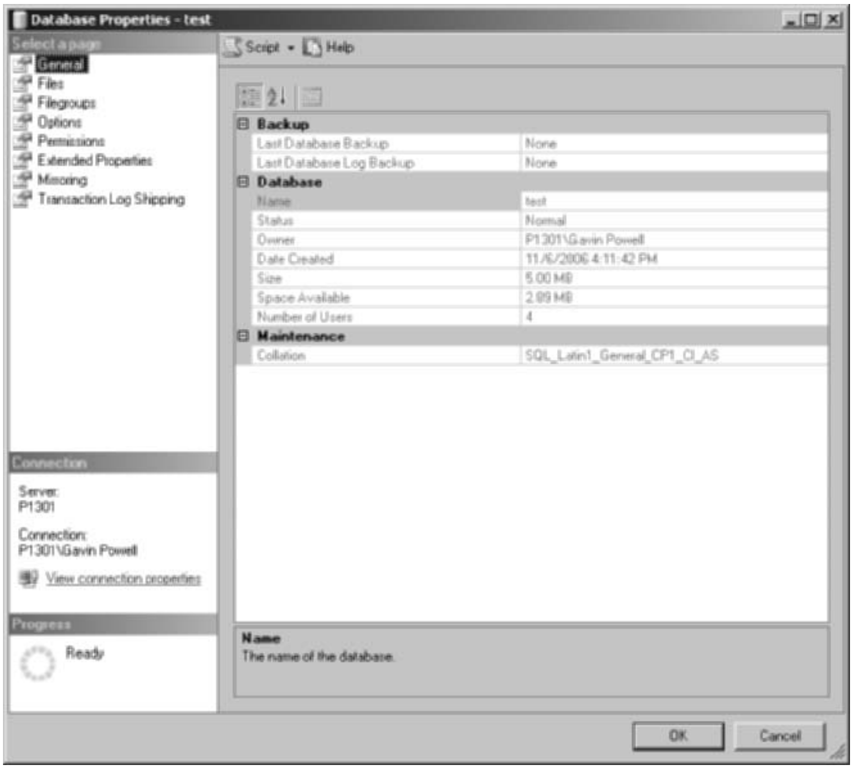
Ready

Figure 4.9
Setting database options

To examine properties for other SQL Server objects, such as a table, you have to select properties from that object, as shown in Figure 4.11.

In Transact-SQL, the sp_helpdb system stored procedure is very useful. This is as follows:

```
EXEC sp_helpdb

name         db_size     owner dbid created       status

BankingDB     1500.00 MB    sa 6     Oct 23 2000   Status=ONLINE…
Derivatives     25.00 MB    sa 8     Oct 18 2000   Status=ONLINE…
master          17.00 MB    sa 1     Oct 12 2000   Status=ONLINE…
model            1.00 MB    sa 3     Oct 12 2000   Status=ONLINE…
msdb             8.00 MB    sa 5     Oct 12 2000   Status=ONLINE…
pubs             3.00 MB    sa 4     Oct 12 2000   Status=ONLINE…
tempdb           2.00 MB    sa 2     Oct 19 2000   Status=ONLINE…
```



Server:
P1301

Connection:
P1301\Gavin Powell

Backup	
Last Database Backup	None
Last Database Log Backup	None
Database	
Name	test
Status	Normal
Owner	P1301\Gavin Powell
Date Created	11/6/2006 4:11:42 PM
Size	5.00 MB
Space Available	2.09 MB
Number of Users	4
Maintenance	
Collation	SQL_Latin1_General_CP1_CI_AS

Name
The name of the database.

Figure 4.10
Viewing the details of a database

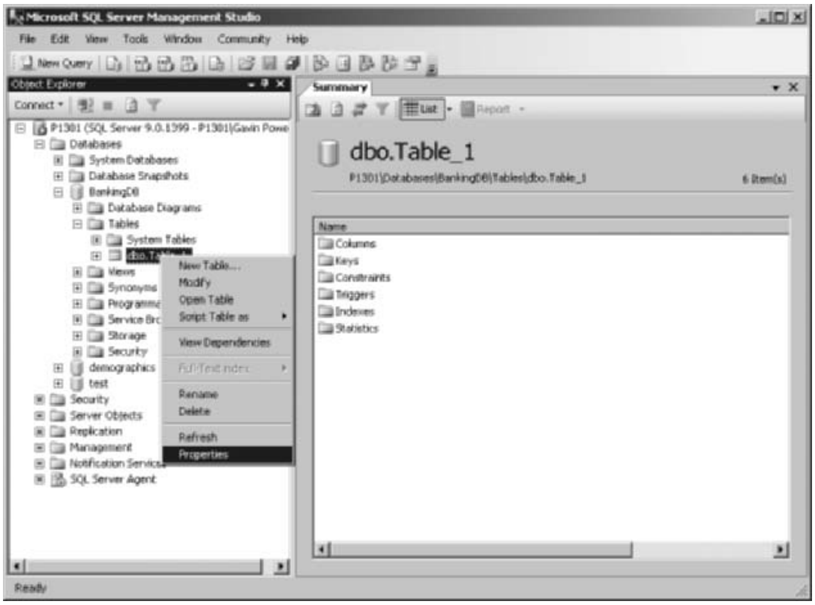


Figure 4.11
Displaying space allocation information

This outputs one row for each database on the server. The db_size column is the total size of all the files in the database. A database name can be specified as a parameter, as follows:

```
EXEC sp_helpdb BankingDB

name         db_size     owner  dbid    created      status

BankingDB    1500.00 MB    sa   6       Oct 23 2000  Status=ONLINE…

Name    fileid  filename                 filegroup   size       maxsize  growth   usage

bankingdata  1  d:\data\bankingdata.mdf  PRIMARY  1024000 KB  Unlimited  1024 KB  data only
bankinglog   2  d:\data\bankinglog.ldf   NULL      512000 KB  Unlimited  1024 KB  log only
```

This displays information about the files in the database. Other useful system-stored procedures, which can be used to obtain information about files and filegroups, are sp_helpfile and sp_helpfilegroup. Another useful system-stored procedure is sp_spaceused, which returns space use information.

SQL OS (SQL Server Operating System) is a new tool used to examine a SQL Server 2005 installation, at the point between the database server and the underlying Windows operating system. SQL OS is thus more of a monitoring tool, as opposed to simple examination, and thus is presented in a later chapter which covers performance monitoring.

4.8 System tables used in database configuration

The configuration of a database is reflected in various system tables held in the master database and the user database. The master database contains a system table, SYSDATABASES, which contains one row for every database resident on the SQL Server. The structure of this system table is shown in Table 4.2.

As can be seen, the SYSDATABASES system table contains a column, filename, which points to the primary data file (.MDF) of a database on the server. This is the pointer from the master database to each user database. Once the primary data file of a database has been located, the SYSFILES system table, which resides in every database, can be located. This has one row representing each file—data or log—found in the database. The SYS-

Table 4.2 *The SYSDATABASES system table*

Column	Datatype	Description
name	sysname	The database name
dbid	smallint	The unique ID of the database
sid	varbinary(85)	The Windows NT system ID of the database creator
mode	smallint	Internal lock mechanism used in database creation
status	integer	Database status bits (O = set by sp_dboption): 1 = autoclose (O) 4 = select into/bulkcopy (O) 8 = trunc. log on chkpt (O) 16 = torn page detection (O) 32 = loading 64 = prerecovery 128 = recovering 256 = not recovered 512 = offline (O) 1,024 = read only (O) 2,048 = dbo use only (O) 4,096 = single user (O) 32,768 = emergency mode 4,194,304 = autoshrink 1,073,741,824 = cleanly shut down

Table 4.2 *The SYSDATABASES system table (continued)*

status2	integer	16,384 = ANSI null default (O) 2,048 = numeric roundabort (O) 4,096 = arithabort (O) 8,192 = ANSI padding (O) 65,536 = concat null yields null (O) 131,072 = recursive triggers(O) 1,048,576 = default to local cursor (O) 8,388,608 = quoted identifier (O) 33,554,432 = cursor close on commit (O) 67,108,864 = ANSI nulls (O) 268,435,456 = ANSI warnings (O) 536,870,912 = full text enabled
crdate	datetime	Date when database was created
reserved	datetime	Reserved by Microsoft
category	integer	Contains a bitmap used for replication: 1 = Published 2 = Subscribed 4 = Merge Published 8 = Merge Subscribed
cmptlevel	tinyint	Set by sp_dbcmptlevel—specifies the database compatibility level
filename	nvarchar(260)	Location of the primary data file for this database
version	smallint	SQL Server internal code version that created the database

FILES system table is shown in Table 4.3.

Table 4.3 *The SYSFILES system table*

Column	**Datatype**	**Description**
fileid	smallint	Unique identifier for the file within the database
groupid	smallint	Identifier of the filegroup to which the file belongs
size	integer	File size in (8 KB) database pages
maxsize	integer	Maximum file size in (8 KB) database pages. 0 = no growth and -1 = unlimited growth.

Table 4.3 *The SYSFILES system table (continued)*

growth	integer	Growth increment of the file. 0 = no growth. This is in units of 8 KB pages or a percentage, depending on the status column. If the status column contains 0x100,000, then growth is in percentage, not pages.
status	integer	Status bits for the growth value in either megabytes (MB) or kilobytes (K): 0x1 = Default device 0x2 = Disk file 0x40 = Log device 0x80 = File has been written to since last backup 0x4000 = Device created implicitly by the CREATE DATABASE statement 0x8000 = Device created during database creation 0x100000 = Growth is in percentage, not pages
perf	integer	Reserved by Microsoft
name	nchar(128)	Logical name of the file
filename	nchar(260)	Full path of filename

Table 4.4 *The SYSFILEGROUPS system table*

Column	**Datatype**	**Description**
groupid	smallint	Unique identifier for the filegroup within the database
allocpolicy	smallint	Reserved by Microsoft
status	int	0x8 = READ ONLY 0x10 = DEFAULT
groupname	sysname	Filegroup name

One other system table found in each database is worthy of note at this point: the SYSFILEGROUPS system table, which contains one row for every filegroup in the database. The SYSFILEGROUPS system table is shown in Table 4.4.

All of these tables can be queried with SELECT statements, but it is easier to use the system stored procedures provided, namely sp_helpdb, sp_helpfile, and sp_helpfilegroup. We have already seen an example of sp_helpdb. Examples of sp_helpfile and sp_helpfilegroup are as follows:

```
EXEC sp_helpfile

Name    fileid filename                filegroup        size  maxsize    growth   usage

bankingdata 1 d:\data\bankingdata.mdf  PRIMARY   1024000 KB  Unlimited  1024 KB  data only
bankinglog  2 d:\data\bankinglog.ldf   NULL       512000 KB  Unlimited  1024 KB  log only

EXEC sp_helpfilegroup

groupname    groupid filecount

PRIMARY
```

4.9 Units of storage

A database is a collection of logical pages, each 8 KB in size. Database pages are always this size and cannot be adjusted by the database designer. The 8 KB page is the fundamental unit of storage and it is also a unit of I/O and a unit of locking (there are other units of I/O and locking).

Tables and indexes consist of database pages. The way that database pages are allocated to tables and indexes is through extents.

An extent is a structure that contains eight database pages (64 KB). Extents are of two types—uniform and mixed. A uniform extent devotes its eight pages completely to one object, for example, a particular table in the database. A mixed extent allows its pages to be used by up to eight different objects. Although each page can only be used for one object, all eight pages in a mixed extent can be used by different objects. For example, a mixed extent can provide space for eight tables. A uniform extent is shown in Figure 4.12.

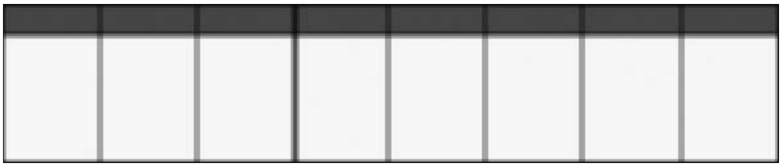


Figure 4.12
A uniform extent

A mixed extent is shown in Figure 4.13.

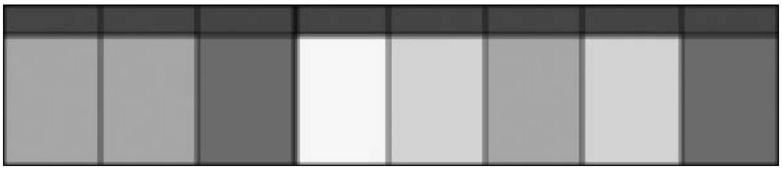


Figure 4.13
A mixed extent

The reason that SQL Server 2000 uses mixed extents is to ensure that a whole eight page (64 KB) extent is not used for a small table. Instead, single pages are allocated to the table one at a time as the number of rows it contains grows. When eight pages have been allocated and more pages are needed, uniform extents are used to allocate eight pages at a time.

To observe the allocation of space to a table, try the following:

1. Create a table, T1, with a single column of data type CHAR(8000). A single row only can fit onto a database page.
2. Insert eight rows, one at a time, checking the space allocated to the table with the sp_spaceused system stored procedure after each insert (e.g., EXEC sp_spaceused T1).
3. Insert another row, checking the space reserved.

What you will find is that after each row is inserted, the data column (the amount of space used by data in the table) is incremented by 8 KB—that is, a single page from a mixed extent. The reserved column (the amount of total reserved space for the table) is also incremented by 8 KB.

The reserved column displays 8 KB more than the data column, since a page is used in the table's page allocation to hold a special structure called an Index Allocation Map (IAM), which we will discuss later. This is the 8 KB that is displayed in the Index_Size column. There is no index space actually used, since there is no index on this table.

After eight rows have been inserted, the data column will display 64 KB and the reserved column will display 72 KB. After row 9 is inserted, however, the data column will display 72 KB but the reserved column will display 136 KB. This is because a whole eight page uniform extent has now been allocated to the table, causing the reserved value to jump by 64 KB.

Let us have a quick look at the sp_spaceused system stored procedure. To see the space allocated to a table use the system stored procedure sp_spaceused, as follows:

```
sp_spaceused branches

name     rows reserved   data   index_size  unused
branches 100     72 KB  64 KB        8 KB    0 KB
```

In the above example, sp_spaceused reports that there are 100 rows in the Branches table and that 72 KB or 9 database pages of space have been reserved for it. Out of the 9 pages, 8 pages have been used by the table to store rows and another 1 page has been used for index space which, as mentioned above, is actually used by an IAM page. Note that the system stored procedure sp_spaceused gets its information from the sysindexes system table, which only holds estimates. It does this to avoid becoming a bottleneck at run time, but it can become inaccurate. To synchronize the sysindexes system table with the real space used, execute a DBCC CHECKTABLE or a DBCC UPDATEUSAGE statement, which will scan the table and indexes.

4.10 Database pages

Database pages are used for a variety of tasks. Database pages that are used to hold table rows and index entries are known as data pages and index pages, respectively. If the table contains columns of the data type TEXT or IMAGE, then these columns are usually implemented as structures of Text/Image pages (unless the TEXT/IMAGE data is stored in the row). There are other types of pages also, namely Global Allocation Map (GAM) pages, Page Free Space (PFS), and Index Allocation Map (IAM) pages. We will discuss these types of pages shortly.

First, though, let us take out the magnifying glass and take a closer look at a typical page structure. The most common database page we are likely to meet is a data page, so we will use a data page as an example.

The basic structure of all types of database pages is shown in Figure 4.14.

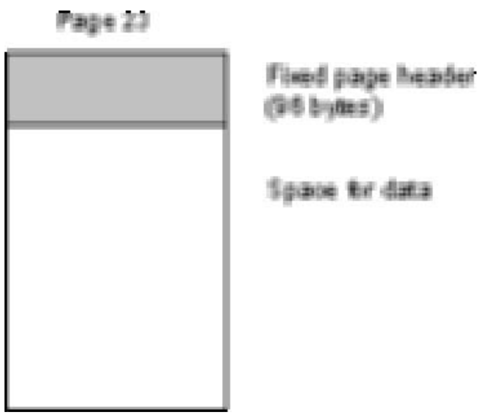


Figure 4.14
The basic structure of a database page

There is a fixed 96-byte page header, which contains information such as the page number, pointers to the previous and next page (if used), and the object ID of the object to which the page belongs. The pointers are needed, because pages are linked together, as shown in Figure 4.15. However, this only happens in certain circumstances.

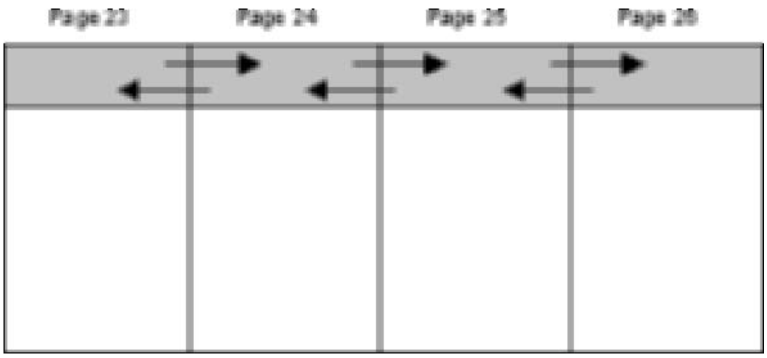


Figure 4.15
Pages linked in a chain

What does a data page look like inside? The internal structure of a data page is shown in Figure 4.16. We can see the data rows, but there is also another structure called a row offset table. The row offset table contains two byte entries consisting of the row number and the offset byte address of the row in the page. The first row in our page is at byte offset 96, because of the 96-byte page header. Our row (plus overhead) is 20 bytes in length, so the next row is at byte offset 116 and so on. The row offset table basically gives us a level of indirection when addressing a row. This is important because non-clustered indexes may contain pointers to data rows in their leaf-level index pages. Such a pointer is known as a Row ID and is made up of a File ID, database page number, and a row number. The File ID and database page number (a Page ID) take SQL Server to an individual page in a file and the row number and then takes SQL Server to an entry in the row offset table. In our example, the Row ID of the row nearest the fixed page header would consist of the page number, 23, and the row number, 0.

Entry 0 in the row offset table contains byte offset address 96. SQL Server can then use this offset to retrieve the row. Because the Row ID is implemented this way, we can see that a row can change position in the table without the Row ID having to change. All that has to change is the offset address in the row offset table entry. Why would a row change position in a page? In Figure 2.16, if row 1 were deleted, row 2 may move up to row 0 in order to keep the free space in the page contiguous if a new row needed to be inserted. The Row ID for row 2 would not change.

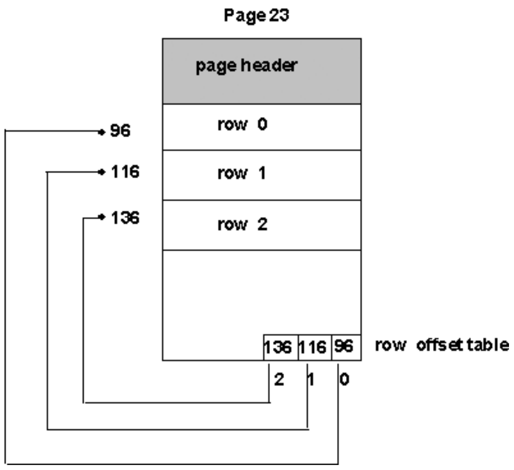


Figure 4.16
The internals of a data page

SQL Server will not shuffle rows like this for the sake of it. It will only do so to accommodate new inserts on the page.

What does a data row look like inside? Data rows contain columns of data, as you would expect, but they also contain overhead. The amount of overhead depends on whether the row contains all fixed-length columns or whether there are also variable-length columns. In Figure 4.17 we have the structure of the Accounts table row in our BankingDB database.

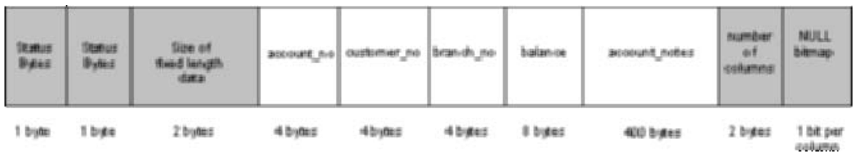


Figure 4.17
A row containing only fixed length columns

The Accounts table has five fixed-length columns. The first three columns are of type integer, the fourth column is of type money, and the last column is of type char(400).

The first two bytes are used for status bits. The first status byte holds information that tells SQL Server, for example, whether the row is a primary data row or a forwarded row. A status bit in this byte also specifies whether there is variable-length data in the row. In our example there are no variable-length data.

The next two bytes hold a number representing the length of the fixed data in the row. This number is the number of bytes of data plus the two status bytes and these two bytes themselves.

The fixed-length data now follow. Finally, there are two bytes holding a number that represents the number of columns in the row and a variable number of bytes holding a NULL bitmap. This contains one bit per every column with a bit set to show whether the column contains a NULL value. (See Figure 4.17.)

The shaded area represents the overhead. Our Account row, which we expected to be 420 bytes in length, has turned out to be 424 bytes in length—and that does not include the fields holding the number of columns and the NULL bitmap.

Suppose the last column in our Accounts table was not a char(400) data type but a varchar(400). The structure of our row containing variable length data is shown in Figure 4.18.

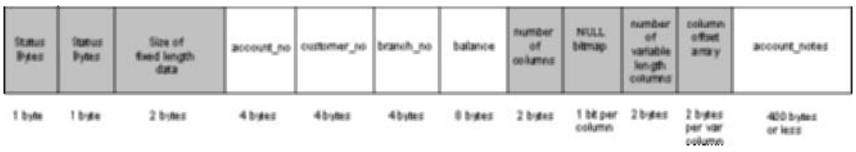


Figure 4.18 *A row containing fixed length and variable length columns*

The structure shown in Figure 4.18 assumes that the account_notes column does indeed contain 400 characters. If it contains less, then less bytes will be used to hold the account notes. We can immediately see two differences between the structure of a row containing only fixed-length columns and a row that also contains variable-length columns. First, the fixed-length columns are grouped together separate from the variable-length columns, which are also grouped together. Second, there are more overhead bytes.

Looking at the structure, the first status byte will now have a bit set to specify that variable-length columns are present in the row. After the two status bytes the next two bytes hold a number representing the length of the fixed data in the row followed by the fixed data, the two-byte field holding the number of columns, and the NULL bitmap. Now we find extra fields. A two-byte field holds the number of variable-length columns followed by a field known as the column offset array, which contains a two-byte cell for each variable-length column used to hold information that SQL Server uses to find the position of the variable-length data.

We can see that the order of the columns in a row that contains variable-length columns is not the same order as the table definition.

SQL Server also allows small amounts of text or image data to be held inside the row. Normally, text and image data is held outside the row.

4.11 Looking into database pages

I often find it useful and educational to be able to burrow into the contents of a database page. A useful DBCC statement that will allow you to do this is DBCC PAGE. This DBCC statement is not documented as an option of the DBCC statement in the Microsoft SQL Server documentation; however, some references to it can be found on TechNet and various other sources.

The most useful form of the syntax of this statement is:

```
DBCC PAGE (dbid  | dbname, file id, page number)
```

or:

```
DBCC PAGE (dbid  | dbname, file id, page number, 1)
```

The first form of the syntax displays the page header; the second form also displays the contents of the page—that is, data in the form of rows and the row offset table.

How do you know which page number to display? One of the columns in the sysindexes system table, described in Chapter 3, contains a column first. This contains the Page ID (File ID plus page number) of the first data page in the table if the sysindexes entry is a table or clustered index (indid = 0 or 1). Also, if the sysindexes entry is a table, the root column holds the Page ID of the last data page in the table.

To find the relevant entry in the sysindexes table you need to convert the table name to an Object ID because the sysindexes table holds the Object ID rather than the table name. The Object_ID function can be used to translate the table name to its Object ID. For example, suppose we want to look at pages in the Accounts table. To get the start Page ID from the sysindexes table, use the following example:

```
SELECT first FROM sysindexes WHERE
  id = OBJECT_ID ('accounts')
 AND
  indid IN (0,1)

first
-----
```

```
0x1E0000000100
```

Unfortunately, the Page ID is represented in hexadecimal and a swapped byte order, so some manipulation will be needed to arrive at the page number. First of all, take away the 0x symbol and separate the number into one-byte (two-digit) values:

```
1E 00 00 00 01 00
```

Now you must reverse the order of the bytes:

```
00 01 00 00 00 1E
```

The first two bytes hold the File ID number, and the last four bytes hold the page number:

```
00 01 | 00 00 00 1E
```

Therefore, in our example, the File ID number is 1 and the page number is 30 (the decimal equivalent of hexadecimal 1E).

To get information out of DBCC PAGE we must initiate tracing to the client:

```
DBCC TRACEON (3604)
```

We are now ready to display the contents of a page, but first of all let us just display the page header so we can see what is in it:

```
DBCC PAGE ('BankingDB',1,30)

PAGE: (1:30)
------------
BUFFER:
-------
BUF @0x18F0BF80
---------------
bpage = 0x1B14C000  bhash = 0x00000000  bpageno = (1:30)
bdbid =     breferences = 1     bstat = 0x9
```

```
bspin = 0   bnext = 0x00000000

PAGE HEADER:
------------
Page @0x1B14C000
----------------
m_pageId = (1:30)         m_headerVersion = 1  m_type = 1
m_typeFlagBits = 0x0      m_level = 0           m_flagBits = 0x8000
m_objId = 199305813       m_indexId = 0         m_prevPage = (0:0)
m_nextPage = (0:0)        pminlen = 424         m_slotCnt = 16
m_freeCnt = 1232 m_freeData = 6928  m_reservedCnt = 0
m_lsn = (5:84:25)         m_xactReserved = 0  m_xdesId = (0:0)
m_ghostRecCnt = 0         m_tornBits = 1

Allocation Status
-----------------
GAM (1:2) = ALLOCATED     SGAM (1:3) = NOT ALLOCATED
PFS (1:1) = 0x63 MIXED_EXT ALLOCATED  95_PCT_FULL   DIFF (1:6) = CHANGED
ML (1:7) = NOT MIN_LOGGED
```

We can see the entry m_pageId = (1:30) telling us that this is page 30 in File ID 1. The entry m_objId = 199305813 tells us what Object ID the page belongs to. OK, we know this but there are occasions when error messages contain page numbers and in that situation the Object ID is very useful.

The m_level and m_indexid fields are meaningful if this page is an index page. The level is the index level where this page resides, and indid tells us the ID of the index to which this page belongs. The field m_freeData is the offset of the start of the free space on the page, and the pminlen field tells us the smallest value a row can be. The entry m_slotCnt tells us how many slots (entries) there are in the row offset table.

Let us now look at the contents of the page. I will omit the page header from the example for clarity:

```
DBCC PAGE ('BankingDB',1,30,1)

DATA:
-----
Slot 0, Offset 0x60
-------------------
```

```
Record Type = PRIMARY_RECORD

Record Attributes =  NULL_BITMAP
1b14c060:  01a80010  00000001  00000001  000003e9 ................
1b14c070:  057e8dbc  00000000  6576654e  766f2072 ..~.....Never ov
1b14c080:  72647265  206e7761  20202020  20202020 erdrawn
1b14c090:  20202020  20202020  20202020  20202020
 :
1b14c1f0:  20202020  20202020  20202020  20202020
1b14c200:  20202020  20202020    000005 ...
 :

Slot 1, Offset 0x20b
--------------------
Record Type = PRIMARY_RECORD

Record Attributes =  NULL_BITMAP
1b14c20b:  01a80010  000186a1  00000001  000003e9 ................
1b14c21b:  03ee6580  00000000  6576654e  766f2072 .e......Never ov
1b14c22b:  72647265  206e7761  20202020  20202020 erdrawn
1b14c23b:  20202020  20202020  20202020  20202020
1b14c24b:  20202020  20202020  20202020  20202020
 :

OFFSET TABLE:
-------------
Row - Offset
15 (0xf) - 6501 (0x1965)
14 (0xe) - 6074 (0x17ba)
13 (0xd) - 5647 (0x160f)

 :
3 (0x3) - 1377 (0x561)
2 (0x2) - 950 (0x3b6)
1 (0x1) - 523 (0x20b)
0 (0x0) - 96 (0x60)
```

We can see, in the DATA section, each row and the offset of the row. We can see, in the OFFSET TABLE section, each entry in the row offset table. Each entry contains a slot number and an offset—for example, the row referenced by slot 0 is at offset 96 (straight after the page header), and the row referenced by slot 15 is at offset 6,501.

4.12 Pages for space management

There are a number of pages resident in a primary or secondary database file that are used to manage space in a file. These special pages are as follows:

- Global Allocation Map (GAM) pages
- Secondary Global Allocation Map (SGAM) pages
- Index Allocation Map (IAM) pages
- Page Free Space (PFS) pages

To understand how GAM and SGAM pages fit into the picture we need to remind ourselves that there are two types of extents in SQL Server 2000. Uniform extents are eight pages in length and are allocated exclusively to one object when it requires space. For example, if a uniform extent is allocated to the Accounts table in the BankingDB database, then only rows from that table can use space on the eight pages.

Mixed extents are eight pages in length also but are allocated one page at a time to many objects when they require space. For example, a mixed extent may provide space for the Accounts table in the BankingDB database plus another seven tables. As we discussed earlier, mixed extents exist to save space, and, as such, the first eight pages of a table (or index) are allocated from mixed extents.

GAM pages hold information concerning which extents are currently allocated—that is, are not free. A single GAM page can manage 64,000 extents, which equates to nearly 4 GB of space. If more than 64,000 extents are present in the file, additional GAM pages are used. A GAM page uses a single bit to represent each extent out of the 64,000 extent range. If the bit is set (1), the extent is free; if it is not set (0), it is allocated.

SGAM pages hold information concerning which extents are currently being used as mixed extents and have one or more unused page—that is, have space that can still be allocated to objects. A single SGAM page can also manage 64,000 extents. If more than 64,000 extents are present in the file, additional SGAM pages are used. An SGAM page uses a single bit to represent each extent out of the 64,000 extent range. If the bit is set (1), the extent is being used as a mixed extent and has at least one unused page; if it is not set (0), it is not being used as a mixed extent, or, alternatively, it

Table 4.5 *GAM and SGAM page settings*

Extent Status	GAM Bit Setting	SGAM Bit Setting
Free, not being used	1	0
Uniform or full mixed extent	0	0
Mixed extent with free pages	0	1

is a mixed extent but all the pages are in use. These settings are shown in Table 4.5.

To find a free extent to allocate as a uniform extent, the GAM is scanned for a bit that is set (1)—that is, an extent not currently in use. The bit is then set to 0 (allocated). To find a mixed extent having at least one free page that can be allocated, SQL Server searches the SGAM for a bit that is set (1). To find a free extent to allocate as a mixed extent, the GAM is scanned for a bit that is set (1)—that is, an extent that is not currently in use. The bit is then set to 0 (allocated). The equivalent bit in the SGAM is set to 1. To free an extent, the GAM bit is set to 1 and the SGAM bit is set to 0.

When allocating extents to a table, SQL Server "round-robins" the allocation from each file if there is more than one file in the filegroup to which the table belongs. This ensures that space is allocated proportionately from each file in the filegroup.

How does SQL Server keep track of which pages belong to a table or index? In previous versions of SQL Server (prior to SQL Server 7.0), data pages in a table were always chained together in a doubly linked list. This behavior changed in SQL Server 7.0 and so in SQL Server 2000 and 2005, this is true only if the table has a clustered index (much more about clustered indexes in Chapter 5).

In SQL Server the extents used by a table or index are managed by IAM pages. A table or index has at least one IAM page, and, if the table or index is spread across more than one file, it will have an IAM page for each file. An IAM page can manage 512,000 pages, and, if the table size exceeds this within a file, another IAM is used. The IAM pages for a file or index are chained together. An IAM page must not only cater to uniform extents allocated to the table or index, but must also cater to single pages allocated from mixed extents.

To do this the first IAM page in the chain of IAM pages holds eight slots which can contain pointers to the eight pages that may be allocated from mixed extents. Other IAM pages in the IAM chain will not hold pointers in these slots. All IAM pages, though, will contain a bitmap with each bit presenting an extent in the range of extents held by the IAM. If the bit is set (1), the extent represented by that bit is allocated to the table or index; if it is not set (0), the extent represented by that bit is not allocated to the table or index.

To find the page ID of the first IAM page for a table or index, use the FirstIAM column in the sysindexes system table (the sysindexes system table will be discussed in Chapter 5). To do this use the following example:

```
SELECT object_name(id) AS Tablename , Name, FirstIAM FROM
sysindexes

Tablename  Name              FirstIAM
---------  ----              --------------
Authors              aunmind  0x7C0000000100
Publishers      UPKCL_pubind  0x650000000100
Titles     UPKCL_titleidind  0x690000000100
```

The Page ID is a hexadecimal number, which can be decoded as described previously in this chapter.

The SQL Server documentation refers to a heap. A heap is a table that does not have a clustered index and, therefore, the pages are not linked by pointers. The IAM pages are the only structures that link the pages in a table together.

Finally, our last special page is a PFS page. A PFS page holds the information that shows whether an individual page has been allocated to table, index, or some other structure. It also documents how free an allocated page is. For each page, the PFS has a bitmap recording whether the page is empty, 1 percent to 50 percent full, 51 percent to 80 percent full, 81 percent to 95 percent full, or 96 percent to 100 percent full. Each PFS page covers an 8,000-page range. When a search is made to look for free space, the PFS page is consulted to see which page in an extent belonging to the table or index may have enough free space.

This results in a fundamental difference between SQL Server 2005 and versions prior to SQL Server 7.0. In these previous versions, if there were no clustered index on the table, new rows were always added at the end—

that is, inserted into the last page. Now, rows can be inserted on any page in the table that has free space.

So, where in the database file do we find these special pages? The first page (0) contains a file header. The second page (1) is the first PFS page. The next PFS page will be found after another 8,000 pages. The third page (2) is the first GAM, and the fourth page (3) is the first SGAM. IAM pages are located in arbitrary positions throughout the file. This is shown in Figure 4.19.

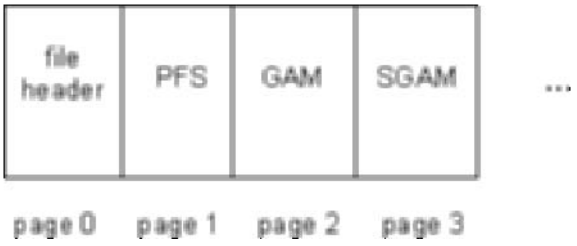


Figure 4.19
The PFS, GAM, and SGAM pages

This chapter has provided an overview of the SQL Server storage structures. In the next chapter we will look at tables and indexes in much more detail. But first of all, now that we have discussed databases, it is time to introduce the BankingDB database used in this book.

4.13 Partitioning tables into physical chunks

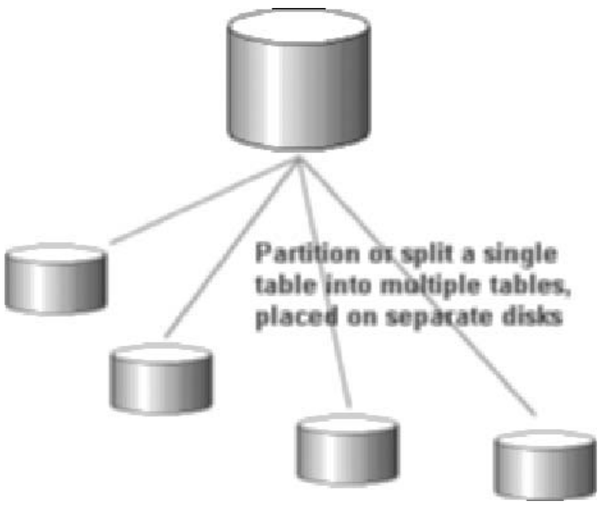
Partitioning involves the physical splitting of large objects, such as tables and indexes, into separate physical parts. Partitioning results in two primary benefits:

- Operations can be performed on individual physical partitions. This leads to a substantial reduction in I/O requirements. For example, you can execute a query on a single partition, or you can also add or remove a single partition. The rest of the table (all other partitions) remains unaffected.
- Multiple partitions can be executed on in parallel. The result is faster processing on multiple CPU platforms.

Both of the above factors make partitioning a tuning method as opposed to something that can be tuned specifically. Any tuning of partitions is essentially related to underlying structures, indexing techniques, and the way in which partitions are constructed.

Figure 4.20 shows a very simple picture of how partitioning can be used to split physical data, containing something like a table, into multiple physical files. Each file is mapped individually to separate partitions within that table. As you can see, physical chunks can even be placed onto separate disk drives. However, splitting to separate disk drives is not necessarily beneficial when using RAID array storage. The reason why is RAID storage already splits physical data into many small pieces.

Figure 4.20 *Physical partitioning*


Partition or split a single table into multiple tables, placed on separate disks

As already stated in Chapter 1, partitioning lets you split large chunks of data into much more manageable smaller physical chunks of disk space. The intention is to reduce I/O activity. For example, let's say you have a table with 10 million records and you only want to read 1 million records to compile an analytical report. If the table is divided into 10 partitions, and your 1 million records are contained in a single partition, then you get to read 1 million records as opposed to 10 million records. On that scale you can get quite a serious difference in I/O activity for a single report.

SQL Server 2005 allows for table partitioning and index partitioning. What this means is that you can create a table as a partitioned table, defining specifically where each physical chunk of the table or index resides.

SQL Server 2000 partitioning was essentially manual partitioning, using multiple tables, distributed across multiple SQL Server computers. Then a view (partition view) was created to overlay those tables across the servers.

In other words, a query required access to a view, which contained a query, not data. SQL Server 2005 table partitions contain real physical records.

Partitioning is much improved in SQL Server 2005 because there is now a physical split between different partitions, rather than just a logical split using a view. An SQL Server 2000 partition view contained a query, against multiple tables, using a UNION statement to merge those multiple underlying tables. Yuk! The effect of partition views is that the partitions you used, the worse performance became—essentially making the use of partitioning pointless with respect to improving performance. In SQL Server 2005, each partition is a separate physical chunk, where indexes are created the same, but also for separate physical chunk. Thus, data plus index partition can be acted on individually without disturbing, or needing to access records in other partitions of the same table.

4.13.1 Types of partitions

SQL Server 2005 has two different variations of partitions:

- **Range partition**: Partition a table based on column in the table where the number of row in each partition can vary. This is because their allocation to each partition is based on the value on the partitioning field.
- **Even distribution partition**: Using partitioning in tandem with filegroups places a partition into a physical disk space area. Splitting data based on quantities of data in partitions. In other words, all partitions should have relatively equal numbers of records, because data is split based on the size of each partition.

4.13.2 Creating a range partition

Create a partition by creating a partition function. Use a CREATE PARTITION FUNCTION metadata statement. This is the syntax:

```
CREATE PARTITION FUNCTION <partition function>
   (<partitioning column>)
   AS RANGE [LEFT | RIGHT]
   FOR VALUES ([<boundary value> [ , … ]])
```

- **Partition function**: name of partition

- **Partitioning column**: This is a data type of field used to create range partition. Generally, only simple data types are allowed, such as integers and fixed length strings. Logically, you really wouldn't want to partition on the kind of data type you would not want to create an index out of. So, things like binary objects, XML data types, and timestamps are not permitted.
- **Boundary value**: Range partition value setting how many partitions are created, field values determining target partitions for each record and index partitions. A boundary value can be a literal value, but also a functional expression value (the result of a function). It cannot be the result of a query, unless of course that query is contained inside a function, which produces an expression. By definition, a function produces a single value (it can be Boolean); a function is thus identical to an expression result.

Partition columns and boundary values must exist in underlying tables and indexes.

LEFT and RIGHT simply determines that a range is specified values occurring to the left or right of each value in the list of boundary values. For example:

```
CREATE PARTITION FUNCTION <partition> (INT)
AS RANGE LEFT (10, 20, 30)
FOR VALUES ( [ <boundary> [ , … ] ] )
```

In the above pseudocoded CREATE PARTITION statement, there are three partitions: <=10, >10, and <=20, >20, and <=30, >30.

4.13.3 Creating an even distribution partition

This type of partition is created using a combination of a CREATE PARTITION FUNCTION statement, followed by a CREATE PARITITION SCHEME statement. The latter will create a mapping between a partitioned table (or a partitioned index), mapping to filegroups. This is the syntax:

```
CREATE PARTITION SCHEME <partition scheme>
AS PARTITION <partition function>
[ALL] TO ({file group | [ PRIMARY ]} [ , ... ] )
```

- **ALL**: All partitions map to file group or PRIMARY (if set).
- **PRIMARY**: Partition stored in the primary filegroup.

The result is a range partition, spliced into an even distribution partition. If there are fewer ranges provided by the partition function than there are filegroups, then there will be empty filegroups that can be used later on.

4.14 The BankingDB database

The BankingDB database is very simple. It consists of just three tables, which are created with the following Transact-SQL syntax:

```
CREATE TABLE customers
(
 customer_no         INT         NOT NULL,
 customer_fname      CHAR(20)    NOT NULL,
 customer_lname      CHAR(20)    NOT NULL,
 customer_notes      CHAR(400)   NOT NULL
)

CREATE TABLE accounts
(
 account_no          INT         NOT NULL,
 customer_no         INT         NOT NULL,
 branch_no           INT         NOT NULL,
 balance             MONEY       NOT NULL,
 account_notes       CHAR(400)   NOT NULL
)

CREATE TABLE branches
(
    branch_no        INT         NOT NULL,
    branch_name      CHAR(60)    NOT NULL,
    branch_address   CHAR(400)   NOT NULL,
    managers_name    CHAR(60)    NOT NULL
)
```

The BankingDB database has customers who have one or many bank accounts. A bank account is managed by a branch of the bank at some geographical location. It is as simple as that.

There are 10,000 bank accounts for 5,000 customers. These are managed by 100 branches. Since we will be creating indexes frequently as we progress through the book, there are no indexes created in the basic database. For the same reason, the tables are also assumed to have no primary key constraints or foreign key constraints.

5 Indexing

There are many bells and whistles that can be tweaked to improve SQL Server performance. Some will provide a more positive benefit than others. To really improve performance, often with dramatic results, the database designer is well advised to concentrate his or her efforts in the area of indexing. The correct choice of index on a table with respect to the WHERE clause in a Transact-SQL statement, so that the query optimizer chooses the most efficient strategy, can have sensational results.

I was once asked to look at a query that performed a complex join and had not completed in over 12 hours. Who knows when the query would have completed had it not been cancelled by the user—it may still have been running at the end of the year! Examination of the query showed that a join condition was missing in the WHERE clause, as was an index on one of the large tables involved in the join.

Making the appropriate changes meant that the query ran in less than eight minutes!

This magnitude of performance improvement is not likely to be achieved every day, but it makes an important point—namely, that focusing effort in the area of indexing and query optimization is likely to produce good results for the effort involved and should be high on the database tuner's hit list.

So, what are these indexes and why are they so important?

5.1 Data retrieval with no indexes

Imagine that this book had no index, and you were asked to find references to the topic page faults. You would have no choice but to open the book at page 1, scan the page looking for the topic, turn to page 2, and continue until you had scanned the last page of the book. You would have to continue your search to the last page in the book, since you would not know when you had found the last reference to the topic. You would have read and scanned every page in the book, which would probably have taken you a considerable length of time.

SQL Server has to behave in a similar fashion when asked to retrieve rows from a table that has no appropriate index. Suppose we were to execute the following Transact-SQL statement against the Accounts table, assuming there was no suitable index present:

```
SELECT * FROM accounts WHERE branch_no = 1100
```

How would SQL Server find the appropriate rows? It would have to search the Accounts table from the start of the table to the end of the table looking for rows that had a branch_no containing the value 1,100. This might be fine for small tables containing just a few rows, but, if the table contained millions of rows, the above query would take a very long time to complete.

What is needed is a fast and efficient way of finding the data that conforms to the query requirements. In the case of a book, there is usually an index section from which the required topic can be found in an alphabetically ordered list, and the page numbers of the pages featuring that topic can then be obtained. The required pages can be directly accessed in the book.

The method used to directly retrieve the required data from a table in SQL Server is not unlike that used with books. Structures called indexes may be created on a table, which enable SQL Server to quickly look up the database pages that hold the supplied key value—in our example the value 1,100 for the branch_no column.

Unlike a book, which normally has one index, a table may have many indexes. These indexes are based on one or more columns in the table. In SQL Server there are two types of index—clustered and non-clustered—which we shall now compare and contrast. The ultimate decision as to whether an index is used or whether a complete scan of the table is performed is made by a component of SQL Server known as the query optimizer, which we will discuss in detail in Chapter 7.

5.2 Clustered indexes

As a database designer you are allowed to create only one clustered index on a table—you have one chance to play this ace and so you must play it carefully. Why only one clustered index per table? Unlike its non-clustered cousin, described shortly, a clustered index imposes a physical ordering of the table data.

Creating a clustered index forces the data rows in the table to be reordered on disk so that they are in the same key sequence order as the clustered index key. For example, if we were to create a clustered index on the customer_lname column of the Customers table, the data rows would be sorted so that their physical order on the disk was in ascending order of the customers' last names—that is, Adamski would precede Tolstoy.

This order would be maintained as long as the clustered index was present. SQL Server would ensure that the insertion of a new data row would cause the row to be placed in the correct physical location in key sequence order.

The structure of a clustered index with its key defined on the customer_lname column of the Customers table is shown in Figure 5.1.

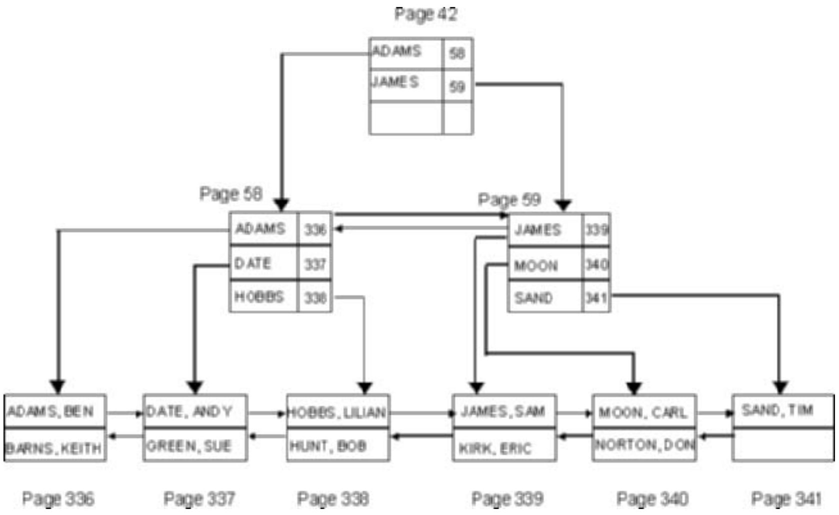


Figure 5.1 *The structure of a clustered index*

The lowest level of the clustered index is composed of the data pages themselves, and in a clustered index the data pages are known as the leaf level of the index. The rest of the clustered index is composed of index pages. The index page at the top of the index is known as the index root. Levels in the index between the root page and the leaf-level pages are known as intermediate-level pages. Another name for an index page is an index node. For simplicity we have shown the structure with the ability to hold two data rows per page and three index entries per page. In reality many more rows and index entries are likely to be found.

At any given level in the index the pages are linked together. This is shown in Figure 5.1, whereas Figure 5.2 emphasizes the linkage. Figure 5.2 shows how index pages are linked together, and this is true regardless of whether the index is a clustered index or non-clustered index.

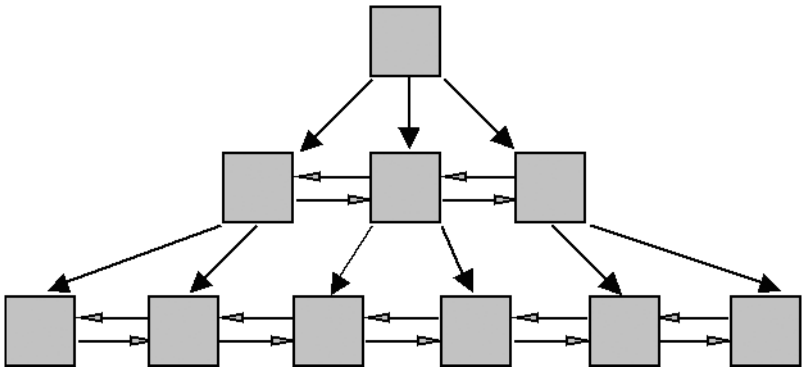


Figure 5.2 *Index pages are linked together*

The entries in the index pages contain a key value and a pointer to the next index page at the next lowest level starting with that key value, plus some control information. The pointer in a clustered index is a page number. In Figure 5.1, for example, the root page has an entry containing a key value, Adams, and a page number, 58, pointing to the intermediate index page 58, whose lowest key value is Adams.

The pointer also contains the File ID as a prefix. This is needed because page numbers are only unique within a database file. A File ID plus a page number is referred to as a Page ID.

The reason why there can be only one clustered index on a table is that the clustered index governs the physical placement of the data, and the data cannot be in two places at once. There can only be one sequence in which the data can be physically placed.

So how can a clustered index support our requirement to perform fast and efficient data retrieval? The clustered index will help us to avoid table scans, since the query optimizer will probably use the clustered index to retrieve data directly. Suppose we issued the following SELECT statement:

```
SELECT * FROM customers WHERE customer_lname =  'Green'
```

Let us assume that the query optimizer decides that the clustered index is the most efficient access path to the data. This is a realistic assumption, since the WHERE clause only specifies the customer_lname column on which the clustered index is based.

SQL Server will first obtain the page number of the root page from the sysindexes table—in our example, page 42. In this root page there will be a number of key values, and in our clustered index these are Adams and James. SQL Server will look for the highest key value not greater than Green, which will be Adams.

In a clustered index an index entry consists of the index key plus a pointer, which is a page number. The pointer held in the Adams key entry points to page 58, and so index page number 58 will be retrieved.

SQL Server will look for the highest key value not greater than Green because page 58 is still an index page. In index page number 58 this is Date. The pointer held in the Date key entry is to page 337, which is a data page, and so this page will be retrieved. The data page is now scanned for a row containing Green in the customer_lname column. The row is found and returned. Note that SQL Server did not know the row existed until the data page was obtained.

Clearly, the clustered index in our example has supported fast access to the data row. If we consider the number of I/Os required to traverse the index in this way, we can see that one I/O is required to retrieve the root page, one I/O is required to retrieve the intermediate index page, and one I/O is required to retrieve the data page—a total of three I/Os. A table scan would probably result in many more I/Os.

Would the three I/Os required to traverse our index be physical reads to the disk? Probably not. The root page of an index is accessed by every query that needs to traverse the index and so is normally always found in cache if the index is accessed frequently. The intermediate nodes and data pages are less likely to be, but if the data cache is large enough it is possible that they will stay in the cache.

We have looked at a SELECT statement that retrieved a single row. What about a SELECT statement that retrieves a range of rows?

```
SELECT * FROM customers
WHERE customer_lname BETWEEN 'Date' AND 'Kirk'
```

In the above example a range of values is specified based on the customer_lname column. It can be seen from Figure 5.1 that because our clustered index is based on the customer_lname column and the data is thus in key sequence order, the rows that meet the criteria are all stored together—that is, clustered. In our example, the six rows that meet the criteria of the SELECT statement are found in three data pages, and so only three I/Os would be required to retrieve these data pages.

If the clustered index had not been based on the customer_lname column, the rows would have not been clustered together (unless fate had intervened or the rows were loaded in that manner with no other clustered indexes on the table).

In the worst case, the six rows would have been stored across six data pages, resulting in six I/Os to retrieve them.

In the BankingDB database there are about 15 customer rows per data page. As an example, eight I/Os would return 120 rows. As we will see, when a clustered index is not present to keep the rows in key sequence order, indexed access of these rows may require 120 I/Os. A not inconsiderable difference!

In a similar manner, clustered indexes support searches using the LIKE operator. Suppose we execute the following query:

```
SELECT * FROM customers WHERE customer_lname LIKE  'N%'
```

All the customers with last names beginning with N will be returned. Again, our clustered index on customer_lname will ensure that these rows are stored together, resulting in the least number of I/Os to retrieve them. Of course, duplicate last names would also be stored in the same cluster of pages.

Finally, what about returning the data in order? Suppose we execute the following query:

```
SELECT * FROM customers ORDER BY customer_lname
```

The query optimizer will know that the clustered index guarantees that the data is in key sequence order, and so there is no need to perform a sort of the rows to satisfy the ORDER BY clause, again saving disk I/O.

5.3 Non-clustered indexes

Similar to their clustered counterparts, non-clustered indexes are balanced trees with a hierarchy of index pages—starting with the index root page at the top, leaf-level pages at the bottom- and intermediate-level pages between the root page and the leaf-level pages. Again, at any given level in the index the pages are linked together, as shown in Figure 5.2.

Data pages in a table without a clustered index will not be chained together, even if non-clustered indexes are present on the table. As was mentioned in Chapter 2, the data pages of the table will only be related through the IAM page(s) managing that table.

Unlike their clustered counterparts, non-clustered indexes have no influence on the physical order of the data, and the leaf level of a sorted index is not considered to be the data but is the lowest level of index pages. The structure of a non-clustered index with its key defined on the customer_fname column of the Customers table is shown in Figure 5.3.

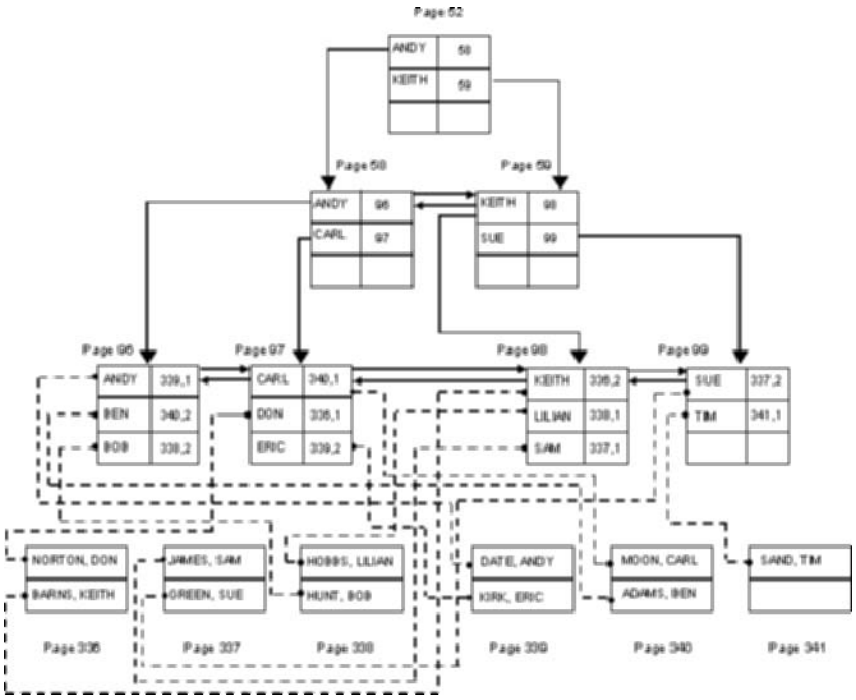


Figure 5.3 *Structure of a non-clustered index when no clustered index on the table*

The first observation we can make is that every data row in the table has a pointer to it from the index leaf level (the dashed lines). This was not the case with the clustered index in Figure 5.1, where the leaf level only contained pointers to the lowest keyed data row in each page. This means that non-clustered indexes are typically larger than their clustered counterparts, because their leaf level has to hold many more pointers. There are about 15 customer rows per data page, so the leaf level of the non-clustered index will need to hold 15 times more pointers than the lowest-level index page in the clustered index. The typical effect of this is that a non-clustered index on a key will usually have one more level of index pages than a clustered index on the same key.

What do the index entries in a non-clustered index look like? Similar to a clustered index, they contain a key value and a pointer to the relevant index page at the next lowest level. This pointer is a Page ID (File ID and database page number). The lowest index level, the leaf level, has index entries also containing a key value and a pointer. While in versions of SQL Server prior to 7.0 the pointer was always a Row ID, which pointed directly at the data row, this is no longer always true.

A Row ID is a Page ID plus a row number. In Figure 5.3 the leaf-level index page 96 has an entry for the key Ben, which points to Page ID 1:340, slot number 2.

So when is a pointer a Row ID and when is it not? If there is no clustered index present on the table, then the pointer is a Row ID. If there is a clustered index present on the table, then the pointer becomes something else. We shall see what this something is shortly and why this is so.

The most important observation to make about Figure 5.3 is that although the index levels are in key sequence order, the data is not. This means that any kind of range retrieval performed using the sorted index will have to use a logical read to follow each relevant leaf-level pointer to the data rows. This is an important point, which we will revisit later. Note also that once the leaf level has been accessed, SQL Server knows whether a row exists or not.

So far we have discussed the behavior of clustered indexes and non-clustered indexes with respect to data retrieval. Let us now look at the behavior of these indexes with respect to data insertion, update, and deletion.

5.4 Online indexes

Simply put, an online index is an index created or altered without affecting any activity against that index. In other words, any queries or change activity using that index will not wait for the creation or rebuilding of a new index, but will rather use existing structures until the new index is fully available. In the past, without generation of indexes with the rest of the database online, any activities against an object in the database, using an index which is rebuilt, would have halted until that index had completed building. For a very large table with millions of billions of records, indexes can take a very long time to create. The way online indexes are created is done using the ONLINE key word in a CREATE INDEX or ALTER INDEX REBUILD statement. It's that easy!

5.5 The more exotic indexing forms

More exotic (new) forms of indexing include parallel created and accessed indexes, partition indexing (parallel indexing is often heavily dependent on partitioning), and native XML data type indexing.

5.5.1 Parallel indexing

The term parallel indexing implies two things: (1) indexes can be created in parallel, and (2) indexes can be read in parallel. Essentially, the result is something called parallel indexing operations, as opposed to just parallel indexing. So, when an index is first created, using a multi-CPU platform, that index can actually be created or rebuilt in parallel, using the multiple CPUs to execute multiple index threads simultaneously. At least that's the theory. And obviously with multi-CPU platforms any index can be read in parallel when different parts of an index structure are required to satisfy a query. Partitioning will more than likely help when executing parallel index operations, only encouraging a greater degree of parallelism. Most importantly, parallel operations in any form are only really beneficial with very large amounts of data and equally large-sized queries. Sometimes parallelism can be of benefit on single CPU platforms but not very often. The most effective use of parallel processing, of any form, in modern relational databases is where table and index partitioning are utilized, given exact physical/logical splits to data stored in a database. RAID arrays do not necessarily internal database parallel (as opposed to sequential) processing, within a database, unless RAID mirrors are divided up beneficial to database internal partitioning. If you want to execute parallel queries, indexes, index creations, and so on—your best bet is to use table and index partitioning.

5.5.2 Partition indexing

A partitioned index is simply an index which is physically divided into separate partitions or sections. Those sections are usually stored on separate areas of disk space but not necessarily. Partitioning can be used to logically divide records in large tables, and underlying RAID array physical splitting, striping, and mirroring can perform physical partitioning in the underlying disk subsystem. In other words, partitioning is implemented at the database level, and RAID storage is executed at the disk subsystem (or operating system) level.

5.5.3 XML data type indexes

The most recent versions of modern relational database engines allow the storage and internal processing of XML documents, much in the same way that a Native XML database does. In other words, an XML document can be acted upon, in the database, as it would in textual form outside of a database using all XML standards such as XSL (eXtensible Style Sheets), and a plethora of other XML standards. Essentially, XML documents can be stored into relational databases, including SQL Server 2005, as a fully functional and executable Native XML database (for each individually stored XML document). Herein lies a problem—relational databases rely heavily on indexing of large chunks of data for rapid access. Essentially, an index creates a search-enhanced structure, occupying much less space than the table itself. Address pointers from index leaf values into table records allow fast access from index to table once an index record has been found. This avoids reading a large table to find small quantities of data. As XML documents become large, it becomes less efficient to use the native XML structure of each document to scan through an entire XML document just to find one little-bitty snippet of data. So, a relational database quite sensibly should allow developers and administrators to create indexes of the contents of XML documents using specific repetitive data items. The resulting index is an index like any other, except that the target table data pointed to is not actually a table but an XML document.

5.6 The role of indexes in insertion and deletion

The existence of indexes on tables is usually considered with respect to query execution time. However, SQL Server indexes, in particular clustered indexes, also affect the behavior of SQL Server when rows are inserted. For non-clustered indexes insertion and deletion change the index. In a clustered index, because the entire table is built as an index, then any DML activity will more likely change the index.

Consider the Customers table shown in Figure 5.4. The table has been allocated four pages from a mixed extent. Three pages are full, and the fourth page is partly filled.

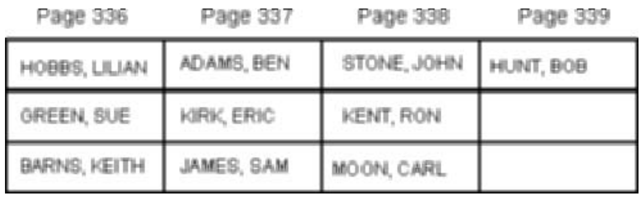


Figure 5.4
Customers table with no indexes present

We will assume, for clarity, that a database page only holds three rows and that only the customer_lname and customer_fname columns are shown.

Suppose we wish to add a new row. Where is it stored? Since the table has no indexes present and there is no free space anywhere else in the pages allocated to the table, the new row is inserted at the end of the table on the last page, as shown in Figure 5.5.

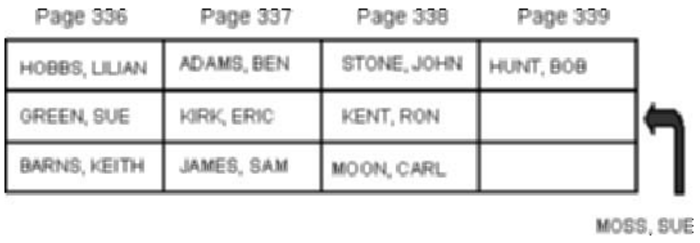
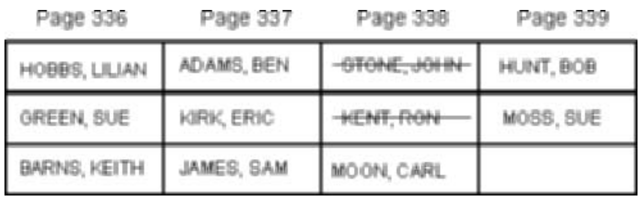


Figure 5.5
Insertion at the end of a table

We shall see shortly that this behavior is true even if there are non-clustered indexes present on the table. Only the creation of a clustered index can modify this behavior.

One can imagine that in a multiuser system many users will be attempting to insert customer rows. In previous versions of SQL Server prior to 7.0 this would have resulted in a hot spot at the end of the table, since a full implementation of row-level locking was not present. However, SQL Server 2000 has a full and robust implementation of row-level locking, and so the hot spot has been virtually eliminated. Locking is discussed in Chapter 9. What happens when rows are deleted from a table?

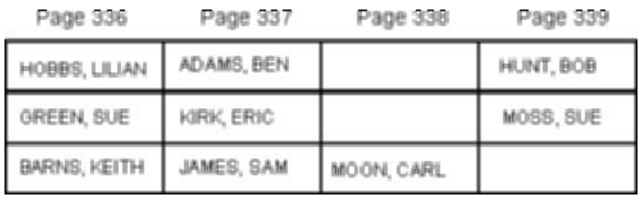
Suppose some rows are now deleted, as shown in Figure 5.6.



Page 336	Page 337	Page 338	Page 339
HOBBS, LILIAN	ADAMS, BEN	~~STONE, JOHN~~	HUNT, BOB
GREEN, SUE	KIRK, ERIC	~~KENT, RON~~	MOSS, SUE
BARNS, KEITH	JAMES, SAM	MOON, CARL	

Figure 5.6
Deleting rows from the table

Free space, shown in Figure 5.7, is left on the pages from which the rows are deleted.



Page 336	Page 337	Page 338	Page 339
HOBBS, LILIAN	ADAMS, BEN		HUNT, BOB
GREEN, SUE	KIRK, ERIC		MOSS, SUE
BARNS, KEITH	JAMES, SAM	MOON, CARL	

Figure 5.7
Space freed as a result of row deletion

If a new row is now inserted, where will it go? In versions of SQL Server prior to 7.0, SQL Server would not have reused the space freed by the deletion of the rows. More sophisticated page management algorithms using more sophisticated page management structures (see Chapter 4) mean that space freed by deleting old rows can be reused by new rows. This is shown in Figure 5.8.

Once all the rows are removed from a page it becomes available for use by the table again. If all the rows are removed from an extent, it may be deallocated and so no longer belongs to the table.

If a row size is used so that only one row can fit on a page, the deletion of a row will mean that there is no remaining row on the page. The page will immediately become available for reuse, and free space will not be wasted.

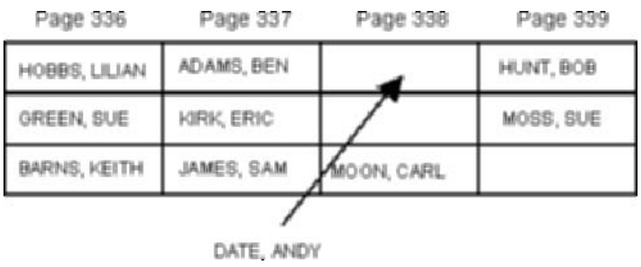


Figure 5.8
Free space being reused

The previous discussion has shown that in a table without any indexes at all, rows will be inserted at the end of the existing data—that is, appended to the rows already present if there is no free space elsewhere in the table. However, if there is free space present in existing database pages in the table because some rows stored earlier have been deleted, then SQL Server can make use of this space to accommodate newly inserted data rows. In Figure 5.9, new rows can be inserted where free space has been left by deleted rows. The PFS management pages hold information about the free space in each page and so can be consulted when a page with sufficient free space is required.

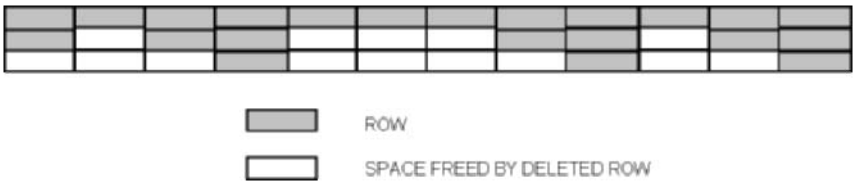


Figure 5.9
Pages with sufficient free space can be used for new rows

This behavior stays the same if non-clustered indexes are present on the table, since they do not govern the physical placement of data. However, a clustered index will modify this behavior. This is because a clustered index will always ensure that new rows are inserted in key sequence order. In our Customers table example, this means in ascending order of the customer's last name. So let's delete some rows and see what happens.

We'll delete the customers who have the last names Green and Hunt. Pages 337 and 338 now have free space in them, as shown in Figure 5.10. Let's now insert two new customers, French and Hood. The clustered index forces these rows to be inserted in key sequence order, so French will need to be inserted after Date but before Hobbs, and Hood will need to be inserted after Hobbs but before James.

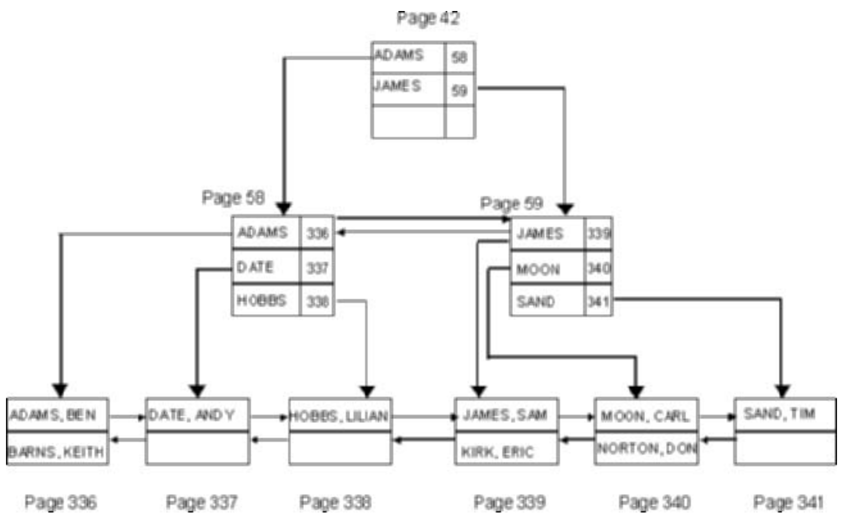


Figure 5.10 *Clustered index with some deleted rows*

Well, we are lucky. It just so happens that there is free space on the pages where we want to insert the rows, and this space is therefore reused, as shown in Figure 5.11.

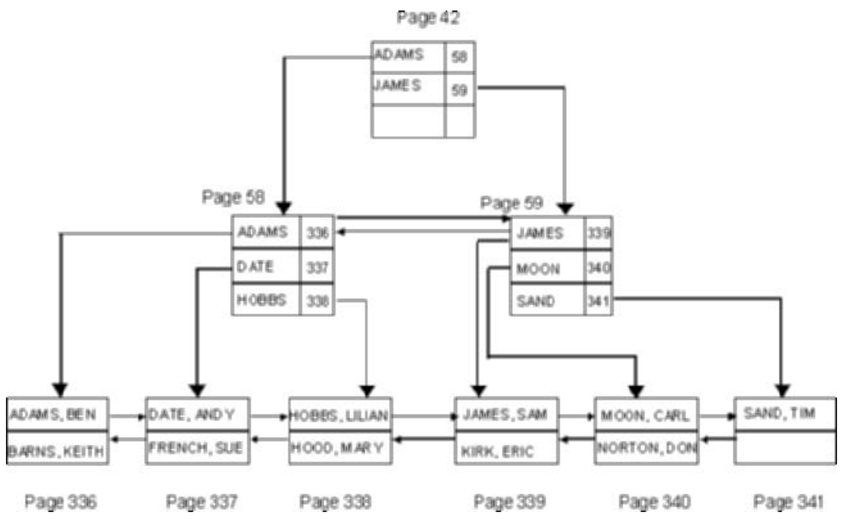
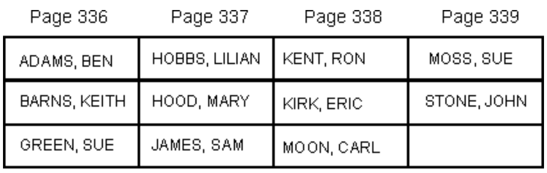


Figure 5.11 *Full clustered index leaf pages*

We can see that in our clustered index, space freed by deleting rows can be reused. Of course, if our clustered index key had been an increasing key value such as that generated in a column with the identity property, new rows would always be inserted at the end of the table and free space in a page may not be efficiently reused.

Our example is, of course, a little contrived, since there will be many occasions where there is not going to be free space in the page where we want to insert the new row, and we will deal with this scenario now.

Suppose that our clustered index contains the entries shown in Figure 5.12. We want to insert a row with a key value of Jones, which SQL Server must store between the key values James and Kent, but there is obviously insufficient space in page 337 to hold the new row. In this case SQL Server must perform a page split. This involves acquiring a new empty page and chaining it into the existing chain of pages.



Page 336	Page 337	Page 338	Page 339
ADAMS, BEN	HOBBS, LILIAN	KENT, RON	MOSS, SUE
BARNS, KEITH	HOOD, MARY	KIRK, ERIC	STONE, JOHN
GREEN, SUE	JAMES, SAM	MOON, CARL	

Figure 5.12
Full clustered index leaf pages

This type of page splitting is known as a 50:50 split, since SQL Server ensures that approximately 50 percent of the rows on the existing page are moved onto the new page, as shown in Figure 5.13. This is only part of the work that SQL Server must do. The intermediate index pages in the clustered index must be updated so that the new page is referenced. This will involve adding a new entry into an index page at the next level up. Of course, if there is insufficient room for the new entry, the index page might split also! In our example, a new entry must be made for the key James pointing to page 202.

What about any non-clustered indexes that point to the table? Previously we mentioned that the index entries at the leaf level of a non-clustered index pointed directly at the data rows and these pointers, known as Row IDs, are of the form Page ID plus a row number on the data page. A Page ID is of the form File ID and database page number. We have just seen that when a page split occurs in a clustered index, rows can migrate from the old page to the newly chained-in page. So does this mean that the Row IDs for these rows are now incorrect? In versions of SQL Server prior to SQL Server 7.0 this is exactly what this would mean. The pointers in any non-clustered indexes present on the table pointing to the rows that had migrated would have to be changed to point to the row locations on the new page. This

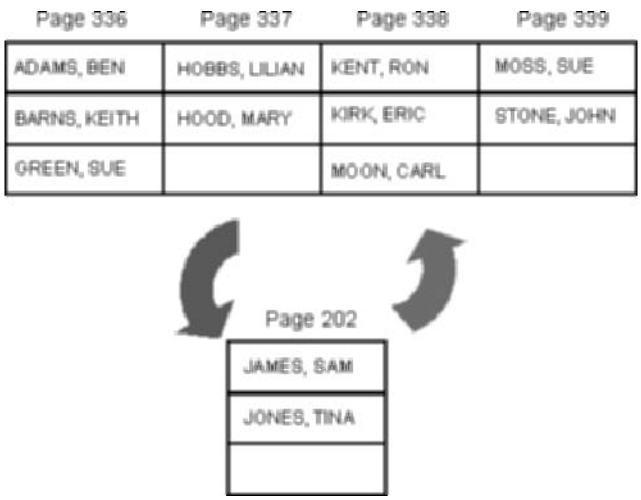


Figure 5.13
A 50:50 page split

would result in a lot of non-clustered index update activity and a consequent increase in lock activity in these non-clustered indexes.

For this reason, as of SQL Server 2000, if a clustered index is present on a table, the non-clustered index pointers are no longer Row IDs. Instead, the non-clustered index pointers are the clustering index key. This is shown in Figure 5.14.

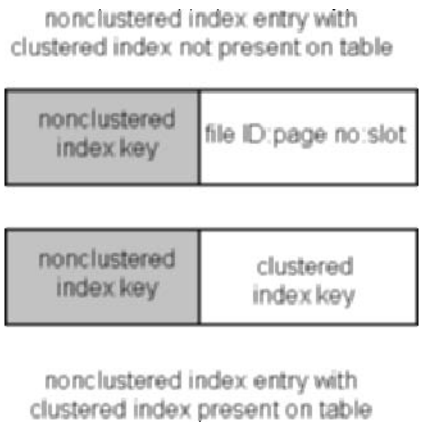

nonclustered index entry with clustered index not present on table
nonclustered index key
file ID:page no:slot
nonclustered index key
clustered index key
nonclustered index entry with clustered index present on table

Figure 5.14
Non-clustered index pointers with and without a clustered index present

This needs a little more discussion! Instead of the index entries at the leaf level of a non-clustered index consisting of the non-clustered index key plus a Row ID pointer, each entry is composed of the non-clustered index key plus the clustered index key. A leaf-level index entry, therefore, no longer points directly at a data row; rather, it takes a route through the clus-

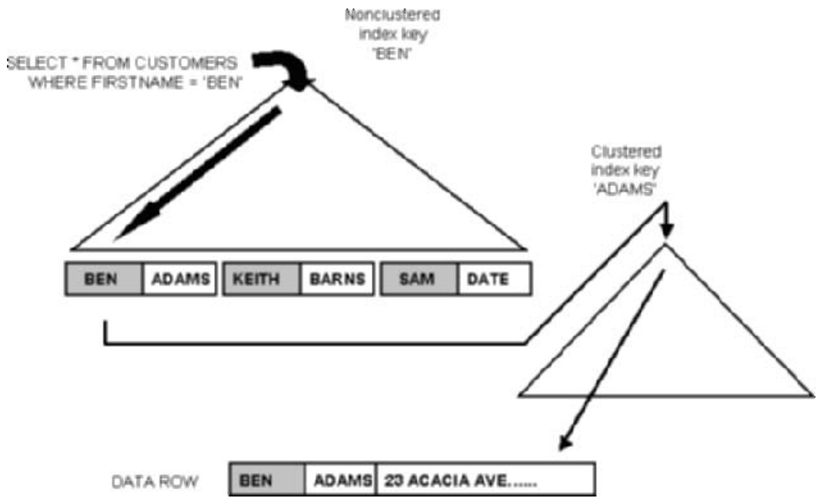

Nonclustered index key 'BEN'
SELECT * FROM CUSTOMERS WHERE FIRSTNAME = 'BEN'
Clustered index key 'ADAMS'
BEN
ADAMS
KEITH
BARNS
SAM
DATE
DATA ROW
BEN
ADAMS
23 ACACIA AVE......

Figure 5.15 *Non-clustered index traversal with a clustered index present*

tered index using the clustering key and then out to the data row. This is shown in Figure 5.15.

The query specifies a column in the non-clustered index on the customer_fname column, and this index is chosen by the query optimizer. The index is traversed until the relevant index entry is found in the leaf-level index page. The pointer in this index entry is the clustered index key for this row. Since the clustered index is defined on the customer_lname column, this pointer is the customer's last name, in this case, Adams. The clustered index is now traversed using this key value, and the data row is fetched.

So, when is a pointer a Row ID and when is it not? If there is no clustered index present on the table, then the pointer is a Row ID. If there is a clustered index present on the table, the non-clustered index pointer (at the leaf level of the index) is the clustered index key. The primary reason for this approach is to avoid the work that must be performed by the server adjusting non-clustered index entries when a data page splits because of insertion into a clustered index, causing data rows to migrate to new pages.

Since the non-clustered index leaf entries do not contain page numbers, if they contain the clustered index key, then the fact that data rows might move to a new page is irrelevant. The pointers will not need to be changed—in other words, they are stable. Because data page splits are a phenomenon only observed when a clustered index is present on a table, it follows that if there is no clustered index present on a table, data page splits cannot occur. The non-clustered index leaf entries are stable with respect to the insertion of new data rows, and the pointers can remain Row IDs, as in versions of SQL Server prior to 7.0.

This is all well and good, but suppose that we issue the following query:

```
SELECT * FROM customers WHERE customer_fname = 'John'
```

If we assume that there is a non-clustered index on the firstname column and a clustered index on the lastname column, then, from what we have just discussed, we can state that the pointer in the non-clustered index will be the clustered index key. Now suppose that for our customer John our pointer is Smith (John's last name). We traverse the non-clustered index searching for a key value of John and find the leaf-level index entry. We will assume for simplicity that there is only one customer with the first name John.

The pointer will contain the clustered index key Smith, and so the clustered index is now traversed from the top searching for this key. If there is only one customer with the last name Smith, we will traverse the clustered index and finally retrieve the data page containing our row. That's fine, but suppose in our Customer table we have more than one customer with the last name Smith. Perhaps we have a customer named Mary Smith. Now we have an interesting scenario. If the clustered index is now traversed from the top searching for a key of Smith, two rows will be found. Clearly this is nonsense, so how does SQL Server find the correct Smith?

The answer can be found in the way that duplicate clustered index key values are handled. If a clustered index is not created as a unique index, then duplicate key values will be allowed in the index. In our example this is not unreasonable—some customers will have the same last name. Internally, SQL Server will, however, add an extra column to the key, known as a uniqueifier. The first instance of a key value will not have a uniqueifier but subsequent instances will. The second instance will have a uniqueifier of 1, the third 2, and so on. In this way, SQL Server internally makes all the key values unique, and it is, in fact, the clustered index key and the uniqueifier that are held as the pointer in a non-clustered leaf-level index pointer. This pointer is then used to traverse the clustered index, and it will return a single, uniquely identified row. The uniqueifier will be completely transparent to the query and the application.

OK, let's now return to where we left off. We had just inserted a customer with the last name Jones, which caused a page split to occur. We might wish to insert another data row with a key value that is close to

Jones. Are the split pages going to split again soon? We can see that if inserts continue, with key values greater than and less than James, there will be a delay before page splitting occurs again. This delay is caused by the fact that the page splitting left us with pages that had free space in them. We can store about 15 Customer rows on a data page, so in reality the page split will leave us with approximately seven rows per page and, therefore, room for another seven or eight rows more per page, which will delay the page splitting.

On average we can expect to find pages that range from 50 percent full having just split to 100 percent full just before they split, giving us an average page fullness of about 75 percent.

This is fine, but suppose the clustered index is based on an ever-increasing key value such as that provided by a column with the identity property or a column containing the date and time an order is taken. Insertion of new rows will always happen at the end of the clustered index. In this case there is no point in SQL Server performing a 50:50 split when a new page is chained in, since space that is reserved physically before the last row inserted will never be used.

Figure 5.16
Insertion at the end of the key range

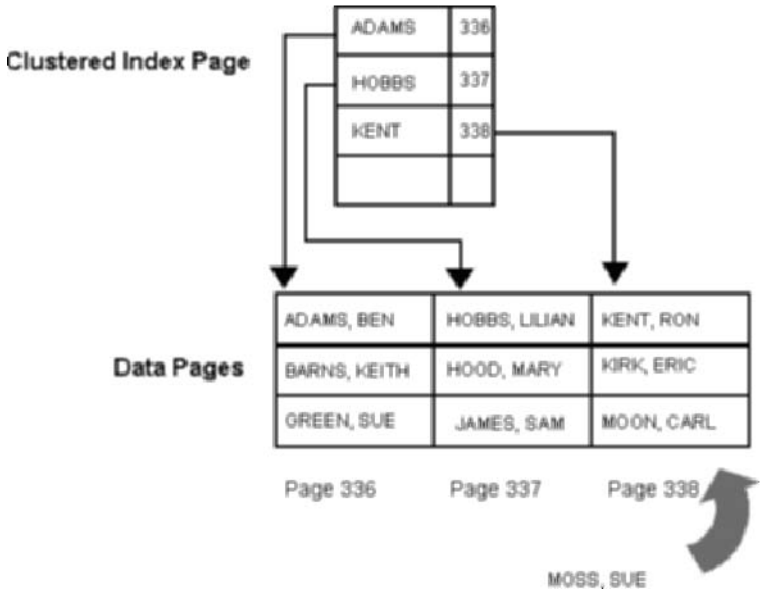


Figure 5.16 shows the insertion of a key value of Moss. There is no space in which to store this row on page 338, so a new page must be chained in. In this case SQL Server does not shuffle rows from page 338 onto the new

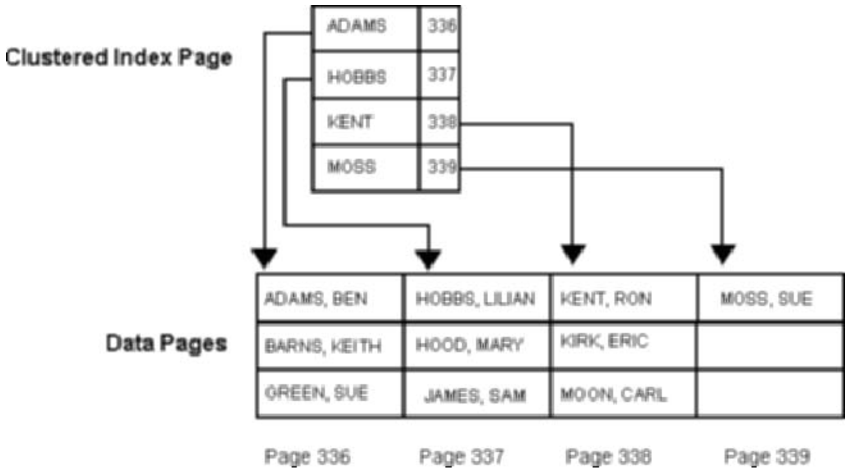


Figure 5.17 *A page split that does not shuffle data*

page but instead inserts only the new row on the new page, as shown in Figure 5.17.

An entry is added into the index page to point to the new key value on the new page.

The action of page splitting when a 50:50 split occurs is clearly going to give SQL Server some work to do. The new page must be obtained and chained in, rows must be shuffled, and entries in many cases will be inserted into a clustered index. Also, of course, new entries will have to be added to the non-clustered indexes to point to the new row.

It would clearly be beneficial to minimize page splitting, but how can we achieve this? One obvious way would be to not use clustered indexes, but the benefits they can bring to the performance of some queries can often outweigh the overhead of page splitting.

Is there another way to minimize page splitting? Fortunately, there is. We can reserve space in a clustered index or a non-clustered index when we create the index using a FILLFACTOR. During the creation of the index the index pages have free space reserved in them and, most importantly in a clustered index, free space is reserved in the data pages.

This free space is only reserved during the index creation process. Once the index has been created, the free space in the index and data pages can be used for newly inserted rows. The size of the index will be larger if space is reserved in it, and in the case of a clustered index the number of data pages in the table will also be greater, but this does mean that the point when SQL Server needs to page split will be delayed.

When SQL Server starts to split pages, fragmentation is said to occur. If many rows are inserted into a clustered index, such that page splits occur, many data pages will be chained into the table, and the table will become fragmented. This affects both insertion and scan efficiency, and so we want to avoid it. We can tell if a table is becoming fragmented by using the DBCC SHOWCONTIG statement.

SHOWCONTIG contiguous implies that two physical chunks of data are contiguous, or close to each other on disk. This means that when searching to find both data chunks, less disk scanning will be performed, thus saving time.

5.7 A note with regard to updates

Obviously, if an indexed column is updated to a new value, the index must also be updated. In the case of a non-clustered index the index entry must change position since index keys are held in key sequence order. In the case of a clustered index, the data row may also have to change position, since the data rows are stored in key sequence order. But what happens to a data row when there is no clustered index present on the table?

Usually the update is performed in-place, which means that the row does not move to another page. Usually an update is logged as a single modification operation in the transaction log. In the case of the table having an update trigger or being replicated, the update is logged as a delete and insert operation. Even in this case the update will usually be an in-place update.

However, there comes a point where a variable-length column is updated to a size greater than its original size and there is no free space available on the page to accommodate it. In this case SQL Server 2000 will delete the row and insert it into a page that has free space. To avoid the overhead of having to adjust index pointers in non-clustered indexes to the new page, a forwarding pointer is left in the original location, which points to the new location. The index pointers will continue to point to the original location. This does mean that a retrieval of the row will incur an extra data page request for the forwarding pointer. If a subsequent update moves the row again, the pointer is adjusted to the new location. If a subsequent update means that the row can return to its original location, it will—and the forwarding pointer will disappear.

To detect the number of forwarding pointers in a table the DBCC SHOWCONTIG statement may be used with the TABLERESULTS option.

Note that a table with a large number of forwarding pointers will experience performance degradation, especially if groups of rows are scanned, due to the extra accesses required. To tidy up the forwarding pointers the clustered index on the table can be rebuilt. If there is no clustered index, if possible create a dummy one and then drop it. Alternatively, unload the data into a file, truncate the table, and reload the data.

5.8 So how do you create indexes?

We have discussed the mechanics of indexes, and later we will discuss indexes with reference to performance, but it is time that we looked at how you create them. Indexes can be created using a Transact-SQL CREATE INDEX statement, or better still, with just the SQL Management Studio.

If you don't like any of the above options, you can always use the SQLDMO (Distributed Management Objects) and the Index object to create an index!

An index is also created when a primary or unique key constraint is added to a table.

First, let us look at the Transact-SQL options, and then we will look at the graphical approach provided by the SQL Manager Studio. We will also have a quick peak at how this may be done in the SQL-DMO.

5.8.1 The Transact-SQL CREATE INDEX statement

The Transact-SQL syntax for a relational table index is as follows:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX
index_name
    ON table | view ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    [ WITH (
         PAD_INDEX  = { ON | OFF }
       | FILLFACTOR = fillfactor
       | SORT_IN_TEMPDB = { ON | OFF }
       | IGNORE_DUP_KEY = { ON | OFF }
       | STATISTICS_NORECOMPUTE = { ON | OFF }
       | DROP_EXISTING = { ON | OFF }
       | ONLINE = { ON | OFF }
       | ALLOW_ROW_LOCKS = { ON | OFF }
       | ALLOW_PAGE_LOCKS = { ON | OFF }
       | MAXDOP = max_degree_of_parallelism
       [ ,...n ] )
    ]
    ]
[ ; ]
```

The following syntax creates an XML index against an XML document:

```
CREATE [ PRIMARY ] XML INDEX index_name
    ON table | view ( xml_column_name )
    [ USING XML INDEX xml_index_name
        [ FOR { VALUE | PATH | PROPERTY } ]
    [ WITH (
         PAD_INDEX  = { ON | OFF }
       | FILLFACTOR = fillfactor
       | SORT_IN_TEMPDB = { ON | OFF }
       | STATISTICS_NORECOMPUTE = { ON | OFF }
       | DROP_EXISTING = { ON | OFF }
       | ALLOW_ROW_LOCKS = { ON | OFF }
       | ALLOW_PAGE_LOCKS = { ON | OFF }
       | MAXDOP = max_degree_of_parallelism
       [ ,...n ] )
    ]
[ ; ]
```

The ONLINE option is not allowed for XML document indexes, because the entire XML document is stored in a single object.

An XML document is ultimately stored in a relational structure as an embedded object, and is thus a field (column) in a table.

The different options will now be described. To create a clustered index in Transact-SQL the CLUSTERED keyword is used:

```
CREATE CLUSTERED INDEX CI_AccountNo
   ON accounts (account_no)
```

The above example creates a clustered index on the account_no column of the Accounts table. The next example creates a unique clustered index, as follows:

```
CREATE UNIQUE CLUSTERED INDEX CI_AccountNo
    ON accounts (account_no)
```

The unique keyword ensures that only one row has a particular key value, in this case account_no. In other words, the uniqueness of the key is enforced. Note that the table may or may not already contain data. If it does, and if there are duplicate values, the above CREATE INDEX statement will fail:

```
CREATE UNIQUE CLUSTERED INDEX CI_AccountNo
    ON accounts (account_no)

Server: Msg 1505, Level 16, State 1, Line 1
CREATE UNIQUE INDEX terminated because a duplicate key
was found. Most significant primary key is '105000'.
The statement has been terminated.
```

Similarly, once the index has been successfully created, an attempt to insert or update a row that would result in a duplicate key value will fail:

```
INSERT INTO accounts (account_no, customer_no, branch_no,
balance, account_notes)
    VALUES (1916, 103424, 1012, 10765, 'A busy account')

Server: Msg 2601, Level 14, State 3, Line 1
Cannot insert duplicate key row in object 'accounts' with
unique index 'CI_AccountNo'.
The statement has been terminated.
```

This is fine, since we want the account_no column to contain no duplicate values, since this is the way we uniquely identify an account.

As mentioned previously, only one clustered index can be created on a table. This makes sense, since data can only be physically sorted in one order. Any attempt to create a second clustered index will fail:

```
CREATE CLUSTERED INDEX CI_AccountBalance
    ON accounts (balance)

Server: Msg 1902, Level 16, State 3, Line 1
Cannot create more than one clustered index on table
'accounts'. Drop the existing clustered index
'CI_AccountNo' before creating another.
```

To create a non-clustered index the CREATE INDEX statement is used, as it was for creating the clustered index, only in this case the NONCLUSTERED keyword is specified:

```
CREATE NONCLUSTERED INDEX NCI_AccountBalance
    ON accounts (balance)
```

If neither CLUSTERED nor NONCLUSTERED is specified, a non-clustered index is created. The UNIQUE keyword has the same effect as it does for a clustered index. Hence, the following CREATE INDEX statement defaults to a non-unique, non-clustered index:

```
CREATE INDEX NCI_AccountBalance ON accounts (balance)
```

The name of the index can be any name considered legal by SQL Server. I prefer to prefix the name with CI_ or NCI_ to signify a clustered or non-clustered index, respectively. I also find it useful to then use meaningful text that indicates the column name. This does, however, become unwieldy when you have an index that is comprised of many columns, so some compromises will have to be made. No naming scheme is ever perfect!

The INCLUDE option allows specification of non-key fields which can be added to a non-clustered index at the leaf level. In other words, this is where the data gets added into the index making what is sometimes called an Index Organized Table or IOT.

IOT means that the entire table, or at least some of the fields in the table, in addition to indexed fields, are included in the index. Some databases call these types of index clusters where index and high usage data fields are clustered together. In some databases, an IOT is exclusively the indexing of all fields in a table—not the case in SQL Server 2005. In SQL Server 2005, a clustered index, a non-clustered index, and IOT are all the same thing.

So, the previous non-clustered index creation statement above could be altered as follows to create a cluster of indexes and some data fields:

```
CREATE INDEX NCI_Cluster_AccountBalance ON accounts
   (account_no, customer_no, branch_no, balance)
```

And this statement creates the entire table as an index or Index Organized Table (IOT):

```
CREATE CLUSTERED INDEX IOT_AccountBalance ON accounts
   (account_no, customer_no, branch_no, balance,
account_notes)
```

So far our examples have shown indexes that consist of only one column. It is not uncommon to create an index that consists of more than one column. Such an index is known as a composite index. An index can be created consisting of no greater than 16 columns, which, in practical terms, is a limit few people are likely to hit. Also, the sum of the column sizes in the index cannot be greater than 900 bytes. It is not a good idea to choose a composite key of 900 bytes in length, because very few index entries will be able to fit into an index page and so many index pages will be used in the index. This will ultimately result in deep indexes consisting of many index levels. Traversing the index may then require many disk I/Os. In SQL Server 2005 it is, in fact, possible to create an index that contains columns defined with variable-length data types, such as VARCHAR, where the sum of the maximum sizes appears to exceed 900 bytes. However, if an attempt is made to insert a row so that the actual size of the index key would exceed the 900-byte limit, an error is returned.

For example, suppose a table consists of the following structure:

```
CREATE TABLE account_details
 (
    account_no  INT           NOT NULL,
    account_notes  VARCHAR(1000)  NOT NULL
 )
```

If we attempt to create a non-clustered index on the account_notes column, SQL Server will successfully create the index but will warn us that the index key is potentially too large:

```
CREATE NONCLUSTERED INDEX NCI_AccountDetails
    ON account_details (account_notes)

Warning! The maximum permissible key length is 900 bytes. The
index 'NCI_AccountDetails' has maximum length of 1000 bytes.
For some combination of large values, the insert/update
operation will fail.
```

If we then attempt to insert a short string into the table, there is no problem:

```
INSERT INTO account_details VALUES (1000, 'This string is
less than 900')
```

However, if we attempt to insert a row with a string value large than 900 bytes, we are not allowed to do so:

```
INSERT INTO account_details
    VALUES (1001, 'This string is more than 900'+
REPLICATE('*',900))

Server: Msg 1946, Level 16, State 4, Line 1
Operation failed. The index entry of length 928 bytes for
the index 'NCI_AccountDetails' exceeds the maximum
permissible length of 900 bytes.
```

How do we specify an option to reserve space in index pages when an index is created? Remember that in the case of a clustered index the data pages are considered to be the lowest level of the index, whereas in the case of a non-clustered index the bottom level of the index is considered to be the lowest level of the index pages. In either case the lowest level of index is known as the leaf level.

The FILLFACTOR option is used to reserve space, and this option takes a value from 0 to 100. An index created with a FILLFACTOR of 100 will have its index pages completely filled. This is useful if no data is to be entered into the table in the future.

An index created with a FILLFACTOR of 0 will have its leaf pages completely filled, but other levels in the index will have enough space for a minimum of another index entry. An index created with a FILLFACTOR of between 0 and 100 will have its leaf pages filled to the FILLFACTOR percentage specified, and, again, other levels in the index will have enough space for a minimum of another index entry.

The default FILLFACTOR value is 0, and this default value can be changed with the sp_configure system stored procedure or via the Database Settings tab in the Server Properties dialog box in the SQL Enterprise Manager. Table 5.1 shows the consequence of different FILLFACTOR values.

Table 5.1 *The effect of differing FILLFACTOR values*

FILLFACTOR Value %	Non-leaf Page	Leaf Page
0	One index entry	Completely full
1-99	One index entry	∑ FILLFACTOR % full
100	Completely full	Completely full

A FILLFACTOR value of 0 percent specifies that the leaf-level page of the index should be completely filled, leaving no free space; however, the non-leaf pages should reserve space for one extra index entry. A FILLFACTOR value of 100 percent specifies that the leaf-level page of the index should be completely filled, leaving no free space. There should also be no free space reserved in the index pages. A FILLFACTOR value of 1 percent to 99 percent specifies that the leaf-level page of the index should be filled no more than the FILLFACTOR value. The non-leaf pages should reserve space for one extra index entry. Note that for non-unique clustered indexes, space is reserved for two index entries.

Care should be taken when choosing a FILLFACTOR, since its relevance will depend on the way the application uses the table data. There is little point in reserving space throughout an index if the row inserted always has a key greater than the current maximum key value. The following example creates an index with a FILLFACTOR of 50 percent, meaning that each data page (leaf page) will only be filled to 50 percent. Index pages at the other levels will have room for one or two more index entries:

```
CREATE CLUSTERED INDEX CI_AccountBalance ON accounts
   (balance) WITH FILLFACTOR = 50
```

SQL Server will round up the number of rows placed on a page, so if the FILLFACTOR value would allow 3 rows, then 4 rows are stored.

Over time, as rows are inserted into the table, the effectiveness of the FILLFACTOR value will vanish, and a planned rebuilding of critical indexes at periodic intervals should be considered if heavy inserts are made to the table. Because SQL Server merges index pages with only one index entry to keep the index compact, the number of items on an index page is never less than two, even if a low value of FILLFACTOR is specified.

Another option, PAD_INDEX on the CREATE INDEX statement, is relevant to reserving space. The PAD_INDEX clause means that the FILLFACTOR setting should be applied to the index pages as well as to the data pages in the index.

The IGNORE_DUP_KEY option is useful when a unique clustered or non-clustered index is to be created on a table that might have rows with duplicate key values inserted. If the IGNORE_DUP_KEY option is set, rows containing duplicate key values are discarded, but the statement will succeed. However, if the IGNORE_DUP_KEY option is not set, the statement as a whole will be aborted.

The DROP_EXISTING option can be a very useful performance optimization. Suppose we have a scenario where we have a table on which we have built a clustered index and perhaps two non-clustered indexes. As discussed earlier, if there is a clustered index present on a table, then the pointers at the leaf level of any non-clustered indexes on that table will be the clustered index key. Suppose we drop the clustered index from the table. The non-clustered index leaf pages can no longer contain index entries that use the clustered index key as the pointer value—there is no clustered index and therefore no clustered index key!

When the clustered index is dropped, SQL Server will rebuild all the non-clustered indexes on that table so that their index leaf pages will now contain index entries that use the Row ID as the pointer value. Remember, a Row ID is a Page ID (File ID plus page number) plus the position of the row on the page. The important point here is that SQL Server will rebuild all the non-clustered indexes on that table. This obviously can be a very time-consuming and resource-intensive process. But this is only the half of it.

Suppose the reason we wished to drop the clustered index was because we wanted to rebuild it. Perhaps we wanted to reorganize it so that page fragmentation was eliminated. Well, this means that after dropping the clustered index we are now going to create it again. Guess what's going to happen to all the non-clustered indexes on that table? You guessed! SQL Server will rebuild all the non-clustered indexes on that table so that their index leaf pages will now contain index entries that use the clustered index key as the pointer value.

This means that our clustered index reorganization has caused our non-clustered indexes to be rebuilt twice. What's annoying is that their leaf-level pointers have ended up as they started out anyway—clustered index key pointers. So what can we do to reduce the impact of rebuilding a clustered index?

Luckily for us the CREATE INDEX statement allows us to specify the DROP_EXISTING option. This allows us to issue a CREATE INDEX statement with the same name as an existing index. Using this option when you wish to rebuild a clustered index will give you a performance boost. The clustered index will be re-created on a new set of database pages, but, because the clustered index key values remain the same, the non-clustered indexes on the table do not have to be rebuilt. In fact, the re-creation of the clustered index can make use of the fact that the data is already sorted in key sequence order so this data does not have to be sorted.

The DROP_EXISTING option can also be used if the clustered index key definition changes. Perhaps a new column is used. In this case the non-clustered index will have to be rebuilt—but only once.

The DROP_EXISTING option can also be used for a non-clustered index, and there will be a performance advantage over dropping and creating the non-clustered index. However, the real benefit is with rebuilding clustered indexes. Using this option will definitely use fewer resources than performing a DROP INDEX followed by a CREATE INDEX.

A CREATE INDEX using this option can also be used to rebuild the index that is created when a primary key constraint is defined on a table. This was previously accomplished with DBCC DBREINDEX. Comparing the resource use of both approaches, they seem identical—so there is probably no need to change existing scripts on this basis alone.

The STATISTICS_NORECOMPUTE option dictates that out-of-date index statistics are not automatically recomputed. This is an option I have never had to use. I have found that ensuring that index key distribution statistics are as up-to-date and accurate as possible is the best approach. Index key distribution statistics are discussed in Chapter 7.

The ON FILEGROUP option allows the database administrator to create the index on a filegroup different from the table itself. The use of filegroups was discussed in Chapter 4. The idea is that by using multiple filegroups, disk I/O to the index and table can be spread across separate disk drives for better performance. However, most database administrators typically use a form of disk striping to spread disk I/O. Disk striping is discussed later on in this book.

Filegroups are also used to facilitate the backing up of large databases. However, if one filegroup contains a table and a separate filegroup contains an index for that table, then both filegroups must be backed up together.

Another index creation option that needs to be discussed is the column [ASC | DESC], which is part of the CREATE INDEX statement. Using these options determines whether an ascending or descending index is created. When an index is created, each column in the index key can be flagged with ASC or DESC. This specifies whether the index column has its data sorted in an ascending or descending manner. The default is ASC, which ensures that scripts written to create indexes in earlier versions of SQL Server behave correctly.

Suppose we create an index on the Accounts table, as in the following example:

```
CREATE NONCLUSTERED INDEX NCI_CustNoAccountNo
    ON accounts (customer_no ASC, account_no DESC)
```

The data in the customer_no key column will be held in ascending order, whereas the data in the account_no key column will be held in descending order. Why bother providing this capability? After all, the doubly linked lists that chain the index pages in an index level together allow SQL Server to rapidly move backward and forward along the sequence of keys. This is true, but if the query requests data to be sorted in the ascending order of one column and the descending order of another column, then just moving along the chain is not going to provide the optimum performance. If, however, the key columns are actually held in a sequence that matches the ORDER BY, then the chain can be followed in one direction and this will provide the optimum performance, so no additional sorting will be required.

The following query will be fully supported by the NCI_CustNo-AccountNo index without an additional sort step:

```
SELECT customer_no, account_no FROM accounts
    WHERE customer_no BETWEEN 1000 AND 1500
    ORDER BY customer_no ASC, account_no DESC
```

The following query will not be fully supported by the NCI_CustNoAccountNo index, and it will need an additional sort step:

```
SELECT customer_no, account_no FROM accounts
    WHERE customer_no BETWEEN 1000 AND 1500
    ORDER BY customer_no ASC, account_no ASC
```

A new metadata function named INDEXKEY_PROPERTY reports whether an index column is stored in ascending or descending order. The sp_helpindex system stored procedure has also been enhanced to report the direction of index key columns.

Finally, the SORT_IN_TEMPDB option can be used to place the data from intermediate sort runs used while creating the index into tempdb. This can result in a performance improvement if tempdb is placed on another disk drive or RAID array. The default behavior, if this option is not used, is to utilize space in the database in which the index is being created. This means that the disk heads are moving back and forth between the data pages and the temporary sort work area, which may degrade performance.

One aspect of index creation that can be seen from the CREATE INDEX syntax diagram is that SQL Server can create indexes on views. This is significant from a performance perspective and therefore is treated separately later in this chapter.

The ALLOW_ROW_LOCKS and ALLOW_PAGE_LOCKS options are both defaulted to on, which allow both individual record locking and page locking (usually a group of records stored within a 2Kb page lock).

Page size can depend on database block size and underlying block for the operating system. These can vary from 2Kb upwards to as much as 64Kb on high systems.

Switching these options off and disabling record or page locking, or both, is ill advised in some environments. In general, OLTP (small transaction sized) systems require record locking. Some data warehouse systems can make do with page locking if large chunks of data are processed in each transaction (usually).

The MAXDOP option specifies a maximum degree (amount) of parallelism to allow for a particular index. The default value is 0 which utilizes parallel processing on all available CPUs, depending on other server activities at the same time. In other words, it tries not to suck up all the processor time and swamp other activities. Setting this option to 1 disables parallel processing regardless of CPU numbers on the platform. Setting to >1 is the same as setting to 0, except that it restricts the number of CPUs used to the number specified in MAXDOP.

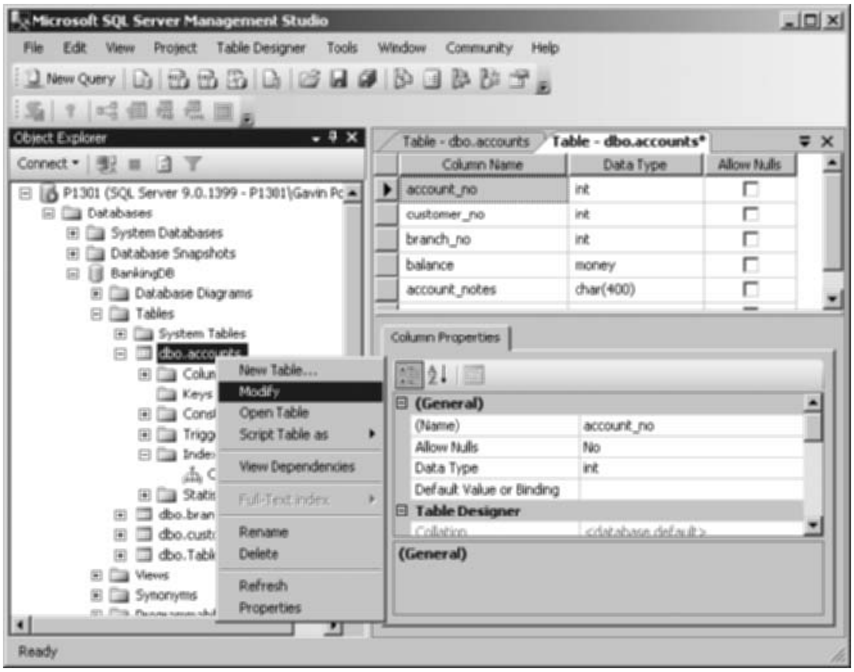
So, we have looked at the CREATE INDEX statement and the options that can be chosen. There are other ways in which we can create indexes and these are discussed in the following sections.

5.8.2 The SQL Management Studio

To create a new index in the SQL Management Studio the following sequence of events can be performed (this is an easy method):

1. Start up the SQL Management Studio and connect to your SQL Server.
2. Open up the Databases folder
3. Open the BankingDB folder. This is a specific database for this book.
4. Open the Tables folder.
5. Right-click one of the tables, and select the Modify option, as shown in Figure 5.18.
6. Next go to the Table Designer option and select the Indexes/Keys…option as shown in Figure 5.19.
7. Then, as shown, in Figure 5.20 various fields can be added to the new index; in this case all the fields in the table.
8. Finally, as shown in Figure 5.21, click the Add button to add or alter the index.
9. There are various other routes in the SQL Server Management Studio that can be used to create a new index and alter existing indexes. For instance, open the Tables folder, modify a table, then open the table folder itself, right-click the Indexes folder, and select the New Index…option from the pop-up menu.

In SQL Server 2000 there were specifically named index tools called Manage Indexes and the Create Index Wizard. These tools are now built into the SQL Server Management Studio for SQL Server 2005.

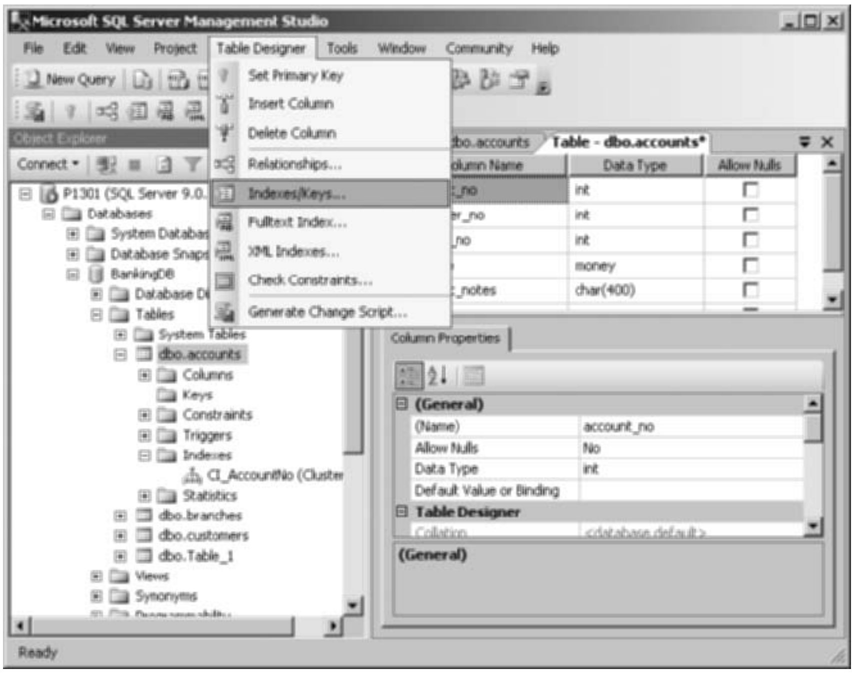


Column Name	Data Type	Allow Nulls
account_no	int	☐
customer_no	int	☐
branch_no	int	☐
balance	money	☐
account_notes	char(400)	☐

(General)	
(Name)	account_no
Allow Nulls	No
Data Type	int
Default Value or Binding	
Table Designer	
Collation	<database default>

(General)

Figure 5.18
Modify a table to create a new index in SQL Management Studio

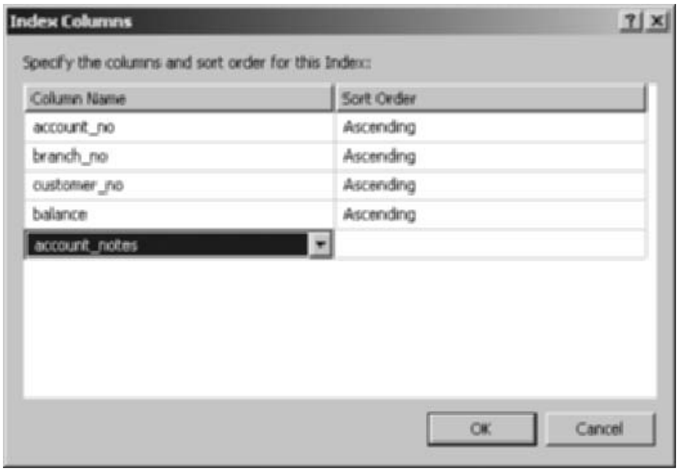


olumn Name	Data Type	Allow Nulls
t_no	int	☐
er_no	int	☐
_no	int	☐
	money	☐
_notes	char(400)	☐

(General)	
(Name)	account_no
Allow Nulls	No
Data Type	int
Default Value or Binding	
Table Designer	
Collation	<database default>

(General)

Figure 5.19
Edit indexes using the index editor under Table Designer

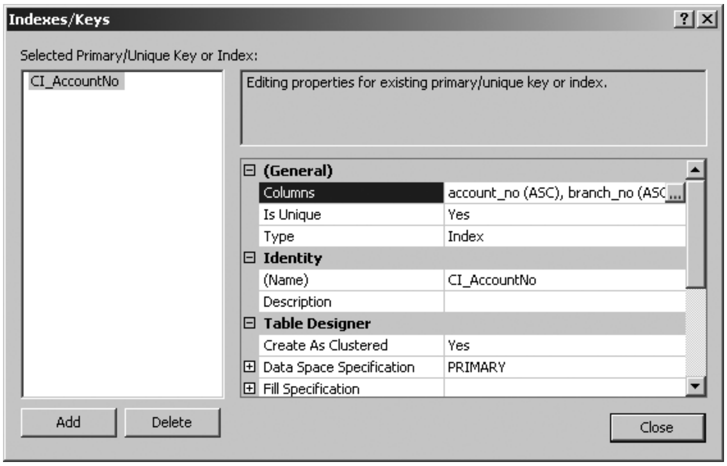


Index Columns

Specify the columns and sort order for this Index:

Column Name	Sort Order
account_no	Ascending
branch_no	Ascending
customer_no	Ascending
balance	Ascending
account_notes	

Figure 5.20 *Add fields to an index*



Indexes/Keys

Selected Primary/Unique Key or Index:

CI_AccountNo

Editing properties for existing primary/unique key or index.

(General)	
Columns	account_no (ASC), branch_no (ASC
Is Unique	Yes
Type	Index
Identity	
(Name)	CI_AccountNo
Description	
Table Designer	
Create As Clustered	Yes
Data Space Specification	PRIMARY
Fill Specification	

Figure 5.21 *Click the Add button to make changes*

5.8.3 The SQL Distributed Management Framework (SQL-DMF)

The SQL Distributed Management Framework (SQL-DMF) is an integrated framework of objects, services, and components that may be used to manage SQL Server. Within the SQL-DMF resides SQL Distributed Management Objects (SQL-DMO). The SQL-DMO is a collection of objects that may be used for SQL Server database management. Index management can be performed through the SQL-DMO. Here is an example of Visual Basic code, which uses the SQL-DMO to create an index:

```
Private Sub cmdCommand1_Click()
On Error GoTo ErrorHandler

    Dim oSQLServer As SQLDMO.SQLServer
    Dim oTblCustomers As SQLDMO.Table
    Dim oIdxCustomerNo As SQLDMO.Index

    Dim bConnected As Boolean
    Set oSQLServer = New SQLDMO.SQLServer
    Set oIdxCustomerNo = New SQLDMO.Index
    Set oTblCustomers = New SQLDMO.Table

    bConnected = False
    oSQLServer.LoginTimeout = 30

    oSQLServer.Connect "KENENG01", "SA", ""
    bConnected = True

    Set oTblCustomers =
        oSQLServer.Databases("BankingDB").Tables("Customers")

    ' Create a new Index object, then populate the object
    ' defining a unique, non-clustered index

    oIdxCustomerNo.Name = "NCI_CustomerNo"
    oIdxCustomerNo.Type = SQLDMOIndex_Unique
    oIdxCustomerNo.IndexedColumns = "[customer_no]"

   ' Create the index by adding the populated Index object
   ' to its containing collection.

    oTblCustomers.Indexes.Add oIdxCustomerNo
    oSQLServer.DisConnect

    Set oSQLServer = Nothing
    Set oTblCustomers = Nothing
    Set oIdxCustomerNo = Nothing

    Exit Sub

ErrorHandler:
    MsgBox (Err.Description)

    If bConnected = True Then
      oSQLServer.DisConnect
      Set oSQLServer = Nothing
      Set oTblCustomers = Nothing
    End If

End Sub
```

5.9 Dropping and renaming indexes

Both clustered and non-clustered indexes can be dropped with the DROP INDEX Transact-SQL statement:

```
DROP INDEX CI_AccountBalance on accounts
```

Note that the table name must also be specified. Indexes can also be dropped by using the graphical interfaces. As discussed previously, if there is a clustered index present on the table, then all the non-clustered indexes will use the clustered index key as a pointer in the leaf-level index pages. Therefore, dropping a clustered index may be a slow, resource-intensive operation, since all the non-clustered indexes will have to be rebuilt. On the other hand, dropping a non-clustered index will be a relatively fast operation, since no other indexes on the table will be affected.

It follows, therefore, that the order in which you drop indexes is important. Drop the non-clustered indexes first, before you drop the clustered index if there is one present on the table. Otherwise, you will waste time rebuilding the non-clustered indexes you are just about to drop.

Indexes can be renamed by using the sp_rename system stored procedure:

```
EXEC sp_rename 'accounts.CI_AccountBalance',
CI_AccountCurrentBalance
```

The use of the single quotes. Indexes may also be renamed by using the graphical interfaces.

You can also drop an index in the SQL Server Management Studio interface.

5.10 Displaying information about indexes

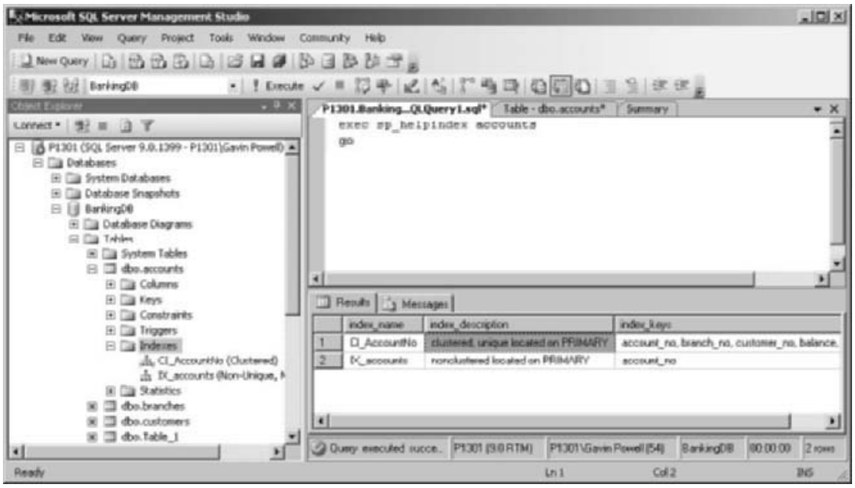
Information can be graphically displayed by using the SQL Server Management Studio as seen in the previous section. There are, however, some other tools worth mentioning.

5.10.1 The system stored procedure sp_helpindex

The indexes that are present on a table can be listed by using the sp_helpindex system stored procedure:

```
EXEC sp_helpindex accounts

index_name index_description index_keys

NCI_CustomerNo non-clustered located on PRIMARY  customer_no
NCI_Balance non-clustered located on PRIMARY  balance
```

The above command executed within a query window of the SQL Server Management Studio is shown Figure 5.22, with the same information.



```
exec sp_helpindex accounts
go
```

	index_name	index_description	index_keys
1	CI_AccountNo	clustered, unique located on PRIMARY	account_no, branch_no, customer_no, balance,
2	IX_accounts	nonclustered located on PRIMARY	account_no

Figure 5.22
The sp_helpindex procedure

5.10.2 The system table sysindexes

The stored procedure sp_helpindex looks in the system table sysindexes, which contains much useful information about indexes. Sysindexes is present in every database. The definition of the table is shown in Table 5.2.

Table 5.2 *Sysindexes table definition*

Column	Datatype	Description
id	int	ID of table (for indid = 0 or 255)—else, the ID of table on which the index is created.
status	smallint	Internal system-status information: 1 = Terminate command if attempt to insert duplicate key. 2 = Unique index. 4 = Terminate command if attempt to insert duplicate row. 16 = Clustered index. 64 = Index allows duplicate rows. 2048 = Index created to support PRIMARY KEY constraint. 4096 = Index created to support UNIQUE constraint.
first	int	If indid = 0 or indid = 1, pointer to first data page. If indid > 1 or ∑ 250, pointer to first leaf page. If indid = 255, pointer to first text or image page.
indid	smallint	Index ID: 0 = Table, 1 = Clustered index, >1 = Non-clustered index, 255 = text or image data.
root	int	If indid > 0 or ∑ 250, pointer to root page. If indid = 0 or indid = 255, pointer to last page.
minlen	smallint	Minimum length of a row.
keycnt	smallint	Number of key columns in the index.
groupid	smallint	ID of the filegroup in which the object is created.
dpages	int	If indid = 0 or indid = 1, dpages is the count of used data pages. If indid > 1 or ∑ 250, dpages is the count of index leaf pages.

Table 5.2 *Sysindexes table definition (continued)*

Column	Datatype	Description
reserved	int	If indid = 0 or indid = 1, the total of pages allocated for all indexes and data pages. If indid > 1 or ∑ 250, the total pages allocated to this index. If indid = 255, the total pages allocated for text or image data.
used	int	If indid = 0 or indid = 1, the total of pages used for all indexes and data pages. If indid > 1 or ∑ 250, the total pages used by this index. If indid = 255, the total pages used for text or image data.
rowcnt	bigint	If indid ∏ 0 and indid ∑ 250, the number of rows in the table—else this is set to 0.
rowmodctr	int	Holds the total number of inserted, deleted, or updated rows since the last time statistics were updated for the table.
xmaxlen	smallint	Maximum size of a row.
maxirow	smallint	Maximum size of a nonleaf index row.
OrigFillFactor	tinyint	The original FILLFACTOR value used when the index was created.
StatVersion	tinyint	Reserved.
reserved2	tinyint	Reserved.
FirstIAM	binary(6)	Page ID of first IAM page for object.
impid	smallint	Reserved.
lockflags	smallint	Used to constrain locking in index.
pgmodctr	int	Reserved.
keys	varbinary (1088)	List of the column IDs of the columns that make up the index key.
name	sysname	Name of table (for indid = 0 or 255)—else index name.
statblob	image	Distribution statistics.

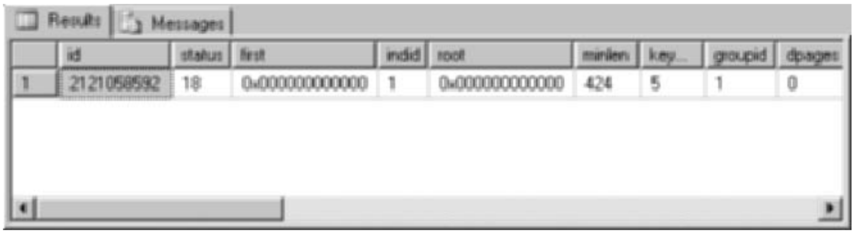
Table 5.2 *Sysindexes table definition (continued)*

Column	Datatype	Description
maxlen	int	Reserved.
rows	int	If indid ∏ 0 and indid ∑ 250. The number of rows in the table—else this is set to 0 (for backward compatibility).

The following example shows a sysindexes entry for the clustered index on the Accounts table. The column headings have been edited and moved for clarity:

```
SELECT * FROM sysindexes WHERE name = 'CI_account'
```

The result of the above query is as shown in Figure 5.23.



```
   id          status  first            indid  root             minlen  key...  groupid  dpages
1  2121058592  18      0x000000000000   1      0x000000000000   424     5       1        0
```

Figure 5.23 *Examining the sysindexes system table in SQL Server Management Studio*

The indid is 1, which shows that this is a clustered index. The number of data pages, dpages, is 0 (there are no records in the table).

5.10.3 Using metadata functions to obtain information about indexes

There are a number of extremely useful functions that can be used to obtain information about the properties of an index. Probably the most useful one is the INDEXPROPERTY function.

This function takes the following form:

```
INDEXPROPERTY(table_ID, index, property)
```

The table_ID holds the object ID of the table (remember that the ID of an object can be obtained by using the object_id function passing the object's name).

The index contains the name of the index whose properties you are investigating.

The property is the property to return and can be one of the values shown in Table 5.3.

An example of the INDEXPROPERTY function is as follows:

```
SELECT INDEXPROPERTY(OBJECT_ID('accounts'), 'NCI_Balance',
'IndexDepth')
```

There are other functions that can also be useful when displaying information about an index. The INDEXKEY_PROPERTY function returns information about an index key—for example, whether a column in the key is sorted in ascending or descending order. Another useful function is the OBJECTPROPERTY function. Some properties specified in this function are concerned with indexing, such as whether a table has a clustered index or not.

Table 5.3 *Property values for the INDEXPROPERTY function*

Value	Description	Value Returned
IndexDepth	Depth of the index.	Number of levels the index has.
IndexFillFactor	Index specifies its own fill factor.	Fill factor used when the index was created or last rebuilt.
IndexID	Index ID of the index on the table or indexed view.	Index ID NULL = Invalid input.
IsAutoStatistics	Index was generated by the auto create statistics option of sp_dboption.	1 = True, 0 = False, NULL = Invalid input.
IsClustered	Index is clustered.	1 = True, 0 = False, NULL = Invalid input.
IsDisabled	Index is disabled.	1 = True, 0 = False, NULL = Input is not valid.

Table 5.3 *Property values for the INDEXPROPERTY function (continued)*

Value	Description	Value Returned
IsFulltextKey	Index is the full-text key for a table.	1 = True, 0 = False, NULL = Invalid input.
IsHypothetical	Index is hypothetical and cannot be used directly as a data access path. Hypothetical indexes hold column-level statistics.	1 = True, 0 = False, NULL = Invalid input.
IsPadIndex	Index specifies space to leave open on each interior node.	1 = True, 0 = False, NULL = Invalid input.
IsPageLockDisallowed	Page locking is disallowed through sp_indexoption.	1 = disallowed, 0 = allowed, NULL = Invalid input.
IsRowLockDisallowed	Row locking is disallowed through sp_indexoption.	1 = disallowed, 0 = allowed, NULL = Invalid input.
IsStatistics	Index was created by the CREATE STATISTICS statement or by the auto create statistics option sp_dboption. Statistics indexes are used as a placeholder for column-level statistics.	1 = True, 0 = False, NULL = Invalid input.
IsUnique	Index is unique.	1 = True, 0 = False, NULL = Invalid input.

5.10.4 The DBCC statement DBCC SHOWCONTIG

This DBCC statement is used to obtain information about an index or table that can be used to investigate performance degradation. It is a very useful tool for performance analysis. Some of the output is a little arcane and is not very useful, but that is more than made up for by the fact that DBCC SHOWCONTIG outputs useful information concerning the level of fragmentation that has occurred in a table—in other words, the level of page splitting. The following DBCC SHOWCONTIG output was from the Accounts table after it had been loaded with 12,500 rows with even values in the account_no column and a clustered index created on the account_no column.

```
DBCC SHOWCONTIG scanning 'accounts' table...
Table: 'accounts' (709577566); index ID: 1, database ID: 7
```

```
TABLE level scan performed.

- Pages Scanned               : 695
- Extents Scanned             : 88
- Extent Switches             : 87
- Avg. Pages per Extent       : 7.9
- Scan Density [Best Count:Actual Count] : 98.86% [87:88]
- Logical Scan Fragmentation             : 12.52%
- Extent Scan Fragmentation              : 0.00%
- Avg. Bytes Free per Page               : 380.2
- Avg. Page Density (full)               : 95.30%
```

The above was taken from a filled database available from the previous edition of this book.

The first line of output, Pages Scanned, is the number of pages in the page chain; in our example, it is the number of pages in the table (dpages in sysindexes). Another way of looking at this item is that it has taken 695 pages to hold the 12,500 rows. Since a page will hold about 18 rows by the time you have taken away the 96-byte page header and other overhead from the 8 Kb page size, this is in the right ballpark.

Extents Scanned is the number of extents read, which means that this is the number of extents used to hold the data rows. Since we have 695 pages, the best we can hope for is (number of pages/8 pages per extent) extents to hold the data. In our case 695/8 is 86.9, and, therefore, the best we can hope for is to hold the data in 87 extents. The data is actually held in 88 extents, slightly over our theoretical calculation but because of the initial allocation in mixed extents, this is reasonable.

Extent Switches is the number of times the DBCC statement moved off an extent while it was scanning the pages in the extent. We would expect an extent switch to happen after the whole extent had been scanned and a new extent needed to be scanned next. Our extent switches value is 87, which is expected, since the jump onto the first extent is not counted.

The Average Pages per Extent is merely the number of pages per extent, which is the (number of pages/number of extents). In our example this is 695/88, which gives us 7.9.

Perhaps the most useful line of output is the Scan Density [Best Count:Actual Count]. This is our measure of fragmentation. The Best Count is the ideal number of extents used to hold our data pages if everything is contiguously linked, whereas the Actual Count is the actual number of extents used to hold our data pages. The Scan Density is the ratio of these two values expressed as a percentage. In other words ([Best Count/ Actual Count] * 100). In our example Scan Density is 87/88 * 100, giving us 98.86 percent, which is close enough to perfect—we are pretty much utilizing our data pages and extents in the most effective way.

The Logical Scan Fragmentation and Extent Scan Fragmentation are not particularly useful, but they do represent the non-contiguity of pages and extents in the index leaf level. The Average Bytes Free per Page and Avg. Page Density (full) are a measure of the average free bytes on the pages in the chain and the percentage of fullness, respectively. These are values that are affected by the FILLFACTOR used.

Next, 12,500 rows with odd values in the account_no column were loaded. This results in page splitting, since the even-numbered rows now have odd-numbered rows inserted between them.

Output after loading 12,500 rows with odd values in the account_no column:

```
DBCC SHOWCONTIG (accounts)

DBCC SHOWCONTIG scanning 'accounts' table...
Table: 'accounts' (709577566); index ID: 1, database ID: 7
TABLE level scan performed.

- Pages Scanned    : 1389
- Extents Scanned  : 176
- Extent Switches  : 1388
- Avg. Pages per Extent  : 7.9
- Scan Density [Best Count:Actual Count] : 12.53% [174:1389]
- Logical Scan Fragmentation  : 50.04%
- Extent Scan Fragmentation   : 1.14%
- Avg. Bytes Free per Page  : 374.6
- Avg. Page Density (full)  : 95.37%
```

The above was taken from a filled database available from the previous edition of this book.

After loading our second batch of 12,500 rows, we can see that the situation has deteriorated. We have doubled the number of rows in the table and the Pages Scanned value is now 1,389, which is double the number of pages scanned previously, 695. The number of extents used to hold the data is now 176, which, again, is not far off from double the number we have just seen, which was 88. The most dramatic increase is in the number of extent switches performed, which is now 1,388—about 16 times greater than the previous value. This gives us a Scan Density of only 12.53 percent.

The bottom line is that there is much page fragmentation. Many pages have been inserted into the original page chain and SQL Server would have to jump around a lot to scan this table. Note also that the page fullness has not changed much. This is often not the case with real-world applications. After page splitting, pages are often found to be between two-thirds and three-quarters full. This is common when page splitting is occurring and is due to the fact that 50:50 splitting is taking place, as mentioned earlier in this chapter. An index rebuild, preferably with an appropriate FILLFACTOR value, would be advisable here.

The full syntax of the DBCC SHOWCONTIG statement is as follows:

```
DBCC SHOWCONTIG
[
( { table_name | table_id | view_name | view_id }
[ , index_name | index_id ] )
]
[ WITH
{ ALL_INDEXES
| FAST [, ALL_INDEXES]
| TABLERESULTS [, {ALL_INDEXES}] [, {FAST | ALL_LEVELS}]
}
]
```

IDs may be used instead of names, if preferred. The index name is optional and if omitted DBCC SHOWCONTIG reports information for the table—unless there is a clustered index on the table, in which case it reports information for that. So, if you want to report on a non-clustered index, it should be named. The option ALL_INDEXES outputs information on all the indexes on the table. The FAST option specifies whether to perform a fast scan of the index and output minimal information. A fast scan does not read the data on each page. The TABLERESULTS option displays results as a rowset and also outputs extra information. Some of this extra information can be very useful. For example, the number of rows referenced by forwarding pointers (as discussed in Chapter 4) is output. By default, information pertaining to a table's data pages (also by convention the clustered index leaf-level pages) or the non-clustered index leaf-level index pages is output. If the ALL_LEVELS option is specified, information pertaining to all index levels is output.

5.11 Creating indexes on views

Unlike previous versions of SQL Server, in SQL Server 2000 indexes can be created on a view, if its definition meets certain criteria. Unlike a non-indexed view, which does not physically hold data, an indexed view has its result physically stored in the database. Any modifications to the base data are reflected in the indexed view, so they are best created on tables that are changed infrequently.

The first index created on a view that is to be indexed must be a unique clustered index. Other indexes may then be created. For a view to be indexed it must satisfy a number of criteria.

One criterion is that it must be created with the SCHEMABINDING option. This option binds the view to the schema of the underlying base tables. This means that any views or tables participating in the view cannot be dropped, unless that view is dropped or changed so that it no longer has schema binding. Also, ALTER TABLE statements on tables that participate in views having schema binding will fail if these statements affect the view definition. Some, but not all, of the other criteria are as follows:

- The view must only use base tables in its definition—no views.
- Any user-defined functions in the view must use the SCHEMA-BINDING option.
- The ANSI_NULLS and QUOTED_IDENTIFIER options must have been set to ON for the connection that defined the view.
- The ANSI_NULLS option must have been set to ON for the connection that defined the tables referenced by the view.
- The base tables referenced in the view must be in the same database and have the same database owner.
- Base tables and user-defined functions referenced in the view must use a two-part name. No other combination of names is allowed.
- All functions referenced by expressions in the view must be deterministic. This means that for a given set of inputs, the same result is always returned.

- The select_list of the SELECT statement in the view must not include the * notation—the columns must be listed explicitly.
- Columns must not appear more than once, unless they appear the second time (or third time, etc.) in a complex expression. The select_list Col1, Col2 is valid and so is Col1, Col2, Col1+Col2 but not Col1, Col2, Col1.
- Also not allowed are derived tables, rowset functions, the UNION operator, subqueries, outer or self joins, the TOP clause, the ORDER BY clause, the DISTINCT keyword, and COUNT(*); however, COUNT_BIG(*) is allowed.
- If the AVG, MAX, MIN, STDEV, STDEVP, VAR, or VARP aggregate functions are specified in queries referencing the indexed view, the optimizer can often calculate the result if the view select_list contains SUM and COUNT_BIG. For example, AVG() can be calculated from SUM() / COUNT_BIG().
- A SUM function that references an expression that can be nullable is not allowed.
- The full-text search predicates CONTAINS or FREETEXT are not allowed.
- The view select_list cannot contain aggregate expressions unless a GROUP BY is present.
- If GROUP BY is present, the view select_list must contain a COUNT_BIG(*) expression, and the view definition cannot include HAVING, CUBE, or ROLLUP.
- A column that results from an expression that either evaluates to a float value or uses float expressions for its evaluation cannot be a key of an index in an indexed view.

We've not finished yet! Indexes created on the view have some restrictions also, as shown in the following list. Most importantly, the first index that is created on the view must be clustered and unique.

- The user executing the CREATE INDEX statement must be the owner of the view.
- The following options must be set to ON for the connection creating the index: CONCAT_NULL_YIELDS_NULL, ANSI_NULLS,

ANSI_PADDING, ANSI_WARNINGS, and ARITHABORT. The QUOTED_IDENTIFIERS and NUMERIC_ROUNDABORT options must be set to OFF.

- Even if the CREATE INDEX statement does not reference them, the view cannot include text, ntext, or image columns.
- If the SELECT statement in the view definition specifies a GROUP BY clause, then the key of the unique clustered index can reference only columns specified in the GROUP BY clause.

An example view definition is as follows:

```
CREATE VIEW dbo.BranchTotalFunds
    WITH SCHEMABINDING
AS
    SELECT branch_no,
    COUNT_BIG(*) AS AccountInstances,
    SUM(balance) AS TotalBalance
    FROM dbo.accounts
    GROUP BY branch_no
```

The following clustered index can now be created:

```
CREATE UNIQUE CLUSTERED INDEX CIV_BranchTotalFunds
    ON dbo.BranchTotalFunds (branch_no)
```

Although the clustered index key will only contain the branch_no column, being a clustered index, the complete set of data rows with all the columns will be stored at the clustered index leaf level in the database. Nonclustered indexes may also now be created on the indexed view if desired.

The query optimizer automatically makes use of indexed views—they do not have to be named explicitly—however, this is only true of the Enterprise Edition. We will discuss this behavior in Chapter 7.

5.12 Creating indexes with computed columns

In SQL Server it is possible to utilize computed columns in an index definition. The definition of the computed column must be deterministic. This means that for a given set of inputs, the same result is always returned.

A computed column definition is deterministic if the following occur:

- All functions referenced in the definition are deterministic and precise.
- All columns referenced in the definition are from the same table as the computed column.
- Multiple rows are not used to provide data for the computed column—for example, using SUM().

FLOAT data types are not precise. Also, various connection options, such as ANSI_NULL, must be set to ON when the table is created, and other options must be set to ON for the connection that creates the index.

As an example, the GETDATE() and @@IDENTITY functions are nondeterministic, whereas SQUARE() and DATEDIFF() are deterministic.

Suppose we create the following table:

```
CREATE TABLE accounts
(
    account_no      INT NOT NULL ,
    customer_no     INT NOT NULL ,
    branch_no       INT NOT NULL ,
    balance         MONEY NOT NULL ,
    account_notes   CHAR (400) NOT NULL ,
    taxed_balance   AS (balance * 0.9)
)
```

The computed column is deterministic, since, for a given input, it produces the same output. Therefore, we can create an index using this column:

```
CREATE INDEX nci_taxed_balance ON accounts (taxed_balance)
```

A SELECT statement that specifies the column in its WHERE clause will use this index if it makes sense to do so:

```
CREATE TABLE accounts
(
    account_no    INT NOT NULL ,
    customer_no   INT NOT NULL ,
    branch_no     INT NOT NULL ,
    balance       MONEY NOT NULL ,
    account_notes CHAR (400) NOT NULL ,
    account_date  AS (GETDATE())
)
```

We could not, however, create an index on the account_date column, since the computed column is nondeterministic.

5.13 Using indexes to retrieve data

Now that we have seen how indexes are put together and how they behave when data is retrieved and added, we can investigate how indexes are used to support good performance.

The choice of whether to use an index or not and if so which index is a decision that the query optimizer makes. We will discuss the query optimizer in detail in Chapter 7, but we need to look at the different mechanisms of using an index to understand what the query optimizer is considering when it is in the process of making its decision.

If there are no indexes present on a table, there is only one mechanism by which the data can be accessed and that is by means of a table scan. When a table scan is performed, each page in the table is read starting at the first page and ending at the last page. To read each page, a page request, SQL Server performs a logical read, also known as a logical I/O. If the page is not found in the data cache, this results in a physical read from disk. Each time a query is run the number of physical reads generated by the query is likely to change, because data will be cached from the previous execution of the query. For this reason, when comparing the work performed by different query optimizer strategies, it is better to compare the logical read values.

The table scan is a useful baseline, since we know that we can always access our data in the number of logical reads the table scan requires. Anything more is likely to be a poor strategy. However, be aware that the query optimizer in SQL Server 2000 considers other factors, such as CPU, when choosing a plan, and so the point at which the query optimizer chooses a table scan in preference to an indexed access is not just the point at which the logical reads used by an index plan exceed the pages in the table, as it was with SQL Server 6.5. With this in mind let us consider different types of index access.

We will use simplified diagrams for our two index types, as shown in Figures 5.24 and 5.25. Figure 5.24 shows a simplified clustered index.

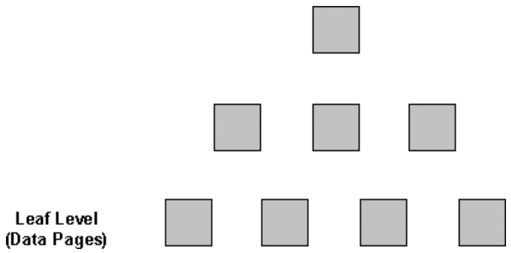


Figure 5.24
A simplified clustered index

Figure 5.25 shows a simplified non-clustered index. Note that, as is commonly found, the clustered index contains one less level than the non-clustered index.

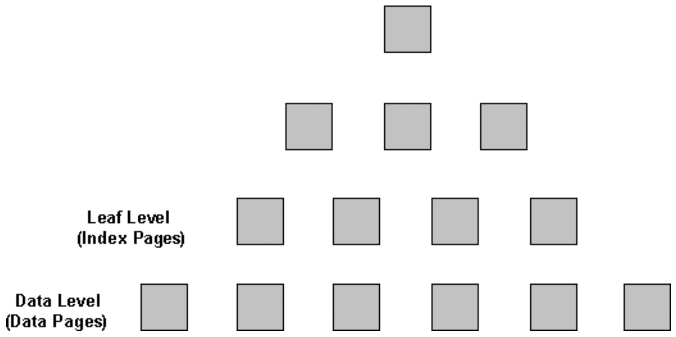


Figure 5.25
A simplified non-clustered index

We will use a number of scenarios. First of all, we will use a scenario where we request a single row from the Accounts table using a clustered index on the account_no column and then a non-clustered index on the account_no column.

Our second scenario will perform a range retrieval from the Accounts table with the same indexing strategy.

Our third scenario will perform an access to the Accounts table that can be satisfied completely by the non-clustered index key columns.

Our fourth scenario will revisit the above scenarios; however, there will still be a non-clustered index on the account_no column of the Accounts table, but we will also add a clustered index on the customer_no column of the Accounts table.

Our fifth scenario will involve the use of multiple non-clustered indexes on our Accounts table.

5.13.1 Retrieving a single row

This is sometimes called a direct key lookup. We are attempting to retrieve a single row as opposed to a range of rows. Often this is a result of using the equality operator (=) on a primary key, for example:

```
SELECT balance FROM accounts WHERE account_no = 4000
```

In the case of the clustered index, SQL Server will first obtain the page number of the root page from the sysindexes table. In this root page there will be a number of key values, and SQL Server will look for the highest key value that is not greater than the key we wish to retrieve.

Remember that with both clustered indexes and non-clustered indexes, the index entries in the index pages are always held in key sequence at a given index level. Refer to Figures 5.1 and 5.3 to clarify this point.

As we have already seen, in a clustered index an index entry consists of the index key plus a pointer, which is a page number (ignoring the fileID), so the index key retrieved in the root page will point to an intermediate index page.

Again, SQL Server will look for the highest key value that is not greater than the key we wish to retrieve. In our diagram, the key found will now contain a page pointer to a data page, and this page will be retrieved. The data page is now scanned for a row containing the key we wish to retrieve. The rows in the data page in a clustered index are in key sequence, so the row is either found and returned or SQL Server will return a message stating "(0 row(s) affected)." This is shown in Figure 5.26.

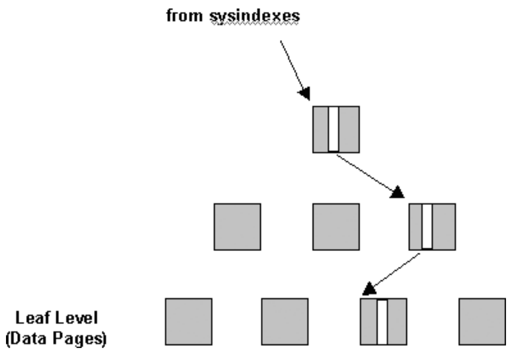


Figure 5.26
A direct key lookup in a clustered index

In the case of a non-clustered index, the traversal of the index is performed in a similar manner. However, once the leaf level is reached the key value of the key we wish to retrieve is found, and this leaf-level index entry will contain the Row ID of the data row, so SQL Server will go directly to it in the appropriate data page, as shown in Figure 5.27.

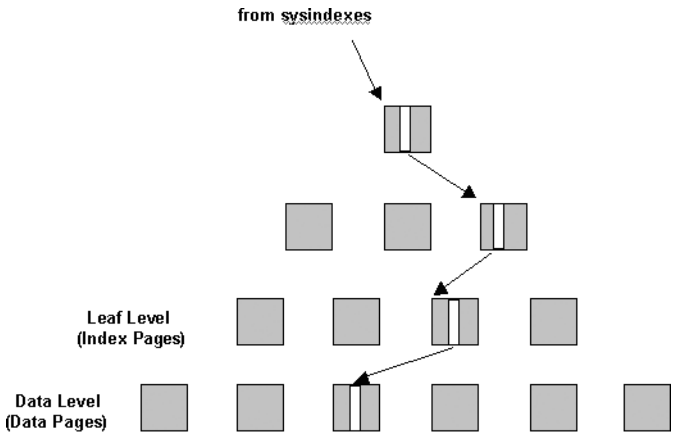


Figure 5.27
A direct key lookup in a non-clustered index

The non-clustered index has taken one more logical read. Is this important? Taken on its own probably not; however, if this is a query we are trying to optimize for an online transaction processing (OLTP) system with a large user population, it might just influence our design. On the whole though, the difference between using a clustered index or a non-clustered index for single row retrieval is slim. Therefore, it can be a wise design choice to save the clustered index specification for columns that are accessed as a range more frequently than they are accessed directly (as a single row hit). Furthermore, as we will see in the next section, retrieving sequential ranges using a clustered index can consume significantly less I/O than when using a non-clustered index because the data pages are stored contiguously. This of course assumes you access the data in the sequence that the index is built in. If you don't access in the index order of the clustered index, you might as well read the entire table.

5.13.2 Retrieving a range of rows

We shall now attempt to retrieve a range of rows, as opposed to a single row. Often this is a result of using operators such as BETWEEN, <, >, and LIKE—for example:

```
SELECT balance FROM accounts WHERE account_no BETWEEN 4001
AND 4500
```

In the case of the clustered index, SQL Server will first obtain the page number of the root page from the sysindexes table. In this root page there will be a number of key values, and SQL Server will look for the highest key value that is not greater than the lowest key we wish to retrieve.

The page pointer will be followed to the intermediate index page.

Again, SQL Server will look for the highest key value that is not greater than the lowest key we wish to retrieve. In Figure 5.28, the key found will now contain a page pointer to a data page, and this page will be retrieved. The data page is now scanned for a row containing the lowest key we wish to retrieve. The row is retrieved and so is the next row and so on until the key value of a retrieved row is found to be higher than the range we require.

This is shown in Figure 5.28 with the query returning three rows. Note that SQL Server is directed to the data page that contains the lowest key value in the range. Once there, SQL Server needs only to retrieve the rows sequentially until the range is exhausted. SQL Server can do this because the clustered index has ensured that the rows are in key sequence order.

In the case of a non-clustered index the traversal of the index is performed in a similar fashion. However, once the leaf level is reached the key value of the key we wish to retrieve is found, and this leaf-level index entry will contain the Row ID of the data row, so SQL Server will go directly to it in the appropriate data page. Now the leaf level of the non-clustered index is in key sequence order but the data is not. What this means is that the key values in the range are found next to one another in the index leaf pages, but it is highly unlikely that the data rows will be found next to one another in the data pages. In Figure 5.29 the query has returned three rows. The leaf

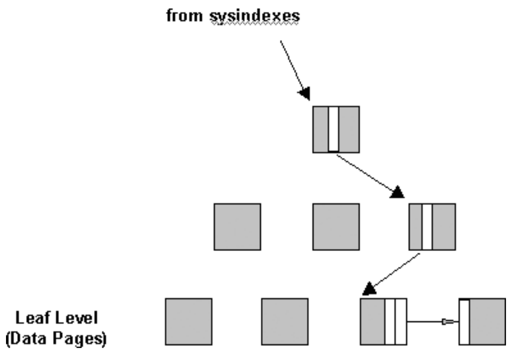


Figure 5.28
A range retrieval in a clustered index

level of the non-clustered index contains the three index entries next to one another, but the data rows are on different data pages.

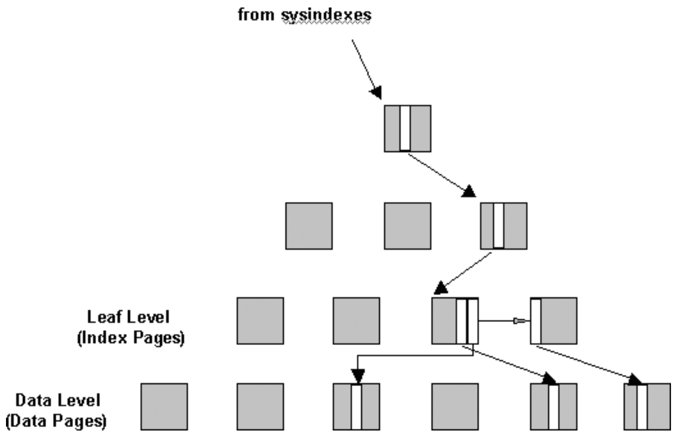


Figure 5.29
A range retrieval in a non-clustered index

This is a very important point and is a fundamental difference between the behavior of a clustered index and a non-clustered index with range retrievals. In our example the clustered index has required less logical reads to retrieve the data than the non-clustered index because in the clustered index the data rows are adjacent.

We have only retrieved three data rows in our example, but suppose we had retrieved 180 data rows. We can hold 18 rows from the Accounts table in one page, so the clustered index could theoretically retrieve the 180 data rows with ten logical reads to the data pages. The non-clustered index will take 180 logical reads to the data pages, which could equate to 180 physical reads if the data rows were all on their own separate data pages and none were found in the data cache (more on data caching in Chapter 8).

Suppose one data page happened to hold ten of the rows that satisfied the range. The non-clustered index would have ten pointers addressing that page and would still generate ten logical reads to it.

If the query optimizer decided that the number of logical reads needed to traverse the non-clustered index, scan the relevant leaf-level pages, and retrieve the data was greater than the number of pages in the table, a table scan would be performed—assuming that other factors such as CPU had been taken into consideration.

5.13.3 Covered queries

The leaf level of a clustered index contains the data rows, whereas the leaf level of a non-clustered index contains only the key and a pointer; as long as the key is only a small portion of the total row we can see that a database page will hold more key values than complete data rows. That is, an index page in the database can hold more index entries than a data page in the database can hold data rows.

We can use this fact to provide fast access for certain queries using a non-clustered index. Suppose we have created a composite index—that is, an index that consists of more than one column. An example of this might be the following:

```
CREATE INDEX NCI_AccountNoBalance
    ON accounts (account_no, balance)
```

Now, suppose we execute the following query:

```
SELECT balance FROM accounts
    WHERE account_no BETWEEN 4001 AND 4500
```

The query optimizer will realize that this is a covered query and that the index named NCI_AccountNoBalance is a covering index. This means that SQL Server does not have to go to the data level to satisfy the query. It only needs to go down as far as the leaf level of the non-clustered index, as shown in Figure 5.30.

This is very efficient. In reality, there are 500 rows satisfying the query, but SQL Server only used four logical reads to satisfy the query. Although clustered indexes are often more efficient than their non-clustered cousins, when a non-clustered index is used as a covering index it is normally more

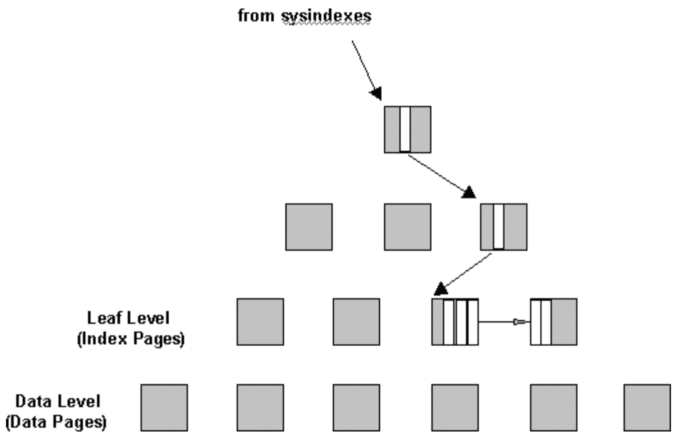


Figure 5.30
A covering index

efficient than an equivalent clustered index. As with a table scan, scanning the leaf level of an index activates the read ahead capability and a parallel data scan is initiated.

5.13.4 Retrieving a single row with a clustered index on the table

The important point to note here is that the non-clustered index on the Accounts table now has its leaf-level index entries containing the clustered index key as a pointer, not the Row ID. This was discussed earlier in the chapter. This means that access to data rows via the non-clustered index will take a route from the non-clustered index leaf-level pointer to the data rows via the clustered index. Let us look at our query again:

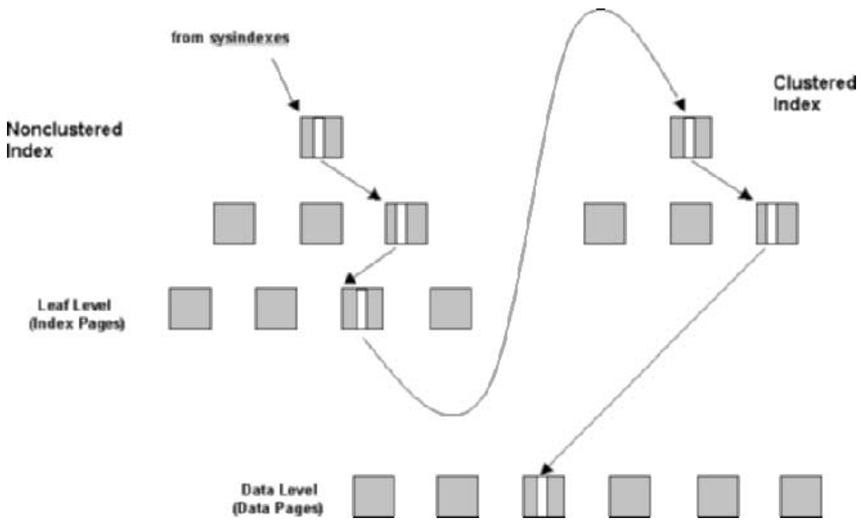
```
SELECT balance FROM accounts WHERE account_no = 4000
```

SQL Server will first obtain the page number of the root page of the non-clustered index on account_no from the sysindexes table. In this root page there will be a number of key values, and SQL Server will look for the highest key value that is not greater than the key we wish to retrieve. As before, the index key retrieved in the root page will point to an intermediate index page.

Again, SQL Server will look for the highest key value that is not greater than the key we wish to retrieve. Having located that, the next-level index page will be retrieved, which will be the leaf-level index page. The leaf-level index entry for account number 4,000 will contain the clustered index key, which will be a customer number.

The root index page of the clustered index will now be retrieved. Using the customer number value to traverse the clustered index, the data row will be retrieved in exactly the same way as any single row retrieval from a clustered index. This is shown in Figure 5.31.

Figure 5.31
A non-clustered index with a clustered index



How does this retrieval strategy compare with our single row retrieval described earlier using a Row ID? Clearly it is less efficient. Instead of following the index pointer directly to the data, we now have to take a trip through the clustered index as well. In reality this is unlikely to be too much of an overhead. A clustered index is a compact index with typically few levels, so we are adding an overhead of a small number of page requests. For a single row retrieval this is not likely to be significant.

5.13.5 Retrieving a range of rows with a clustered index on the table

Again, the basic index retrieval strategy is similar to the range retrieval with a non-clustered index, described earlier. In this case, however, instead of requesting a data page containing the row for each leaf-level index entry found in the range, the clustered index will be accessed to fetch each of the rows in the range. In other words, instead of requesting 180 data pages to fetch our 180 rows, as before, we are now accessing the clustered index 180 times. This is not efficient at all. Again, range retrieval via a non-clustered index is not efficient. Once more than a few rows are returned by the range retrieval, a table scan is likely to be performed by the query optimizer.

5.13.6 Covered queries with a clustered index on the table

This is an interesting scenario. Suppose we wish to execute the following query:

```
SELECT customer_no FROM accounts
    WHERE account_no BETWEEN 4001 AND 4500
```

We will assume that we have a non-clustered index on the account_no column of the Accounts table and a clustered index on the customer_no column of the Accounts table as well.

At first glance, this query does not appear to be covered by the non-clustered index. It is a single column index on account_no. However, we know that the leaf-level pointer is the clustered index key, so the leaf-level index entry contains both the account_no column and the customer_no column. Therefore, the query can indeed be satisfied by the non-clustered index without the data rows being fetched, and the query is, in fact, covered.

The fact that the clustered index key is part of the index entry in a non-clustered index can result in the query optimizer choosing a very efficient strategy.

5.13.7 Retrieving a range of rows with multiple non-clustered indexes on the table

Suppose we wished to execute the following query:

```
SELECT * FROM accounts
WHERE balance BETWEEN 100 AND 200
AND customer_no BETWEEN 1000 AND 1200
```

If there are no appropriate indexes on the table, SQL Server would perform a table scan. If there is a non-clustered index present on the balance column, then the query optimizer might choose to use that index if the number of rows returned was not too large. If there is a non-clustered index present on the customer_no column, then the query optimizer might choose to use that index if the number of rows returned is not too large.

If one of the indexes is present and is chosen, then SQL Server would process the range retrieval by processing the appropriate range in index key values in the leaf level of the non-clustered index and issuing a data page request for each pointer (we'll assume there is no clustered index on the table, so we are dealing with Row IDs). When each data row is fetched, the remaining criteria would be applied to the data row. We say that it is filtered.

One problem with this technique is that it can be wasteful. Suppose we have a non-clustered index present on the balance column alone and that the query optimizer chooses that index to perform the previous query. The index may have 100 leaf-level index key values satisfying the balance range, and 100 data page requests (logical reads) will be performed. SQL Server will then apply the customer number range filter and could eliminate most of the data rows from the resulting set. We have used the non-clustered index to fetch a set of rows, most of which are ultimately discarded. Fetching data pages is a relatively expensive operation.

Now suppose we create a second non-clustered index on the customer_no column. The query optimizer can often make use of both of these indexes in the plan. The result of the query is the set intersection of the set of accounts that have a balance between 100 and 200 and the set of accounts that have a customer number between 1,000 and 1,200. This is shown in Figure 5.32.

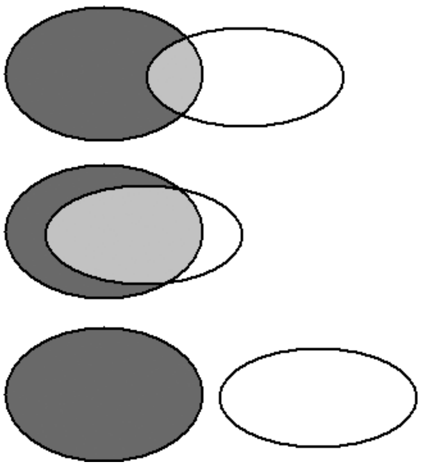


Figure 5.32
Index intersection

From an indexing perspective we can think of this as the set intersection of the valid set of Row IDs from the non-clustered index on balance and the valid set of Row IDs from the non-clustered index on customer_no. As Figure 5.32 shows, the sets of Row IDs may overlap a little, overlap greatly, or not overlap at all. In the latter case, this means that no rows satisfy both criteria. The query optimizer can perform this set intersection in memory (typically) and so find the set of Row IDs that point to data rows satisfying both query conditions before the data pages have been accessed. This will often avoid having many data page requests performed needlessly. How does SQL Server perform the set intersection operation on the Row IDs? It uses a hashing algorithm, which we will discuss in Chapter 7. In Chapter 7 we will also discuss a query optimizer plan, which utilizes index intersection.

So, typically how much benefit can this use of multiple indexes provide? This depends on a number of considerations, but the main one concerns the size of the reduction in the data page requests. Remember: If there are too many, the query optimizer will probably decide a table scan is a more efficient means of querying the data.

If we look at Figure 5.32, we can see that the intersection of the two sets of Row IDs in the second case results in a set that contains most of the Row IDs. In this case the number of data page requests will not be reduced greatly by the use of both indexes.

The intersection of the two sets of Row IDs in the first case results in a set that contains few of the Row IDs. In this case the number of data page requests will be reduced by the use of both indexes and this is a win.

In the third case the two sets of Row IDs do not intersect. This results in a set that contains no Row IDs. In this case the number of data page requests will be reduced to zero by the use of both indexes in the query plan, since clearly no rows satisfy the query. This is a big win.

We have just looked at a variety of scenarios using clustered and non-clustered indexes. In Chapter 7 we will look more closely at the query optimizer itself and how these fundamental scenarios are used.

5.14 Choosing indexes

The choice of indexes can dramatically affect performance and can mean the difference between data being retrieved in seconds, with few disk I/Os or minutes, even hours, with many disk I/Os. Choosing the optimum number of indexes to support the critical queries is therefore an extremely important task.

5.14.1 Why not create many indexes?

If queries can be assisted by indexes, why not create lots of indexes on every table? Unfortunately, as in so many areas of database technology, there are swings and roundabouts concerning the use of indexes. On one hand, indexes can speed up access to data, but, on the other hand, they can slow down table insertions, updates, and deletions. This is because SQL Server has more work to do maintaining all the indexes to ensure that they always truly reflect the current data in the table. Indexes also take up disk space.

Clearly, if disk space is plentiful and the database is read only, there are good reasons to create many indexes. In reality most databases experience a mixture of read and write activity, so the correct choice of indexes is critical to good performance. The choice of appropriate indexes should be a product of good upfront design and transaction analysis.

We have already seen the effect that inserts can have on a clustered index. If the index key is not an increasing key value—that is, the newly inserted key is not always higher than existing key values—data rows will be inserted throughout the page chain. This will cause page splitting to occur.

Either way, row insertion means that SQL Server must perform work to maintain the clustered index. If there are also non-clustered indexes on the table, which is usually the case, each non-clustered index must also be maintained when row insertions occur. Every non-clustered index must accommodate a new index entry, which may cause page splitting to occur in the index pages.

What about row deletion? In a clustered index a row may be deleted from a data page, and, if there is no index entry pointing to it because it is not the lowest key value in the page, little maintenance activity need be performed. In the case of non-clustered indexes there will always be maintenance activity if a row is deleted. Every non-clustered index must remove the index entry. If this leaves a single row in an index page, SQL Server will merge the index page with another in order to keep the index compact. Again, this means work for SQL Server.

The behavior of updates was discussed earlier. It is possible that an update to a row can result in the row being deleted and then reinserted, which has the overhead of deletion and insertion.

The bottom line is that too many indexes on a table can be disastrous for the performance of transactions that write to the table. How many indexes should there be on a table? There is no correct answer, but for a volatile table I start to worry if someone wants to put more than three on it. That's not to say that it will be a problem. I'm just saying I worry, which means I don't leave things to chance—I test them!

5.14.2 Online transaction processing versus decision support

Online transaction processing (OLTP) systems have characteristics that are different from decision support systems (DSSs), and you should have a good appreciation of where your application fits into this spectrum.

OLTP systems tend to involve a high frequency of short, predefined transactions that affect small amounts of data. More often than not, OLTP systems change data by insertion, update, and deletion. OLTP systems frequently support large user populations and provide guaranteed response times in the subsecond range.

DSS systems tend to be read only. They tend to involve a low frequency of long, complex, ad hoc queries that affect large amounts of data. Usually DSS systems do not support large user populations, and the response time of queries may be measured in minutes or even hours. Unlike OLTP systems, DSS systems are often not mission critical. This is shown in Figure 5.33.

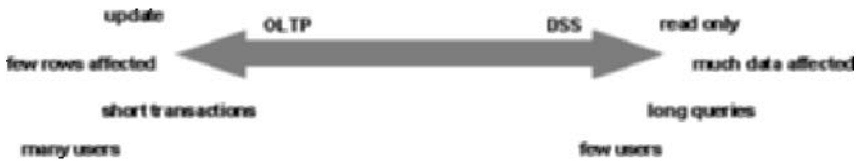


Figure 5.33 *The OLTP DSS spectrum*

Examples of OLTP systems are sales order entry systems and travel booking systems; examples of DSS systems might be anything from MIS reporting systems to large data warehousing systems.

Given the differences in the two application types it is clear that the indexing strategies are going to be different. In the case of OLTP there are likely to be high transaction rates involving transactions that change data. Having too many indexes will adversely affect the performance of OLTP systems, so the designer should limit the number of indexes to those that are really necessary. In the case of DSS the system is likely to be predominantly read only, and therefore the designer can use as many indexes as are needed to support the query mix.

Unlike OLTP transactions, DSS queries are ad-hoc by nature, and the designer will often be unable to perform much upfront transaction analysis in order to arrive at a fixed indexing strategy; therefore, using a good mix of indexes is frequently necessary.

5.14.3 Choosing sensible index columns

When the query optimizer is investigating different access strategies, it will cost each strategy to determine the number of logical reads the strategy will use. This will be an estimate, but, depending on the choice of columns in an index, the query optimizer might decide very quickly that an index is not worth bothering with.

When we are choosing index columns, we should be looking for a combination of columns that support our queries, as well as the number of duplicate values in the index column or columns. Suppose we were to index a column that could contain only the code M (male) and F (female). Would this be a good column to index? It would not be a good column to index, because probably half the rows would contain M and half would contain F. We can say that the following query is not very selective:

```
SELECT * FROM clients WHERE gender = 'F'
```

If there is a non-clustered index on gender, it is highly unlikely that the query optimizer would use it.

Another example would be the state column in a table holding client information. If we executed the following query on a 100,000-row table, how many rows would be returned?

```
SELECT * FROM clients WHERE state = 'CA'
```

If our company is based in San Francisco, we might expect that most of our clients were in California, and therefore 90 percent of the rows in the table might be returned. However, if our company is based in Ulan Bator, we might expect that few of our clients were in California, and therefore 5 percent of the rows in the table might be returned.

We can define selectivity as the percentage of the rows returned. For example:

```
selectivity = (the number of rows returned
  / the count of rows in the table) * 100
```

If 5,000 of the rows in our 100,000-row table were returned, the selectivity of our query would be:

```
selectivity = (5000 / 100000) * 100 = 5%
```

If 90,000 of the rows in our 100,000-row table were returned, the selectivity of our query would be:

```
selectivity = (90000 / 100000) * 100 = 90%
```

The more selective a query the fewer rows returned and the more likely that an index will be chosen by the query optimizer. In the example where 90 percent of the rows in the table are returned, the query optimizer would probably choose a table scan in preference to a non-clustered index on the state column. In the example where 5 percent of the rows in the table are returned, the query optimizer would probably choose to use a non-clustered index on the state column.

The terminology here can be quite confusing. If a query is highly selective, few rows are returned, but the selectivity will be a low percentage value. If a query is not highly selective, many rows are returned, but the selectivity will be a high percentage value.

How does the query optimizer know that 5 percent or 90 percent of the rows in a table will be returned by a query? We shall see later that each index usually has key distribution statistics to help the query optimizer estimate the number of rows returned.

Another value associated with selectivity is density. The density is the average fraction of duplicate index key values in the index. We can easily work out the density by finding the reciprocal of the count of unique values in the index key. Suppose in our example we had clients in 40 states; then the index density would be 1/40 = 0.025.

Once the index density is known, by multiplying the total count of rows in the table by it, we can obtain the likely number of rows hit by specifying a given value, in our example:

```
row hits = 100000 * 0.025 = 2500
```

This is obviously an approximation, since it does not take into account the fact that we might have many or few column values of CA, so index density is only used when key distribution statistics cannot be used.

Again, these terms can be confusing. A high selectivity refers to few duplicates, but a high density refers to many duplicates.

SQL Server holds multiple index densities for a composite index, and we can picture the fact that adding more columns to an index is likely to increase the number of unique values in the index key.

Suppose, in our example, that the index is not based on the state column alone but is based on the state and city columns. Whereas previously 10,000 clients may have been located in California, only ten may be located in Oakland. The selectivity of a query specifying both the state and city columns will be higher than the selectivity of a query specifying only the state column.

SQL Server will hold the index densities for the state column and the state and city columns combined—that is, two density values. The query optimizer can access these values when working out its strategy.

How can we easily find information about the density of an index key? DBCC comes to the rescue with the DBCC SHOW_STATISTICS statement:

```
DBCC SHOW_STATISTICS (accounts, 'NCI_BranchNoCustNo')

Statistics for INDEX 'NCI_BranchNoCustNo'.

                     Updated Rows  Rows Sampled Steps Density   Average key length

Feb 29 2000 11:58AM  10000         10000        295   0.0      11.999647

All density Columns

9.9999998E-3 branch_no
1.9999999E-4 branch_no, customer_no

Statistics for INDEX 'NCI_BranchNoCustNo'.
                    Updated Rows  Rows Sampled Steps Density   Average key length

Oct 19 2000 9:31PM  25000         25000        100   0.0       8.0

All density Average Length Columns

9.9999998E-3 4.0    branch_no
```

```
7.9999998E-5 8.0      branch_no, customer_no

RANGE_HI_KEY RANGE_ROWS EQ_ROWS DISTINCT_RANGE_ROWS AVG_RANGE_ROWS

1000         0.0        250.0   0                   0.0
1001         0.0        250.0   0                   0.0
1002         0.0        250.0   0                   0.0
1003         0.0        250.0   0                   0.0
1004         0.0        250.0   0                   0.0
1005         0.0        250.0   0                   0.0
1006         0.0        250.0   0                   0.0
```

This DBCC statement displays information about the key distribution statistics. Most of this information will be discussed with respect to the query optimizer later in the book. However, there is some information, referred to as All Density, which is the index density we have been discussing. Our index is a composite index of two columns, branch_no and customer_no. The branch_no column has a density value of 9.9999998E3—that is, approximately 0.01. This is representative of the fact that we have 100 unique branch_no values (density = 1/100).

The density of both columns combined is very low (1.9999999E-4 or 0.0002). Suppose there are 10,000 rows in the Accounts table. A query containing the following:

```
WHERE branch_no =  1000
```

would return (10,000 * .01 = 100) rows, whereas a query containing:

```
WHERE branch_no =  1000 AND customer_no = 34667
```

would return (10,000 * 0.0002 = 2) rows.

Let us have a look at another example to emphasize a point. Let us assume that we have a non-clustered index on the balance column in the Accounts table. Here is a fragment of the DBCC SHOW_STATISTICS output:

```
DBCC SHOW_STATISTICS (accounts, NCI_Balance)

Statistics for INDEX 'NCI_Balance'.
```

```
             Updated Rows     Rows   Sampled Steps   Density        Average key length

Oct 19 2000  9:46PM           25000  25000   106     4.0426468E-5  12.0

All density    Average Length   Columns

4.0471085E-5   8.0              balance
3.9999999E-5   12.0             balance, customer_no
```

If we look at the All Density information, we can see that statistics are displayed not only for the balance column but also for the balance, customer_no combination. Why is this? This is a single-column index containing only the balance column. This is because the database administrator has just created a clustered index on the Accounts table on the customer_no column.

Therefore, all non-clustered indexes use this clustered index key as the pointer at the index leaf level. Since the leaf-level index entry for our NCI_Balance index is then effectively balance, customer_no, SQL Server can keep meaningful index density information using both columns. Note that in our previous example the index NCI_BranchNoCustNo would be holding the customer_no column redundantly if there was a clustered index present on the Accounts table on the customer_no column.

This raises an interesting point. If a clustered index is dropped from a table, we know that the non-clustered indexes will be rebuilt so that their leaf-level pointers become Row IDs. This means that they no longer contain the clustered index key, which previously made the non-clustered indexes effectively composite indexes. Therefore, be prepared for some query optimizer strategy changes if you change a clustered index into a non-clustered index at some point.

5.14.4 Choosing a clustered index or a non-clustered index

As we have seen, a table can only have one clustered index, so it is important that we use it carefully—it's our ace, and we want to play it at the right time. So when is a clustered index useful?

Consider using a clustered index when the following occur:

- The physical ordering supports the range retrievals of important queries, or equality returns many duplicates.

- The clustered index key is used in the ORDER BY clause or GROUP BY clause of critical queries.
- The clustered index key is used in important joins to relate the tables—that is, it supports the foreign key.
- The clustered index columns are not changed regularly.

However, remember that there is a downside to using a clustered index. Every non-clustered index leaf-level pointer will become the clustered index key. If the clustered index is large, this may significantly impact the size and efficiency of the non-clustered indexes on the table. Also, creating a clustered index on a large table will require a large amount of free space in the database to accommodate the mechanics of the clustered index creation algorithm. A 1 GB table will require free space equal to 1 GB plus at least 0.2 GB during the creation phase.

Consider using a non-clustered index when the following occur:

- Once or more rows will be retrieved—that is, the query is highly selective.
- The non-clustered index key is used in the ORDER BY clause or GROUP BY clause of critical queries.
- The non-clustered index key is used in important joins to relate the tables.
- A covered query is required.
- Multiple indexes can be used for index intersection.

Also consider that many applications will require the selection of a row by its primary key. This is a single-row selection and therefore would normally benefit from the creation of an index containing the same columns as the primary key. Since it is less common to request ranges of primary keys, a non-clustered index is probably the best option.

There are occasions when neither a clustered index nor a non-clustered index should be used. If the table is small the query optimizer will probably choose a table scan anyway, and if the index has a low selectivity, the query optimizer might ignore it.

Creating an index in these instances just increases disk space use and maintenance overhead. The choice of index and index columns is often a compromise, in my experience, regardless of the database product. This choice is perhaps the most critical one the database designer must face, since incorrect indexes will result in potentially greater disk I/O, CPU, locking contention, and a lower caching efficiency. A piece of good news, though, as we shall see later in this book, is that SQL Server possesses an Index Tuning Wizard, which can assist us when designing our indexing strategy.

6

Basic Query Tuning

There are some very basic guidelines for writing efficient SQL code. These guidelines largely constitute nothing more than writing queries in the proper way. In fact, it might be quite surprising to learn that as you work with a relational database and applications, one of the most common causes of performance problems, can usually be tracked down to poorly coded queries.

This chapter will discuss in general terms what in SQL statements is good for performance, and what is not. The approach to performance in this chapter will be one of focusing purely on the SQL coding itself. This approach is somewhat database independent as well, in that it could apply to any relational database vendor product. So, this chapter focuses on tuning SQL code statements and ignores performance factors, such as I/O, cache, and configuration parameters. It is essential to understand the basic facts about how to write well-performing SQL code first, without considering specific details of database structure and hardware configuration.

The most important rule of thumb with SQL statements, and particularly SELECT statements, those most subject to tuning, is what I like to call the *KISS* rule, or the *Keep It Simple Stupid* rule. The simpler your SQL statements are, then the faster they will execute. There are two reasons for this:

1. Simple SQL statements are much more easily tuned because there is less code to consider when programming it. In other words, it is also easier for a person to write simple code, even a very intelligent person.
2. The optimizer will function a lot better when accessing less complex SQL code, incidentally, also because there is less code to con-

sider—in this case, for the optimizer. The optimizer has a limit to its capabilities.

The negative effect of keeping things simple is granularity but this negative effect depends on how applications are coded. For instance, connecting to and disconnecting from the database for every SQL code statement is extremely inefficient. That's far too much granularity.

By no means will all the queries in this chapter be executed always as stated in this text. There are many factors that will influence the way a query will execute at any given time. All this chapter will do is describe some rules as to how to approach building queries.

6.1 The SELECT statement

It is always faster to SELECT exact column names:

```
SELECT country_id, country, population, area FROM country
```

is faster than (even if all fields were selected above):

```
SELECT * FROM country
```

And if there is a primary key index on the country table then this query may read only the index, ignoring the table altogether:

```
SELECT country_id FROM country
```

The index contains only a single column and the table contains nine columns; reading the index is faster because there is less physical space to traverse.

Using composite indexes can become quite complicated because sometimes the index will be read, and sometimes not. The country table could have an index on the country and region id columns. This query reads only the index:

```
SELECT region_id, country_id FROM country
```

In some cases, the optimizer will read the index only for the query above. However, in other cases, the optimizer can use an index when part of the composite index is missing (even the prefix field). So, this can use the index:

```
SELECT * FROM country WHERE region_id BETWEEN 1 AND 10
```

So, can this:

```
SELECT * FROM country WHERE country_id BETWEEN 5 AND 100
```

In reality, sometimes the optimizer will completely ignore an index, even if the index and query structure match properly. For example, where a table is very small, it can sometimes be more efficient to read the entire table (ignoring the index), rather than reading both index and table. Also, when reading over say 5 to 10 percent of a large table, then reading the entire table can be more efficient than reading both index and table.

6.1.1 Filtering with the WHERE clause

Filtering the results of a SELECT statement using a WHERE clause implies retrieving only a subset of rows from a larger set of rows. The WHERE clause can be used to either include wanted rows, exclude unwanted rows, or both.

In the following query we filter rows to include only those rows we want, thus retrieving only those rows we want:

```
SELECT * FROM country WHERE country LIKE 'U%'
```

Now we do the opposite and filter out rows we do not want:

```
SELECT * FROM country WHERE country NOT LIKE 'U%'
```

The above query reads the entire table, ignoring any indexing, simply because it finds what is not there. The only option for a negative search is a full table scan.

The fastest possible WHERE clause is reading a single record, utilizing a unique key, preferably an integer valued key.

So far we have looked at WHERE clauses containing single comparison conditions. In tables where multiple column indexes exist there are other factors to consider. As with the ordering of index columns in the SELECT statement, the same index should be used for both queries regardless of the sequencing of the columns in each query—but not always:

```
SELECT * FROM country WHERE region_id = 6 AND country_id = 92
SELECT * FROM country WHERE country_id = 92 AND region_id = 6
```

In addition, the above queries could force the use of only one column in the composite index and either the first (prefix) or second columns of the index alone.

Try to always do two things with WHERE clauses:

1. Try to match comparison condition column sequence with existing index column sequences (although it is not strictly necessary).
2. Always try to use unique, single-column indexes wherever possible. A single-column unique index is much more likely to produce exact hits. An exact hit is the fastest access method.

6.1.2 Sorting with the ORDER BY clause

The ORDER BY clause sorts the results of a query. The ORDER BY clause is always applied after all other clauses are applied, such as the WHERE and GROUP BY clauses.

The GROUP BY clause will be covered shortly.

Without an ORDER BY clause in a query, rows will often be retrieved in the physical order in which they were added to the table. Also, rows are not always appended to the end of a table as space can be reused. Therefore, physical row order is often useless. Additionally, the sequence and content of columns in the SELECT statement, WHERE, and GROUP BY clauses can help to determine returned sort order to a certain extent.

In the following example, we are sorting based on the content of the primary key index. There is no use of the index because the entire table is being read:

```
SELECT country_id, country FROM country ORDER BY country_id
```

In the next example, the country column is removed from the SELECT statement and thus the primary key index is used. Selecting only the country_id column forces use of the index. The ORDER BY clause does not force use of the index. Additionally, there should be no sorting required because the index is already sorted in the required order. In this case the ORDER BY clause is unnecessary because an identical result would be obtained without it:

```
SELECT country_id FROM country ORDER BY country_id
```

The next example re-sorts the result by country. Again, the whole table will be read. So, no index is used. The results are the same as for the query before the previous one:

```
SELECT country_id, country FROM country ORDER BY country
```

The ORDER BY clause will re-sort results even if records are already sorted according to the ORDER BY clause.

6.1.2.1 Overriding WHERE with ORDER BY

In the following query, if the WHERE clause reads an index, then the ORDER BY clause is likely re-sorting what the WHERE has already performed. This is a waste of processing power:

```
SELECT * FROM country WHERE region_id < 10
ORDER BY region_id, country_id
```

It is better just to remove the ORDER BY clause in some cases:

```
SELECT * FROM country WHERE region_id < 10
```

An ORDER BY clause can be used as a refinement of previous clauses, rather than to replace those previous clauses. The WHERE clause will filter rows, and the ORDER BY clause re-sorts those filtered rows. The ORDER BY clause can sometimes persuade the optimizer to use a less efficient key.

Be cautious about removing an ORDER BY clause, especially where composite indexing is used. There can be unexpected results. It is best to verify by examining an optimizer query plan. If the query plan might change in the future because your data changes frequently, or a lot over time, then don't use this option at all. Also, check the sorting of the result.

6.1.3 Grouping result sets

The GROUP BY clause can perform some inherent sorting. As with the SELECT statement, WHERE, and ORDER BY clause, matching of GROUP BY clause column sequences with index column sequences is relevant to SQL code performance.

In the following query, the region_id column is indexed, and thus the GROUP BY clause will read the index:

```
SELECT region_id, SUM(region_id) FROM country
GROUP BY country_id
```

In the next example both columns in a composite index are read. The composite index is much larger in both size and rows, so it will be more expensive:

```
SELECT region_id, country_id, SUM(region_id) FROM country
GROUP BY region_id, country_id
```

6.1.3.1 Sorting with the GROUP BY clause

The next example uses a non-indexed column to aggregate, and the whole table is accessed. The amount column is not indexed. The GROUP BY clause is now performing sorting on the AMOUNT column:

```
SELECT balance, SUM(balance) FROM accounts
GROUP BY balance
```

The ORDER BY clause in the next query is unnecessary. The ORDER BY clause in this query will likely be ignored by the optimizer:

```
SELECT balance, SUM(balance) FROM accounts
GROUP BY balance
ORDER BY balance
```

6.1.3.2 Using DISTINCT

DISTINCT retrieves the first value from a repeating group. When there are multiple repeating groups, DISTINCT will retrieve the first row from each group. DISTINCT will always require a sort (or read a presorted index) in order to return all repeating groups in the correct sequence. DISTINCT can operate on single or multiple columns. For example, the following query executes a sort in order to find the first value in each group:

```
SELECT DISTINCT(region_id) FROM country
```

6.1.3.3 The HAVING clause

The HAVING clause is intended to filter records from the result of a GROUP BY clause. The only common mistake with the HAVING clause is filtering records that can be more efficiently filtered using a WHERE clause. The WHERE clause is executed as the records are retrieved. So, a WHERE clause filter reading 10,000 rows from 1 million rows will find 10,000 rows. On the other hand, if somehow a HAVING clause can result in an aggregation on those 10,000 rows, then it is possible to read 1 million rows, aggregate on a million rows, and then return an aggregation of 10,000 rows—this is inefficient because you are reading 1 million rows, instead of 10,000 rows. So, this query finds the total amount of all orders from each customer:

```
SELECT customer_no, SUM(balance) FROM accounts
GROUP BY customer_no
```

This query reads all the customers, adds them all up, and then only returns the first 10 customers:

```
SELECT customer_no, SUM(balance) FROM accounts
GROUP BY customer_no
HAVING customer_no < 10
```

A more efficient form of the above query is to move the HAVING clause filter to the WHERE clause. Now the unwanted customers are not even added together but simply ignored:

```
SELECT customer_no, SUM(balance) FROM accounts
WHERE customer_no < 10
```

```
GROUP BY customer_no
```

The point of the HAVING clause is to apply the filter to the aggregation itself. In the following query we only return customer order totals in excess of $100:

```
SELECT customer_id, SUM(amount) AS total FROM orders
WHERE customer_id < 10
GROUP BY customer_id
HAVING total > 100
```

6.2 Using functions

The most relevant thing to say about functions is that they should not be used where you expect an SQL statement to use an index. To resolve this issue, some databases allow creation of a function-based index. A function-based index contains the resulting value of an expression. An index search against that function-based index will search the index for the result of the calculation of the expression, not the expression or contents thereof.

When using a function in a query, do not execute the expression against a table field if possible. This example executes a data type conversion against a string:

```
SELECT * FROM customer WHERE zip = TO_NUMBER('94002')
```

This next example places the function against the table and will cause all records to be read, even if the zip code field is indexed (the index is on the zip code and not the string data type conversion of the zip code):

```
SELECT * FROM customer WHERE TO_CHAR(zip) = '94002'
```

6.2.1 Data type conversions

Some data type conversions will be allowed automatically, particularly when the values themselves are compatible (simple numbers converting to a string is an example), and when values are small. On the contrary, data type conversions are often a problem and will likely conflict with existing indexes. Unless implicit data type conversion occurs, as described above between number and string, indexes will likely be ignored.

The most obvious data type conversion concerns dates. Date fields in all the databases I have used are stored internally in one or another, but not the visible form. For example, a Julian number (Julian form date) is an integer value measured from a database-specific date in seconds. When retrieving a date value in a tool, such as SQL Server Management Studio, there is usually a default date format to make the date readable. The internal date value is converted to that default format. The conversion is implicit, automatic, and transparent. However, it might ignore indexing on date fields.

Note: In short, try to avoid using any type of data conversion function in any part of a SQL statement which could potentially match an index, especially if you are trying to assist performance by matching appropriate indexes.

In general, any ANSI SQL compliant relational database will allow functions to be included in many parts of a SQL statement (queries plus INSERT, UPDATE, and DELETE commands), including the WHERE clause, ORDER BY clause, GROUP BY clause, HAVING clause, and even in the select list of fields for a query.

Note: When using functions in SQL statements, it is best to keep the functions away from any columns involving index matching.

6.3 Comparison conditions

Different comparison conditions can sometimes have vastly different effects on the performance of SQL statements. Let's examine each in turn with various options and recommendations for potential improvement. These are the comparison conditions:

- Equi, anti, and range
  - `expr { [!]= | > | < | <= | >= } expr`
  - `expr [ NOT ] BETWEEN expr AND expr`
- LIKE pattern matching
  - `expr [ NOT ] LIKE expr`

- Set membership
  - `expr [ NOT ] IN expr`
  - `expr [ NOT ] EXISTS expr`
- Groups
  - `expr [ = | != | > | < | >= | <= ] [ ANY | SOME | ALL ] expr`

6.3.1 Equi, anti, and range

Using an equals sign (equi) is the fastest comparison condition if a unique index exists. Any type of anti comparison, such as != or NOT, is looking for what is not in a table— the entire table must be read regardless (sometimes full index scans can be used). Range comparisons scan indexes for ranges of rows.

This example performs a unique index hit in a large table, using the equals sign an exact hit single row is found:

```
SELECT * FROM accounts WHERE account_no = 100
```

The anti (!=) comparison finds everything but the single row specified and reads the entire table because there is no other way to search for something which is not present:

```
SELECT * FROM accounts WHERE account_no != 100
```

In the next case, the (<) comparison searches a range of index values rather than a single unique index value:

```
SELECT * FROM accounts WHERE account_no < 100
```

In the next example the whole table is probably read rather than using an index range scan. This is because for >= 100, most of the table will be read. The optimizer considers reading the table as being faster in a situation such as this one:

```
SELECT * FROM accounts WHERE account_no >= 100
```

In this next example the BETWEEN comparison causes a range scan on an index because the range of rows is small enough to not warrant a full table scan, but also not a single record to allow a unique index match on a single record (even if there is a single record between 10 and 20):

```
SELECT * FROM accounts WHERE account_no BETWEEN 10 AND 20
```

As already stated, by no means will all the queries in this chapter be executed always as stated in this text. The manner in which the optimizer executes a query, using indexes or a full table scan, is dependent on the how much of the table is read. Usually a small percentage forces a full table scan. Other factors also influence the optimizer, such as the presence of statistics and whether those statistics are up to date (reflecting the true nature of the data in a table).

6.3.2 LIKE pattern matching

The approach in the query plan used by the optimizer will depend on how many rows are retrieved and how the pattern match is constructed. This query finds a single row:

```
SELECT * FROM country WHERE country like 'Central Africa
Republic'
```

This next query also retrieves a single row but there is a wildcard pattern match. A full table scan is performed (even though only a single record is the result):

```
SELECT * FROM country WHERE country LIKE 'Central Af%';
```

For the next query, if there are enough matching rows then the next query executes a full scan of the table:

```
SELECT * FROM country WHERE country LIKE '%a%';
```

A pattern match using a % full wildcard pattern matching character anywhere in the pattern matching string will usually produce a full table scan.

In general LIKE will often read an entire table because LIKE usually matches patterns which are in no way related to indexes.

6.3.3 Set membership

Traditionally, IN should be used to test against literal values and EXISTS to create a correlation between a calling query and a subquery. IN will cause a subquery to be executed in its entirety before passing the result back to the calling query. EXISTS will stop once a result is found. IN is best used as a pre-constructed set of literal values:

```
SELECT * FROM country WHERE region_id IN (5, 10, 13)
```

So, when executing a correlation, it is possible that EXISTS is better than IN because EXISTS stops when it finds a result, and IN reads everything in the subquery every time the subquery is executed. The benefit of correlation is matching of indexes between calling query and subquery:

```
SELECT * FROM country WHERE EXISTS
      (SELECT country_id FROM region
      WHERE region_id = country.region_id)
```

The benefit of using EXISTS rather than IN for a subquery comparison is that EXISTS can potentially find much fewer rows than IN. IN is best used with literal values, and EXISTS is best used as applying a fast access correlation between a calling and a subquery.

Note: Some database engines do allow correlation between calling query and subquery using both IN and EXISTS. Some databases only allow correlated subqueries using EXISTS.

6.4 Joins

A join is a combination of rows extracted from two or more tables. Joins can be very specific, for instance an intersection between two tables, or they can be less specific, such as an outer join. An outer join is a join returning an intersection plus rows from either or both tables, not in the other table.

6.4.1 Efficient joins

What is an efficient join? An efficient join is a join SQL query that can be tuned to an acceptable level of performance. Certain types of join queries are inherently easily tuned and can give good performance. In general a join is efficient when it can use indexes on large tables, or is reading only very small tables. Moreover any type of join will be inefficient if coded improperly.

6.4.1.1 Intersections

An inner or natural join is an intersection between two tables. In mathematical set parlance an intersection contains all elements occurring in both of the sets (elements common to both sets). An intersection is efficient when index columns are matched together in join clauses. Intersection matching not using indexed columns will be inefficient. In that case you may want to create alternate indexes. On the other hand, when a table is very small the optimizer may conclude that reading the whole table is faster than reading an associated index, plus the table.

In the example below both of the two tables are so small that the optimizer does not bother with the indexes, simply reading both of the tables fully:

```
SELECT r.region, c.country FROM region r
    JOIN country c USING(region_id)
```

When joining a very small with a very large table, it is likely the small table would be read in full and the large table with an index:

```
SELECT b.branch_name, a.account_no FROM branch b
    JOIN accounts USING (branch_no)
```

The most efficient type of inner join will generally be one retrieving very specific rows, such as in the next example. Most SQL is more efficient when retrieving very specific, small numbers of rows:

```
SELECT r.region, c.country, b.branch_name, a.account_no
FROM region r JOIN country c USING (region_id)
    JOIN branch b USING (country_id)
        JOIN accounts a USING (branch_no)
WHERE a.account_no = 100
```

Note: The above query will cause errors because there is conflict between table alias names and fields in USING clauses—the ON clause should be used to specifically state alias.column = alias.column. The query is left this way for the sake of simplicity in understanding a concept, not to teach you ANSI SQL syntax. Other queries in this chapter are also pseudocoded in this manner.

6.4.1.2 Self joins

A self join joins a table to itself. Sometimes self-joining tables can be handled with hierarchical queries. Otherwise a self join is applied to a table containing columns within each row, which link to each other. One of the classic examples of a self-join hierarchy is a family tree:

```
CREATE TABLE familytree(
     parent CHAR(40) NULL,
     child CHAR(40) NOT NULL
)
```

Note: The parent is NULL only when the very first ancestor is found.

It is fairly efficient to join the tables using the link between parent and child—given that both fields are uniquely indexed—and individually (no composite keys):

```
SELECT parent.name, child.name FROM familytree parent,
familytree child
     WHERE child.parent = parent.child;
```

Note: An equijoin uses the equals sign (=), and a range join uses range operators (<, >, <=, >=, and the BETWEEN operator). In general the = operator will execute an exact row hit on an index and thus use unique index hits. The range operators will usually require the optimizer to execute index range scans. BTree (binary tree) indexes, the most commonly used indexes in relational databases, are highly amenable to range scans. A BTree index is a little like a limited depth tree and is optimized for both unique hits and range scans.

6.4.2 Inefficient Joins

What is an inefficient join? An inefficient join is an SQL query joining tables, which is difficult to tune, or it cannot be tuned to an acceptable level of performance. Certain types of join queries are inherently both poor performers and difficult, if not impossible, to tune. Inefficient joins are best avoided.

6.4.2.1 Cartesian Products

The ANSI join format calls a Cartesian product a cross join. A cross join is only tunable as far as columns selected match indexes, such that rows are retrieved from indexes and not tables. A cross join matches every record in one table, with every record in every other table. The result returns a number of records equal to a direct multiplication of the number of records in both tables. So, if there are 5,000 customers and 10,000 accounts, the following query will return 50 million records:

```
SELECT * FROM customer CROSS JOIN accounts
```

Note: Cartesian products are meaningless in OLTP databases because the information returned is not necessarily related. Cartesian products are sometimes used in data warehouses, or for reports requiring blank rows, or even for hierarchical materialized view structures. I have never used one myself. Also, data warehouse fact tables can become extremely large. Sometimes if you read enough data from a fact table, and read too many dimensions at once, the optimizer may simply give up and join the tables as a Cartesian product before anything else occurs.

6.4.2.2 Outer Joins

Tuning an outer join requires the same approach to tuning as with an inner join. The only point to note is that if applications require a large quantity of outer joins, there is likely to be a potential for data model tuning. The data model could be too granular. Outer joins are probably more applicable to reporting and data warehouse type applications.

An outer join is not always inefficient. The performance, and to a certain extent the indication of a need for data model tuning, depends on the ratio of rows retrieved from the intersecting joins, in comparison to rows retrieved outside of intersecting joins. The more rows retrieved from the intersection the better.

My question is this. Why are outer joins needed? Examine the data model first to see if outer joins are a result of poor data model design. Outer joins are common in OLTP environments where orphaned data exists, or there is too much granularity in design. Outer joins are more appropriate to reporting, data warehouses, and data warehouse analytical processing.

Outer joins can be left outer joins, right outer joins, or even full outer joins. This query finds the intersection of the two tables, in addition to all records in the table on the left, but not in the table on the right:

```
SELECT * FROM customer LEFT OUTER JOIN accounts USING
(customer_no)
```

Similarly, a right outer join finds the intersection, plus all records in the right side table, but not in the left:

```
SELECT * FROM customer RIGHT OUTER JOIN accounts USING
(customer_no)
```

And a full outer join is simply a combination of intersection, left outer join, and right outer join:

```
SELECT * FROM customer FULL OUTER JOIN accounts USING
(customer_no)
```

Note: A full outer join is not the same as a Cartesian product because the full outer join does actually include an intersection and doesn't simply join everything with everything else, without any matching whatsoever.

Note: Excessive use of outer joins is possibly indicative of an overgranular data model structure. However, it could also indicate orphaned child table rows or the opposite: redundant static data. Cleaning out redundant or orphaned rows can sometimes help performance immensely by negating the need for complex and expensive outer joins on what is really useless data.

6.4.2.3 Anti-joins

An anti-join is always a problem. It can be useful from a business perspective but an anti-join simply does the opposite of a requirement. The result is that the optimizer must search for everything not meeting a condition. An anti-join will generally always produce a full table scan, as seen in the following example. Again the Rows and Bytes columns are left as overflowing showing the possible folly of using anti-joins:

```
SELECT * FROM customer c JOIN accounts a
WHERE a.customer_no != c.customer_no
```

Note: An anti-join might be so completely contrary to proper query tuning that it might even be a worse thing than a Cartesian product.

6.4.3 How to tune a join

So how can a join be tuned? There are a number of factors to consider:

- Use equality first.
- Use range operators where equality does not apply.
- Avoid use of negatives in the form of != or NOT.
- Avoid LIKE pattern matching.
- Try to retrieve specific rows, and in small numbers.
- Filter from large tables first to reduce the number of rows joined. Retrieve tables in order from the most highly filtered table downwards, preferably the largest table, which has the most filtering applied.

Note: The most highly filtered table is the largest table having the smallest percentage of its rows retrieved.

- Use indexes wherever possible except for very small tables.
- Let the optimizer do its job.

- Specialized objects and functionality such as materialized views and query rewrite can be very useful, both for OLTP databases and data warehouses.

6.5 Using subqueries for efficiency

Tuning subqueries is a highly complex topic. Quite often subqueries can be used to partially replace subset parts of very large mutable joins, with possible enormous performance improvements. This I have personally seen in highly normalized data models utilizing higher normal forms beyond 3rd normal form.

6.5.1 Correlated versus non-correlated subqueries

A correlated subquery allows a correlation between a calling query and a subquery. A value for each row in the calling query is passed into the subquery to be used as a constraint by the subquery. A non-correlated or regular subquery does not contain a correlation between calling query and subquery. The subquery is executed in its entirety, independently of the calling query, for each row in the calling query. Tuning correlated subqueries is easier because values in subqueries can be precisely searched for in relation to each row of the calling query.

A correlated subquery will access a specified row or set of rows for each row in the calling query. Depending on the circumstances, a correlated subquery is not always faster than a non-correlated subquery. Use of indexes or small tables inside a subquery, even for non-correlated subqueries, does not necessarily make a subquery perform poorly.

6.5.2 IN versus EXISTS

We have already seen commentary on the use of IN and EXISTS in the section on comparison conditions. We know already that IN is best used for small tables or lists of literal values. EXISTS is best used to code queries in a correlated fashion, establishing a link between a calling query and a subquery. To reiterate—it is important to remember that using EXISTS is not always faster than using IN.

6.5.3 Nested subqueries

Subqueries can be nested where a subquery can call another subquery, and the subquery can yet again call another subquery (recursion). The following example contains a query calling a subquery, which in turn calls another subquery, and so on:

```
SELECT * FROM region WHERE region_id IN
(SELECT region_id FROM country WHERE country_id IN
(SELECT country_id FROM branches WHERE branch_no IN
    (SELECT branch_no from accounts)))
```

A query like the above query can sometimes be more efficient than its equivalent join query, particularly for a highly normalized data model. Depending on the relational database, the data model, statistics, and some other factors—EXISTS may be faster than IN because EXISTS allows more efficient correlation between calling query and subquery:

```
SELECT * FROM region WHERE region_id EXISTS
(SELECT region_id FROM country WHERE country_id EXIST
(SELECT country_id FROM branches WHERE branch_no EXISTS
    (SELECT branch_no from accounts)))
```

Nested subqueries can be difficult to tune but can often be a viable, and sometimes highly effective tool for the tuning of mutable complex joins, with three and sometimes many more tables in a single join. There is a point when there are so many tables in a join that the optimizer can become less effective. However, using nested subqueries, it might be easier to tune a large join query because one can tune each subquery independently of the rest of the entire join query. This is why a calling query plus subquery combination is often known as a semi-join.

Note: Subqueries are also known as semi-joins because the calling query does not return any information from a subquery (it is only semi- or partially joined).

6.5.4 Advanced subquery joins

For very large complex mutable joins it is often possible to replace joins or parts of joins with subqueries. Very large joins can benefit the most because they are difficult for programmers to decipher, and thus just as difficult to tune by either human hand or optimizer. Huge joins can be made more tunable in a number of ways:

- A table in a join not returning a column in the primary calling query can be removed from the join and simply checked using a subquery. The table is not really part of the join so why retain it in the data being returned for display? This join query:

  ```
  SELECT r.region_id, r.region FROM region r
       JOIN country c ON(c.region_id = r.region_id)
  ```

  can be changed to:

  ```
  SELECT r.region_id, r.region FROM region r WHERE EXISTS
  (SELECT region_id FROM country
  WHERE region_id = r.region_id)
  ```

- A SELECT statement FROM clause can contain nested subqueries, breaking up joins much in the way that a Transact SQL procedure would use nested loops. This gives better control to programmers, allowing breaking up of queries into simplified parts. And a FROM clause subquery is not actually a semi-join (returns no columns) because it can return columns. This is why a FROM clause subquery is known as an inline view. An inline view is a little like a view, just *inline*, or within a query. This join query:

  ```
  SELECT r.region, c.country FROM region r
       JOIN country c ON(c.region_id = r.region_id)
  ```

  can be changed to:

  ```
  SELECT r.region FROM region r,
       (SELECT country FROM country
       WHERE region_id = r.region_id)
  ```

- An ORDER BY clause is always applied to a final result and should not be included in subqueries if possible. Why sort a subquery when sorting is usually required by the final result? The final result is returned by the query returning the rows and columns for display. A calling query does not need to see a subquery in sorted order because no human eyes are looking at its results; only the calling query is using the results of the subquery.

- DISTINCT will always cause a sort and is not always necessary. Sometimes a parent can be used where a unique value is present.
- When testing against subqueries, retrieve, filter, and aggregate on indexes and not against tables. Indexes usually offer better performance.
- Do not be too concerned about full table scans on very small static tables.

6.6 Specialized metadata objects

A synonym is as its name implies: it is another name for a known object. Synonyms are typically used to reference tables between schemas or to all users in general. Apart from the obvious security issues, there can be potential performance problems when overusing synonyms in highly concurrent environments.

Don't create too many synonyms. With enormous numbers of database objects, a relational database just has more stuff to manage. Metadata objects—such as tables, views, and synonyms—all have very high memory concurrency requirements. In short, they are used a lot. For example, a busy table has its table structure accessed far more frequently than its entire record set, especially if single records are retrieved. Obviously more memory can be allocated to the database (or added to hardware), but then more memory is simply more stuff for the server to manage in other respects.

Overuse of objects like synonyms and views is often a development practice, sensible for the purpose of hiding or burying complexity. This approach is highly effective for organizing programming code and database structure. However, it can often lead to complexity and performance problems in production.

Like synonyms, views are application friendly and security friendly and can be used to reduce, hide, or bury complexity. This is particularly the case in development environments. In general, views are not conducive to good performance because they are often overused, or even misused.

A view is a logical overlay on top of one or more tables. A view is created using an SQL statement. A view does not contain data itself. The biggest problem with a view is that whenever it is queried, its defining SQL statement is re-executed. It is common in applications for a developer to query a view and add additional filtering. The potential results are views containing large queries where programmers will then execute small row number retrievals from the view. The result is that two queries are executed, commonly with the view query selecting all the rows in the underlying table or join.

The performance issue is not that things like view and synonyms are bad things, but that they are often inappropriately used, or just used far too much.

A materialized view is not the same thing as a view. Don't confuse the two. A materialized view materializes data, in that it makes a copy of data. So, when a materialized view is read, underlying tables are not read, but the copy is read. This reduces conflict between OLTP application reading tables, and reporting or data warehousing reading all the data copied to a materialized view. Heavy I/O activity is very bad for OLTP database performance.

SQL Server does not allow explicit creation of, or even direct access to, materialized views. In SQL Server, materialized views are all internalized within analysis services and cannot be used outside of the context of the analytical processing engine. In other words, you can't explicitly build a materialized view, you can't explicitly access a materialized view (using a SQLCMD), and you can't force the optimizer to use or ignore query rewrite. In essence, materialized views are utilized in SQL Server 2005 but completely contained within Analysis Services.

Beware of including binary objects, or large objects, into queries. Sometimes it is most efficient to store binary objects external to a database, and store on a reference in the database itself.

6.7 Procedures in Transact SQL

Transact SQL and other programming languages (using the .NET Framework) allow for a programmer to have more control over how database access code is written. If the coder is skilled, then it is possible that Transact SQL database access stored procedures are not only easier to write and easier to tune, but perhaps even faster to execute because only so much can be done to tweak an absurdly complex query.

There are a number of reasons for resorting to stored procedures in Transact SQL:

- A stored procedure may not provide better performance but can allow a breakdown of SQL coding complexity. Breaking down complexity can allow easier tuning of SQL statements, giving more precise and more easily decipherable programming control—to the programmer.

- Stored procedure coding is stored in the database server and provides centralized control and potential performance increases. It also prevents multiple copies of SQL code with application code or worse on client machines. Stored procedure code is executed on the server. Execution on a database server reduces network traffic. The result is improved performance. However, centralization has its limits where the database server can become top-heavy and performance can actually slow.
- There are some situations where it is impossible to code SQL code using ANSI SQL queries and DML (Data Manipulation Language) statements alone. This is becoming less frequent as basic ANSI SQL syntax becomes more and more sophisticated.

7

What Is Query Optimization?

When we execute a query, either by typing in a Transact-SQL statement or by using a tool such as Microsoft Access, it is highly likely we will require that rows be read from one or more database tables. Suppose we require that SQL Server performs a join of two tables: table A containing a dozen rows, and table B containing a million rows. How should SQL Server access the required data in the most efficient manner? Should it access table A looking for rows that meet the selection criteria and then read matching rows from table B, or should it access table B first? Should it use indexes, if any are present, or perform a table scan? If indexes are present and there is a choice of index, which one should SQL Server choose?

The good news is that SQL Server contains a component known as the query optimizer, which will automatically take a query passed to it and attempt to execute the query in the most efficient way. The bad news is that it is not magic, and it does not always come up with the best solution. A database administrator should be aware of the factors that govern query optimization, what pitfalls there are, and how the query optimizer can be assisted in its job. Database administrators who know their data well can often influence the optimizer with the judicious use of indexes to choose the most efficient solution.

What do we mean by efficient in the context of the query optimizer? Basically, the query optimizer is looking to minimize the number of logical reads required to fetch the required data. The query optimizer is the SQL Server AutoRoute Express, choosing the best route to the data. Unfortunately, the query optimizer doesn't show you the golf courses on the way!

The query optimizer's main task, therefore, is to minimize the work required to execute a query, whether it is a query that retrieves data from a single table or a query that retrieves data from multiple tables participating in a join.

Although we have referred only to queries, the query optimization process is necessary for SELECT, INSERT, UPDATE, and DELETE Transact-SQL statements. This is because the UPDATE and DELETE Transact-SQL statements will often contain a WHERE clause, and the INSERT statement may contain a SELECT clause.

7.1 When is a query optimized?

When a query is submitted to SQL Server, various phases of processing occur. First of all, the query is parsed—that is, it is syntax checked and converted into a parsed query tree that the standardization phase can understand. The standardization phase takes the parsed query tree and processes it to remove redundant syntax and to flatten subqueries. This phase essentially prepares the parsed query tree for query optimization. The output of this phase is a standardized query tree. This phase is sometimes known as normalization.

The query optimizer takes the standardized query tree and investigates a number of possible access strategies, finally eliminating all but the most efficient query execution plan. In order to formulate the most efficient query execution plan, the query optimizer must carry out a number of functions. These are query analysis, index selection, and join order selection.

Once the most efficient query execution plan is produced, the query optimizer must translate this into executable code that can execute under Windows operating systems. This code can then access the appropriate indexes and tables to produce the result set.

Figure 7.1 shows a simplified diagram of how query optimization takes place. In reality the process is much more complex but this gives us a basic idea.

How does the query optimizer work out the most efficient query execution plan? We will look at the way it does this now. We will see that it takes in the information available to it in the form of the query itself, indexes and key distribution statistics, size of the table, and rows per page, and then calculates the logical read cost given a possible access path.

7.2 The steps in query optimization

The query optimization phase is the phase we will concern ourselves with in this chapter. This phase can be broken down into a number of logical steps, as follows:

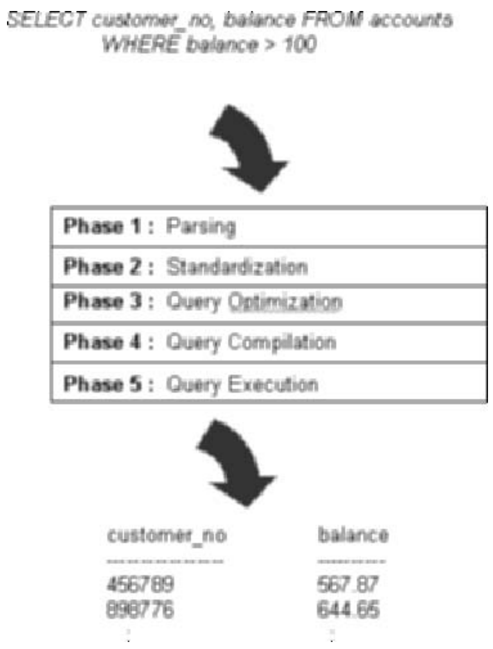

SELECT customer_no, balance FROM accounts
WHERE balance > 100
Phase 1 : Parsing
Phase 2 : Standardization
Phase 3 : Query Optimization
Phase 4 : Query Compilation
Phase 5 : Query Execution
customer_no
balance
456789
567.87
898776
644.65

Figure 7.1 *The phases of query processing*

1. Query analysis
2. Index selection
3. Join order selection

Let us discuss each step in sequence.

7.3 Query analysis

The first step the query optimizer performs during the query optimization phase is query analysis. In this step the query optimizer examines the query for search arguments (SARGs), the use of the OR operator, and join conditions.

7.3.1 Search arguments

A search argument is the part of a query that restricts the result set. Hopefully, if indexes have been chosen carefully, an index can be used to support the search argument. Examples of search arguments are as follows:

```
account_no = 7665332
balance > 30
lname = 'Burrows'
```

The AND operator can be used to connect conditions, so another example of a valid search argument would be as follows:

```
balance > 30 AND lname = 'Burrows'
```

Examples of common operators that are valid in a search argument are =, >, <, , and . Other operators such as BETWEEN and LIKE are also valid, because the query optimizer can represent them with the common operators listed above. For example, a BETWEEN can always be represented as AND. For example:

```
balance BETWEEN 1000 AND 10000
```

Becomes:

```
balance >= 1000 AND balance <= 10000
```

A LIKE operator can always be represented as AND <. For example:

```
lname LIKE 'Burr%'
```

Becomes:

```
lname >= 'Burr' AND lname < 'Burs'
```

The expression balance BETWEEN 1000 AND 10000 is not equivalent to balance BETWEEN 10000 AND 1000, because the expression >= 10000 AND <= 1000 finds nothing between two sensible values. The query optimizer will not detect the mistake and switch the values.

There are a number of expressions that are not considered to be search arguments. The NOT operator is an example:

```
NOT IN ('CA', 'NH', 'TX')
customer_no <> 9099755
```

```
balance <> 78000
```

Another example of this is the use of NOT EXISTS.

NOT is not considered to be a search argument, because it does not limit the search. Whereas account_no = 100,000 specifies a single value in a table that may potentially be efficiently retrieved using an index, account_no <> 100,000 will cause SQL Server to look at every row in the table to ensure that the account_no column does not contain this value.

There are other expressions that are not considered to be search arguments. If a column is used instead of an operator, the expression is not considered to be a search argument. For example:

```
loan < loan_agreed
```

How can SQL Server use such an expression to restrict the result set? It cannot, since the loan_agreed value is not known until the row is read; until it is known, it cannot be used to compare against the loan column. This will normally result in a table scan or index scan if the query is covered.

Another example of an expression that cannot be considered for query optimization is one that involves mathematics or functions. For example:

```
balance * 1.175 > 10000
UPPER(lname) = 'SHARMAN'
```

Against my database, using mathematics as in the first example, the query optimizer chose to use a non-clustered index on balance as long as the number of rows returned was low. The query optimizer had done the math and estimated correctly the number of rows returned whose balance would be greater than 10,000/1.175. However, using a function such as CEILING() caused a table scan to be performed, as in:

```
SELECT * FROM accounts WHERE CEILING(balance) = 100
```

String functions, as in the previous example using the UPPER function, will cause the query optimizer to resort to a table scan. A number of common string functions cause a table scan to be performed—for example, LEFT().

The bottom line is that using a function or expression on the left side of the operator will cause the query optimizer to use a table scan. In short, don't place functions onto table columns if you can avoid it. Apply functions to literal values, not table columns. That way the filter may be able to use an index on the table column. This was already covered in Chapter 6 so further explanation is unnecessary. This is one reason why it is very important to check the query execution plan that the query optimizer has produced—it may not be what you expect! We'll see how to check the query execution plan shortly.

As long as we have just a column on the left side of an appropriate operator, we have a search argument. We can often compare the column with an expression, so that the query optimizer will be able to use the distribution steps in the index key distribution statistics for the index rather than just the density values. Distribution statistics will be covered shortly. This is true as long as the expression can be evaluated before the query execution phase—in other words, before the query actually runs. An example of such a search argument would be as follows:

```
monthly_yield = items_processed/12
yearly_amount = daily_rate * 365
```

However, consider the following query:

```
sell_by_date > DATEADD (DAY, -10, GETDATE())
```

The query optimizer will choose a table scan. Again, this is common when a function is used. So check the query execution plan carefully!

How can we make sure that the index is used? There are various techniques, which we will discuss shortly. We could put the query in a stored procedure and pass the result of the function as a parameter. We may be able to create a computed column on the table and index it. Depending on what we want to achieve, this may or not make sense. However, we can only index a computed column if the computation is deterministic. This was discussed in the previous chapter. The function GETDATE() is not deterministic. We can also force the query optimizer to use an index. This technique is discussed later but should be used with care.

If the query optimizer cannot evaluate the expression until the query runs—that is, until after the query optimization phase has completed, then

SQL Server has no chance of making use of distribution steps. A classic example of this is where variables are used:

```
DECLARE @bal MONEY
SELECT @bal = 9990.23
SELECT * FROM accounts WHERE balance > @bal
```

In the BankingDB database, the previous example used a table scan instead of the non-clustered index on balance. If we do not use a variable, the non-clustered index is used, as follows:

```
SELECT * FROM accounts WHERE balance >  9990.23
```

This is different from stored procedure parameters, which will be discussed later in this chapter.

7.3.2 OR clauses

The query optimizer also checks the query for ORs. The OR clause links multiple search arguments together. For example, we might have a query that looks like the following:

```
SELECT * FROM customers
OR WHERE age > 40
OR height < 2
OR weight < 200
OR state = 'NH'
OR city = 'Manchester'
```

Any row matching any of the above conditions will appear in the result set. A customer will be displayed who lives in the city of Manchester in the United Kingdom or who lives in Nashua in New Hampshire. In other words, it is likely that many rows in the table will meet one or more of these criteria.

Compare the previous query with the following query:

```
SELECT * FROM customers
WHERE age > 40
AND height < 2
AND weight < 200
AND state = 'NH'
AND city = 'Manchester'
```

The number of rows in the table that meet all the criteria is likely to be far less. The ANDs restrict the result set, whereas the ORs widen it. For this reason a query containing ORs is handled in a particular way, which will be discussed later in the chapter. Because of this, the query optimizer looks for OR clauses in the query analysis step.

There may be OR clauses in the query that are hiding. Take the following query, for example:

```
SELECT lname, fname FROM employees
WHERE state IN ('CA', 'IL', 'KS', 'MD', 'NY', 'TN', 'TX')
```

At first glance there are no ORs in this query. The query optimizer sees this, however, as a number of OR clauses, as follows:

```
SELECT lname, fname FROM employees
WHERE state = 'CA'
OR state = 'IL'
OR state = 'KS'
OR state = 'MD'
OR state = 'NY'
OR state = 'TN'
OR state = 'TX'
```

7.3.3 Join clauses

After looking for search arguments and OR clauses the query optimizer looks for any join conditions. When more than one table is processed in a query, a join clause is usually found. The join clause can be in the WHERE clause or in the ON clause of the SELECT statement if ANSI standard join clauses are used. Here is an SQL Server join example:

```
SELECT fname, lname FROM customers, accounts
WHERE customers.customer_no = accounts.customer_no
AND balance > 10000
```

Here is an ANSI SQL join example:

```
SELECT fname, lname FROM customers
INNER JOIN accounts ON customers.customer_no =
    accounts.customer_no
WHERE balance > 10000
```

The following ANSI standard join clauses have been supported from SQL Server 2000: JOIN, CROSS JOIN, INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN. Sometimes a table can be joined with itself. This is known as a self join, or reflexive join. Although only one table is being accessed, the table is mentioned in the query more than once and so a join clause is used. The classic self join is the Employees table containing a column, supervisor_id, that holds a value found in the employee_id column elsewhere in the table. In other words, a supervisor is an employee. The Employees table might be defined as follows:

```
CREATE TABLE employees
(
   employee_id  CHAR(8),
   lname CHAR(10),
   fname CHAR(10),
   supervisor_id CHAR(8)
)
```

A query to retrieve the last name of the employee and the last name of the supervisor would be as follows:

```
SELECT e1.lname AS employee, e2.lname AS supervisor
FROM employees e1
   INNER JOIN employees e2 ON e1.supervisor_id =
e2.employee_id
```

7.4 Index selection

Having identified the search arguments in the query, the next step the query optimizer performs during the query optimization phase is index selection. In this step the query optimizer takes each search argument and checks to see if it is supported by one or more indexes on the table. The selectivity of the indexes is taken into consideration, and, based on this, the query optimizer can calculate the cost of a strategy that uses that index in terms of logical reads and CPU. This cost is used to compare strategies that use different indexes and a strategy that uses a table scan.

7.4.1 Does a useful index exist?

To obtain information on the indexes present on a table and their characteristics, SQL Server can check the sysindexes system table. From the sysindexes table the query optimizer can quickly establish the indexes present on the table by checking the rows that have a value in the id column equal to the object ID of the table (as defined in the sysobjects system table) and an indid column value > 0 and < 255. Other columns in the sysindexes table help the query optimizer determine on which columns the index is based.

The query optimizer will look for an index based on the same column as the search argument. If the index is a composite index, the query optimizer determines if the first column in the index is specified in the search argument.

If a search argument has no matching index, then no index can be used to support the search argument and so the query optimizer will look for indexes supporting other search arguments. If it is the only search argument, then a table scan will be performed.

7.4.2 How selective is the search argument?

Suppose the following query is presented to the query optimizer:

```
SELECT account_no FROM accounts
WHERE branch_no = 1005
AND balance > 5000
AND customer_no BETWEEN 10000 AND 110000
```

If there are indexes present on the branch_no, balance, and customer_no columns, how can the query optimizer decide which indexes are the most efficient to use—that is, which indexes will use the least number of logical reads and CPU to return the data? The query optimizer may choose to use no indexes, since a table scan is estimated to be a more efficient access mechanism, or it may choose to use one or more indexes.

The query optimizer has a number of mechanisms by which it can determine this information. The most accurate method is to use statistical information available in the key distribution statistics associated with the index. We will look at these distribution statistics shortly. If the key distribution statistics do not exist, the query optimizer applies a weighting to each operator. For example, the = operator has a weighting of 10 percent, which means that the query optimizer will assume that 10 percent of the rows in the table will be returned.

The approximate weightings of some common operators are shown in Table 7.1.

Table 7.1 *Weightings of common operations*

Operator	**Weighting**
=	10%
<	33%
>	33%
BETWEEN	12%

As you might imagine, these weightings are very general estimates and can be wildly inaccurate, so it is always best if the query optimizer is able to use the distribution statistics associated with an index.

If we have a unique index matching the search argument, then the query optimizer knows immediately the number of rows returned by the = operator. Because of the unique index, the query optimizer knows that at most one row can be returned (of course, zero rows could be returned), so this figure is used rather than the 10 percent weighting.

7.4.3 Key distribution statistics

Key distribution statistics are usually created when an index is created. The one exception to this is when an index is created on an empty table; otherwise, the index has key distribution statistics held for it. Note that the indexes on a table that has been truncated will have no key distribution statistics held for them. From now on we'll just refer to key distribution statistics as index statistics. We cannot just refer to them as statistics, since this is ambiguous. Why? Because a little later on we'll meet another type of distribution statistics known as column statistics.

Where are these index statistics held? They are held as a column in the sysindexes system table for the relevant row representing the index whose index statistics we wish to keep. This column is named statblob and is an

IMAGE datatype. Since it is an IMAGE datatype, there is plenty of room to hold lots of statistics, if required, although SQL Server keeps the number of statistics held to a fairly small but significant value. In my experience, for most indexes, the number of samples held in this column is in the low hundreds, typically about 300.

If this column is empty (holds NULL), then there are no index statistics associated with the index.

The statblob column holds index statistics information for the index to allow the query optimizer to quickly estimate the proportion of rows that will be returned by a search argument. Suppose we execute the following query on the Accounts table, which holds information for 25,000 accounts:

```
SELECT account_no FROM accounts WHERE balance > 9999
```

Will 25,000 rows be returned, or 1,000 rows, or 25 rows, or 0 rows? The query optimizer needs to know this information so it can decide whether a non-clustered index on the balance column should be considered interesting or whether a table scan is likely to be more efficient. Remember that returning a range of rows using a non-clustered index is going to result in a request for a data page (logical read) for every row returned. If the query optimizer can accurately estimate how many rows are likely to be returned, it knows with reasonable accuracy how many data page requests will be needed, and, therefore, it can calculate the cost of the query and compare this with the cost of a table scan.

In the BankingDB database there are, on average, less than five accounts that have a balance greater than 9,999, and so an indexed access should be more efficient than a table scan. But how can the query optimizer know this? It could count the number of rows that satisfied the search argument before it actually executed the query, but that would defeat the object of the exercise!

This is where the statblob column comes to the rescue. It holds a series of samples across the index key range that the query optimizer can check. Based on these samples the query optimizer can quickly estimate the percentage of the rows in the table that will be returned by the search argument using that index.

The statblob column actually holds a number of key values. This number is a function of the key size and the number of rows in the table. Although, in theory, the statblob column could hold up to 2 GB of key distribution statistics for an index, only a very large table would need that, and the effort in reading the index statistics would be far greater than the data. Of course, in practice, only a small amount of data space is needed in the statblob column, since few key values are actually held. For example, a non-clustered index on the account_no column in the Accounts table, which is a four byte (integer) key, has index statistics information consisting of 200 steps. By comparison, a non-clustered index on the account_notes column, which is a CHAR(400), has index statistics information consisting of 74 steps. The more steps, the more accurate the statistics, so, in this respect at least, it is better to have a smaller key value.

Suppose we have an index key that is an integer, such as the account_no column in our Accounts table. The initial key value found in the index is the first one to be sampled and stored in the statblob column, so we will have the statblob column contents shown in Figure 7.2.

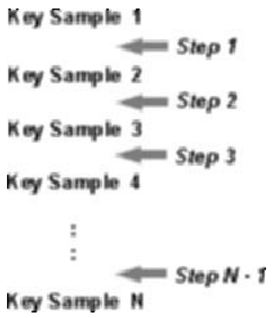


Figure 7.2
Distribution steps and keys in a statblob column

We can see that the number of distribution steps is typically going to be less than the number of key values stored in the statblob column. Apart from the choice of key size we cannot influence the number of key values held. However, as we shall see shortly, we can choose how much of our data is actually sampled in order to generate our index statistics.

What about composite indexes? SQL Server only stores key values for the first column. This means that it is better to choose the most selective column as the first column of an index—that is, the column with the least number of duplicate values. Of course, the first column of the index needs to be specified in the query, and choosing the most selective column will need to be done with this in mind.

I find that many database designers choose the key order in a composite index starting with the first column being the least selective, the next column being the next least selective, and so on, even if the query is going to specify all of the columns in its WHERE clause. Why is this? Usually it is because it is the most natural approach to take.

Consider a three-column composite index on the columns region, state, city. There is a natural hierarchy here—cities belong to states, which in turn belong to regions. There are few regions, more states, and many more cities. It's natural to create the index in the region, state, city column order just as you would in a report. But we can see that if we do this, we are going to populate the statblob column with few distinct values. This could result in the query optimizer choosing a table scan to execute the following statement when the non-clustered index would have been a better choice.

```
SELECT qty FROM sales
WHERE region = 'North'
AND state = 'CO'
AND city = 'Denver'
```

As we saw in Chapter 5, we can use the utility DBCC SHOW_STATISTICS to investigate index statistics. The format of this DBCC statement is as follows:

```
DBCC SHOW_STATISTICS (table_name, target)
```

The target is an index name or a statistics collection name. We will talk about statistics that are not index statistics later. For example, to show the index statistics on the non-clustered index nci-Balance, the following code segment would be used:

```
DBCC SHOW_STATISTICS (accounts, nciBalance)

Statistics for INDEX ' nciBalance'.
Updated  Rows Rows Sampled Steps Density  Average key length

Oct 20 2000  5:50PM 25000 25000 106 4.0426468E-5  8.0

All density           Average Length     Columns

4.0471085E-5          8.0                balance

RANGE_HI_KEy          RANGE_ROWS     EQ_ROWS    DISTINCT_RANGE_ROWS  AVG_RANGE_ROWS

.9500                 0.0            1.0         0                    0.0
88.1000               237.0          2.0         235                  1.0085106
237.0600              357.0          1.0         353                  1.0084746
282.3600              127.0          1.0         127                  1.0
316.1400              107.0          2.0         107                  1.0
```

```
413.7800          225.0        2.0        224              1.0044643
699.9500          735.0        2.0        729              1.0082304
723.5500          64.0         2.0        64               1.0
:
:
9696.2000         383.0        1.0        379              1.0078948
9739.9200         126.0        2.0        124              1.008
9998.5700         603.0        1.0        596              1.0100503
9998.8500         0.0          1.0        0                0.0
```

The index statistics shown above are associated with the non-clustered index based on the balance MONEY data type column in the Accounts table. The index statistics indicate that there are 111 steps and that 25,000 rows from the Accounts table were sampled to create these statistics. There are 25,000 rows in the Accounts table in total, so, in fact, all the rows were sampled. This is expected behavior when a CREATE INDEX statement generates the index statistics. Later we will see that other mechanisms to update the key distribution statistics will not necessarily sample all the rows.

If we look at the All Density value, we can see that it is 4.0471085E-5. The density is the average fraction of duplicate index key values in the index. Since the density is the reciprocal of the count of unique values in the index key, the count of unique values in our non-clustered index must be 1 / 4.0471085E-5, which yields 24,709 unique values, which is correct as checked with a SELECT DISTINCT (balance) query. Multiplying the total count of rows in the table by the index density, 4.0465478E-5, we can obtain the likely number of rows hit by specifying a given value, in our example:

```
row hits = 25000 * 4.0471085E-5 = 1.011777125
```

This means that a query specifying balance = value would usually return one row.

Note: Just to remind us of the terminology, this is an example of high selectivity and low density.

The output from DBCC SHOW_STATISTICS needs a few more words of explanation. The Density value, 4.0426468E-5, is close to the All Density value of 4.0471085E-5. The difference is due to the fact that the Density value considers non-frequent values. These are values that appear only once in a step. If a value appears more than once, it is considered to be a frequent value. The All Density value includes frequent values. The Average Length is the average length of the index key. If the index were a composite index, there would be an entry for the first column, first plus second column, and so on. The Average Key Length is the average length of the total composite index key (including a clustered index key pointer if there is a clustered index on the table). The average in this context is necessary because columns in the key can be variable-length datatypes.

The next section of data contains the sample steps themselves. Remember that they only apply to the first column in the key. The column RANGE_HI_KEY is the upper bound value of a histogram step—that is, the highest value in the step. The first step is the lowest value for the column in the table. The column RANGE_ROWS is the number of rows from the sample that fall within a histogram step, excluding the upper bound. By definition, this must be zero in the first sample. We can see that the seventh step contains the value 735 and that the values are quite varied across other steps. The column EQ_ROWS is the number of rows from the sample that are equal in value to the upper bound of the histogram step. In our data this varies between 1.0 and 2.0. For our seventh step this is 2.0.

The column DISTINCT_RANGE_ROWS is the number of distinct values within a histogram step, excluding the upper bound. For our seventh step there are 729 distinct values within the step, excluding the value 699.9500 (the upper bound). Finally, the column AVG_RANGE_ROWS is the average number of duplicate values within a histogram step, excluding the upper bound. This is defined as:

```
(RANGE_ROWS / DISTINCT_RANGE_ROWS
   for DISTINCT_RANGE_ROWS > 0)
```

For our seventh step this value is 1.0082304.

To check out some of these column values let us focus on the seventh step. Suppose we execute the SQL statement:

```
SELECT COUNT (balance) FROM accounts
   WHERE balance BETWEEN 413.7800 AND 699.9500
```

This SELECT specifies the two RANGE_HI_KEY values: for our seventh step and the one before. We find that 739 rows are returned. This is because the BETWEEN operator is inclusive, but the definition of the RANGE_ROWS column excludes the upper bound; the upper bound of the previous sample will not be included in the RANGE_ROWS column. So we need to rewrite the query, as follows:

```
SELECT COUNT (balance) FROM accounts
   WHERE balance > 413.7800 AND balance < 699.9500
```

We now find that 735 rows are returned, which is the correct value, matching the RANGE_ROWS value. Suppose we execute the following query:

```
SELECT COUNT (balance) FROM accounts WHERE balance=
699.9500
```

We find that the value 2 is returned, which agrees with the value in the EQ_ROWS column for this step. Finally, let us execute this statement:

```
SELECT COUNT (DISTINCT (balance)) FROM accounts
   WHERE balance > 413.7800 AND balance < 699.9500
```

This returns 729, which agrees with the value in the DISTINCT_RANGE_ROWS column for this step. We can then calculate RANGE_ROWS / DISTINCT_RANGE_ROWS, which is 735/729, giving 1.0082304. This agrees with the value in the AVG_RANGE_ROWS column for this step.

7.4.4 Column statistics

As well as maintaining statistics on indexed columns—to be precise, the first column of an index key—SQL Server will optionally maintain statistics on non-indexed columns. This includes columns in a composite index key other than the first. As long as the database option auto create statistics is set to on, if a column on which index statistics are not being maintained is referenced in a WHERE clause, statistics will be gathered if it would help the query optimizer.

This behavior can be suppressed by setting the database option auto create statistics to off. If this option is set to off, the fact that the query optimizer would like statistics information for the column is usually made obvious by a warning in the estimated query execution plan output in the Query Analyzer. A Missing Column Statistics event will also be evident in the SQL Server Profiler if it is being traced. If auto create statistics is set to on, an Auto Stats event will occur.

There are many occasions when SQL Server will automatically create column statistics. Basically, it will do so if they are missing and the query optimizer would really like to have them. Here is an example of such an occasion. Suppose we have created a non-clustered index with a composite key consisting of the customer_no and balance columns. We now execute the following query:

```
SELECT account_no  FROM accounts
WHERE customer_no = 100
AND balance = 100
```

The query optimizer knows the statistical distribution of key values in the customer_no column but not in the balance column. It will create statistics for this column because they are helpful in finding the most efficient query plan.

To find the column statistics that have been created automatically by the query optimizer, look for statistics with names similar to _WA_Sys_ balance_0519C6AF. To display statistics use the system stored procedure sp_helpstats or Tools Manage Statistics in the Query Analyzer.

Of course, you can manually create column statistics. Use Tools Manage Statistics in the Query Analyzer or the CREATE STATISTICS Transact-SQL statement. Alternatively, the system stored procedure sp_createstats can be used to create single-column statistics for all eligible columns for all user tables in the current database.

7.4.5 Updating index and column statistics

When do index and column statistics get updated? Statistics are not automatically updated when transactions that change the index commit. This would cause the statblob column to become a bottleneck. The statblob column is accurate when it is first constructed as part of the index creation (assuming there is data in the table at that time). After that, on a volatile index, the key distribution statistics will diverge from reality. It is the responsibility of the database administrator to ensure that the key distribution statistics are updated to reflect reality, and there are various ways to achieve this. The most common method is to use the Transact-SQL statement UPDATE STATISTICS. The format of this statement is as follows:

```
UPDATE STATISTICS table
[
        index
      | (statistics_name[,...n])
]
[
      WITH
      [
           [FULLSCAN]
          | SAMPLE number {PERCENT | ROWS}]
          | RESAMPLE
      ]
       [[,] [ALL | COLUMNS | INDEX]
       [[,] NORECOMPUTE]
 ]
```

If both the table name and index name are specified, the statistics for that index are updated. If only the table name is specified, the statistics for all indexes present on the table are updated. The same is true for column statistics, which are referred to by statistics_name. The FULLSCAN and SAMPLE number {PERCENT | ROWS} clause allows the database administrator to choose how much data is actually sampled from the table. The FULLSCAN option is used to specify that all the rows in a table should be retrieved to generate the key distribution statistics.

The SAMPLE number {PERCENT | ROWS} option is used to specify the percentage of the table or the number of rows to sample when generating statistics. This is typically used when a large table is being processed. SQL Server will make sure that a minimum number of rows are sampled to guarantee useful statistics. If the PERCENT, ROWS, or number option results in too small a number of rows being sampled, SQL Server automatically corrects the sampling based on the number of existing rows in the table.

Note that updating index statistics for a non-clustered index can be performed by scanning the leaf-level index pages of the non-clustered index, which may well be a lot faster than scanning the table rows. When updating column statistics, it is likely that the table will need to be scanned

If neither of these options is specified, SQL Server automatically computes the required sample size for the scan.

The RESAMPLE option specifies that an inherited sampling ratio will be applied to the indexes and columns. In other words, the sampling ratio from the old statistics will be used. When a table has undergone major changes (e.g., numerous deletes), SQL Server 2000 may override the inherited sampling factor and implement a full scan instead.

The ALL | COLUMNS | INDEX option specifies whether the UPDATE STATISTICS statement updates column statistics, index statistics, or both. If no option is specified, the UPDATE STATISTICS statement updates all statistics.

The NORECOMPUTE option specifies that statistics that become out of date will not be automatically recomputed. When statistics become out of date is a function of the number of changes (INSERT, UPDATE, and DELETE operations) that hit indexed columns. If this option is used, SQL Server will not automatically rebuild statistics. To switch automatic statistics re-computation back on, the UPDATE STATISTICS statement can be executed omitting the NORECOMPUTE option, or the system stored procedure sp_autostats can be used.

One might imagine that omitting the table name would cause the key distribution statistics on all of the indexes on all of the tables in the database to be updated. Not so; this will result in a syntax error. Microsoft provides a convenient way to accomplish this with the system stored procedure sp_updatestats. This will run UPDATE STATISTICS against all user tables in the current database. For example:

```
USE BankingDB
EXEC sp_updatestats
```

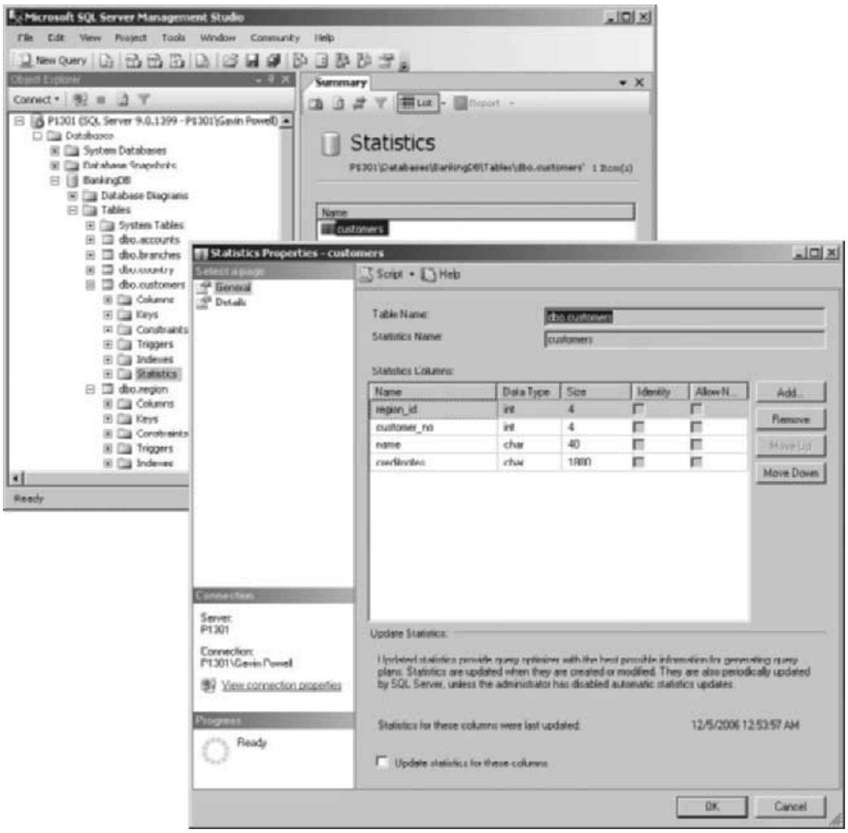
The above will update both index- and column-level statistics. Another way of achieving this might be to use a Transact-SQL cursor, as follows:

```
DECLARE tables_cursor CURSOR FOR
     SELECT table_name FROM information_schema.tables
      WHERE table_type = 'BASE TABLE'
OPEN tables_cursor
DECLARE @tablename NVARCHAR(128)
FETCH NEXT FROM tables_cursor INTO @tablename
WHILE (@@fetch_status <> -1)
     BEGIN
        EXEC ('UPDATE STATISTICS ' +  @tablename)
```

```
            FETCH NEXT FROM tables_cursor INTO @tablename
        END
        PRINT 'The statistics have been updated.'
    CLOSE tables_cursor
    DEALLOCATE tables_cursor
```

The above cursor creates a result set of all the user tables and then proceeds to update the key statistics of all the indexes on each one. Obviously, using sp_updatestats is more straightforward, but the cursor can be modified easily to only update the statistics of certain tables—for example, only those beginning with cust. This may be useful on a database consisting of large tables.

Another method of checking and updating statistics is to use the SQL Server Management Studio, I SQL Server 2005. Simply open up a table and edit the Statistics folder for a table, as shown in Figure 7.3.



Statistics
P1301\Databases\BankingDB\Tables\dbo.customers' 1 Item(s)

Table Name: dbo.customers
Statistics Name: customers

Statistics Columns:

Name	Data Type	Size	Identity	Allow N...
region_id	int	4	☐	☐
customer_no	int	4	☐	☐
name	char	40	☐	☐
coordinates	char	1800	☐	☐

Update Statistics:

Updated statistics provide query optimizer with the best possible information for generating query plans. Statistics are updated when they are created or modified. They are also periodically updated by SQL Server, unless the administrator has disabled automatic statistics updates.

Statistics for these columns were last updated: 12/5/2006 12:53:57 AM

☐ Update statistics for these columns

Figure 7.3
Statistics in SQL Server Management Studio

A possible reason for not using this wizard is that it will run UPDATE STATISTICS on all the tables in the database, and this may become a problem with a database consisting of large tables.

Updating distribution statistics can also be achieved using the Distributed Management Objects (DMO) interface. The Table object has methods named UpdateStatistics and UpdateStatisticsWith, which can be used to update the distribution statistics of all the indexes and columns on a table. The Index object and Column object also support these methods.

How can we easily tell when distribution statistics were last updated? This information is displayed by DBCC SHOW_STATISTICS. However, there is also a function called STATS_DATE that can be used. The format of this function is as follows:

```
STATS_DATE  (table_id, index_id)
```

To check the date the distribution statistics were last updated on all the indexes and column statistics on a given table, the following Transact SQL can be used:

```
SELECT
   ind.name AS 'Index/Column Statistics',
   STATS_DATE(ind.id, ind.indid) AS 'Date Last Updated'
   FROM sysobjects tab INNER JOIN sysindexes ind
   ON tab.id = ind.id
   WHERE tab.name = 'accounts'
```

This might give the following output:

```
Index/Column Statistics  Date Last Updated

nciBalance                2000-10-10 20:38:27.927
stat_branch_no  2000-10-10 20:38:28.627
```

Note that if there are no distribution statistics created for an index, because the index was created on an empty table, the Date Last Updated column will contain null. This should be a red warning light to a database administrator, who should run UPDATE STATISTICS without delay!

Another method that can be used to check when distribution statistics were last updated is to use the system stored procedure sp_autostats, described shortly.

Whichever method is chosen, the distribution statistics for an index or column on a table should be updated regularly or the query optimizer will start to use inaccurate information. An extreme example of this would be an index that was created on a table containing a single row that then had a million rows added. Most cases are not so extreme, but it is easy to forget to update statistics if no automated mechanism such as a scheduled task is set up. When the query optimizer chooses a strategy that you would not expect, the date the statistics were last updated is often the first information to check.

However, there is a safety net for the database administrator who forgets to update statistics. SQL Server 2000 contains functionality to automatically update statistics. This functionality is enabled globally for all the distribution statistics in a database by the database option auto update statistics.

Individual distribution statistics can have the automatic updating of statistics turned on or off by the use of the UPDATE STATISTICS statement with the NORECOMPUTE option. If UPDATE STATISTICS is executed with the NORECOMPUTE option, the automatic updating of statistics is turned off for the index or column distribution statistics referenced in the statement. If UPDATE STATISTICS is executed without the NORECOMPUTE option, the automatic updating of statistics is turned on for the index or column distribution statistics referenced in the statement.

The automatic updating of statistics may also be turned on or off by the sp_autostats system stored procedure. If this is executed with just the table name parameter, information is displayed regarding all the index- and column-level distribution statistics relevant to that table, as follows:

```
EXEC sp_autostats accounts

IndexName         AUTOSTATS  Last Updated

[nciBalance]      ON         2000-10-1020:38:27.927
[stat_branch_no]  ON         2000-10-1020:38:28.627
```

An index or column statistics name can be specified to limit the output:

```
EXEC sp_autostats @tblname=accounts, @indname= nciBalance

IndexName     AUTOSTATS   Last Updated
[nciBalance]  ON          2000-10-1020:38:27.927
```

Note that this system stored procedure also displays when the statistics were last updated.

7.4.6 When can we not use statistics?

Statistics cannot be used by the query optimizer if they are not there! As we have said, this occurs if the index was created on an empty table. In this case the STATBLOB column in the sysindexes table will contain NULL. If a table is truncated, the STATBLOB column will also be set to NULL. It follows, therefore, that if an index is created on an empty table, which is then populated with data, an UPDATE STATISTICS operation should be executed, or the query optimizer may create an inefficient query execution plan based on false assumptions. An UPDATE STATISTICS operation should also be run after a table has been truncated and repopulated. Of course, SQL Server may jump in and automatically update the distribution statistics if the appropriate database options are set, but why leave it to chance!

Not having distribution statistics present means that the query optimizer has little idea how many rows are likely to satisfy the query and, therefore, whether an index should be used. This is of particular importance when dealing with non-clustered indexes, since the query optimizer may decide not to use it and use a table scan instead. As an example, our Accounts table was created with a non-clustered index on the balance column. The table contained 25,000 rows. It was then truncated and repopulated with the 25,000 rows. The following query was then executed:

```
SELECT * FROM accounts WHERE balance = 100
```

The estimated query execution plan showed that the query optimizer had decided to use a table scan, and it had estimated that 1,988 rows would be returned. In fact, zero rows were returned, since no accounts had a balance of exactly zero. A bad decision, since the non-clustered index would have been the most efficient access method.

Even if statistics are present, they may not be used. When we discussed search arguments earlier in this chapter, we introduced cases where the query optimizer cannot evaluate the expression in the WHERE clause until the query runs—that is, until after the query optimization phase has completed. An example of this is using a variable, as follows:

```
DECLARE @bal MONEY
SELECT @bal = 4954.99
SELECT * FROM accounts WHERE balance = @bal
```

In this case distribution steps cannot be used when the query optimizer creates a query execution plan for the Transact-SQL batch, and the query optimizer will use the index density information present in the statblob column. Index density was discussed in Chapter 3 and is the average fraction of duplicate index key values in the index. It is the reciprocal of the count of unique values in the index key.

Suppose we have a Supplier table with a country_code column and we deal with suppliers from 20 countries. The index density would then be 1/20 = 0.05.

By multiplying the total count of rows in the table by the index density, we can obtain the likely number of rows hit by specifying a given value. Suppose our table contains 5,000 suppliers:

```
row hits = 5000 * 0.05 = 250
```

However, this does not take into account the fact that we might have many or few column values of UK; therefore, index density is a poor substitute for statistics.

An even worse substitute are the weightings we saw earlier in this chapter (shown in Table 4.1). These are used if there are no statistics.

7.4.7 Translating rows to logical reads

When the query optimizer has found a particular index interesting and has used the selectivity of the search argument to assess the number of rows returned, it translates this value into logical reads.

The way it does this translation depends on the index type—clustered or non-clustered—and whether there is actually an index present.

7.4.7.1 No index present

If we have no suitable index on the table, a table scan must be performed, as shown in Figure 7.4.

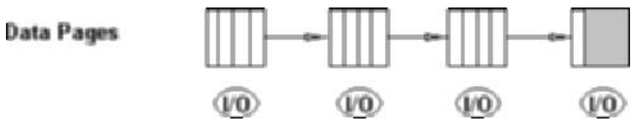


Figure 7.4 *Logical reads required for a table scan*

The number of logical reads a table scan will use is easy to calculate. All we have to do is find the number of database pages used by the table. We can find this information from the sysindexes system table by looking at the dpages column. In the BankingDB database the Accounts table uses 1,389 pages.

This is an extremely important number. We immediately know that we can retrieve all the rows from the Accounts table in 1,389 logical reads. This establishes a baseline value against which the query optimizer measures the cost of index access in terms of logical read.

7.4.7.2 A clustered index present

What if we can use a clustered index? SQL Server will have to traverse the index until the appropriate data page is reached. Because the data is in key sequence, this data page and any other relevant pages will then be retrieved. The cost of using a clustered index is the cost of the index traversal plus the data pages scanned, as shown Figure 7.5.

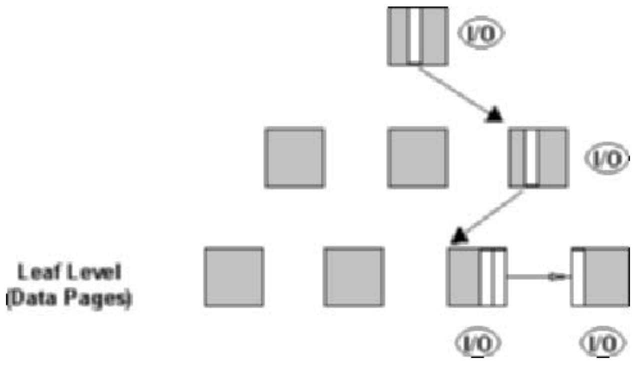


Figure 7.5 *Logical reads required for a clustered index*

We can estimate the number of data pages scanned by knowing the approximate number of rows per page. I tend to use the DBCC SHOW-

CONTIG statement with the TABLERESULTS option to find the average record size and then divide this into 8,000 to get the approximate number of rows per page. In the BankingDB database the Accounts table holds about 18 rows per page. Knowing this, you can estimate the number of data pages scanned if you know roughly how many rows will be returned by the query. But what about the index pages?

To find the number of logical reads used to traverse the clustered index we need to know the number of levels in the index. This is known as the depth of the index. Again, I tend to use the DBCC SHOWCONTIG statement with the TABLERESULTS option and the ALL_LEVELS option to find the number of levels in an index. The number of levels in an index will be the number of logical reads used to traverse the index. Most indexes will consist of a small number of levels, so the number of logical reads used to traverse an index can often be ignored.

7.4.7.3 A non-clustered index present

If there is a non-clustered index present, SQL Server will have to traverse the index until the appropriate leaf pages are reached. The pointers from the leaf pages will then have to be followed to each row pointed at by an index entry in the leaf page. Each data row may reside on its own data page, or a data page may host a number of the rows we wish to retrieve. This is irrelevant. Each row retrieved will result in a data page request—that is, a logical read. The cost of using a non-clustered index is then the cost of the index traversal plus the leaf pages scanned plus the cost of retrieving each row, as shown in Figure 7.6.

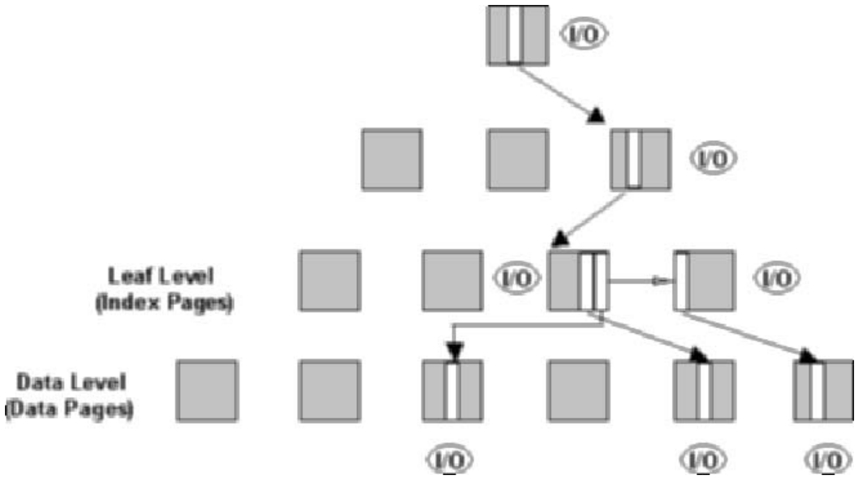


Figure 7.6 *Logical reads required for a non-clustered index*

This could result in many logical reads. If the query returns a range of rows, say 2,000, the query optimizer will assume that this will cost the number of logical reads to traverse the non-clustered index plus the number of logical reads to scan the relevant leaf pages plus 2,000 logical reads to retrieve the data rows. We can immediately see that in the case of our Accounts table, this is greater than our baseline value for a table scan. In other words, all other things being equal, the table scan would be the most efficient retrieval method.

Note: In fact, the query optimizer does not consider only logical reads. It also considers CPU. For this reason, a comparison alone between logical reads and the number of pages in the table is an oversimplification.

Clearly, if the query is only going to return one row—for example, when we use the = operator with a unique index, the cost is the index traversal plus the cost of retrieving the single data page, as shown in Figure 7.7. Compared with performing the same operation using a clustered index, the non-clustered index will usually take only one extra logical read.

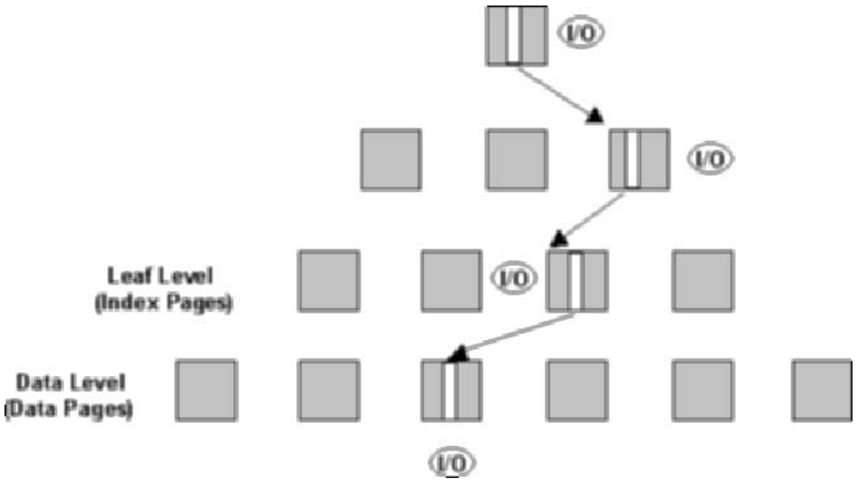


Figure 7.7 *Logical reads required for a non-clustered index, plus a single row retrieval*

We have previously mentioned the covered query, where all the information necessary is satisfied from the index leaf level without visiting the data. SQL Server will have to traverse the index until the leaf level is reached and then the relevant leaf-level pages are scanned, as shown in Figure 7.8.

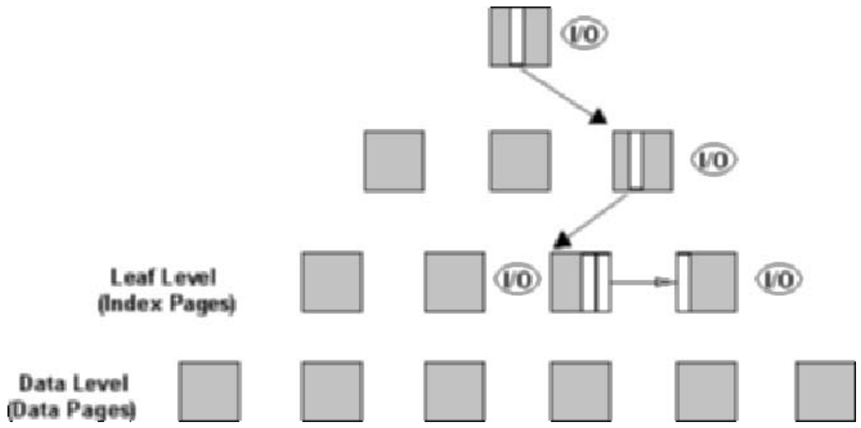


Figure 7.8 *Logical reads required for a covering non-clustered index*

7.4.7.4 A non-clustered index present and a clustered index present

We have already mentioned in Chapter 3 that the presence of a clustered index on a table results in the leaf-level index page pointers in any non-clustered indexes on the table to become the clustered index key instead of the Row ID.

So now, as well as SQL Server 2000 traversing the non-clustered index, it must also traverse the clustered index. Of course, a query that returns a range of rows will return a range of pointers from the non-clustered index, all of which will have to access the clustered index. The number of logical reads performed to access the non-clustered index will therefore be increased by the logical reads needed to access the clustered index.

However, there is a positive side to this. As we have stated, the presence of a clustered index on a table results in the leaf-level index page pointers in any non-clustered indexes on the table to become the clustered index key instead of the Row ID. In other words, the leaf-level index entries in the clustered index will now hold the non-clustered index key plus the clustered index key, and so there is more chance of the non-clustered index covering the query.

7.4.7.5 Multiple non-clustered indexes present

We have discussed the fact that range retrieval in a non-clustered index may result in a large number of data page requests such that a table scan is chosen in preference. But what if the WHERE clause of the query contains more than one filter. For example:

```
SELECT * FROM accounts
```

```
WHERE balance BETWEEN 100 AND 200
AND customer_no BETWEEN 1000 AND 2000
```

If we have a non-clustered index on the balance column and the range is reasonably selective, we should expect the query optimizer to choose a query execution plan that selects the data rows based on the index and then discards the ones where the customer_no column holds a value that is not in range. If the range is not selective, we will expect a table scan. But what if there is also a non-clustered index present on the customer_no column? As we discussed in Chapter 3, the query optimizer may be able to perform an index intersection. If the query optimizer believes that using both indexes will reduce the number of data page requests, then it will do just that.

7.5 Join order selection

If the query contains more than one table or the query performs a self-join, the query optimizer will derive the most efficient strategy for joining the tables. The order in which tables are joined can have a large impact on performance. For example, suppose we wanted to run the following query, which joins the Accounts table with the Customers table:

```
SELECT * FROM accounts INNER JOIN customers
ON accounts.customer_no = customers.customer_no
WHERE balance > 9990
```

Both tables have a non-clustered index on customer_no. Suppose the Customers table was accessed first. There is no restriction on the customer_no column and so all 12,500 customer rows would be retrieved, and for each of these rows the Accounts table would be accessed. It would, therefore, be accessed 12,500 times, and since each customer has two accounts, 25,000 account rows would be retrieved. Each one would then be tested for the restriction > 9,990.

Suppose, instead, the Accounts table was accessed first. The restriction would be applied, removing the majority of rows in the Accounts table and leaving only 21 rows with the balance column containing a value > 9,990. This means that the Customers table will only be accessed 21 times, considerably reducing the logical reads needed to execute the query. In fact in our BankingDB database this join order needed 96 logical reads against the 51,695 logical reads needed by the first join order!

The query optimizer can use information in the statblob column to help it choose an efficient strategy. We have already seen that the statblob column contains index density information, and it is this information the query optimizer uses to estimate how many rows from one table will join with rows from another table—that is, the join selectivity. The statblob column not only holds index density for a single column in a composite index but also the index densities of some of the column combinations. If the composite index contained three columns—COL1, COL2, and COL3, then the index densities held would be for the following combinations:

```
COL1                index density value (a)
COL1, COL2          index density value (b)
COL1, COL2, COL3    index density value (c)
```

Suppose the statblob column is not populated. In this case the query optimizer uses a formula to work out the join selectivity. It is simply the reciprocal of the number of rows in the smaller table. If we had a query that joined the Accounts table (25,000 rows) with the Customers table (12,500 rows), the join selectivity would be (1/12500) = 0.00008. For each row in the Customers table we would expect a match to (0.00008 * 250000) = 2 rows in the Accounts table.

7.6 How joins are processed

Prior to SQL Server 7.0, there was only one basic join mechanism available to the query optimizer to join two tables together. This was the nested loops join. In SQL Server 2000 there are three, as follows:

- Nested loops joins
- Merge joins
- Hash joins

The nested loops join is still the best general-purpose join available, but the merge and hash joins can be utilized by the query optimizer to produce a more efficient join plan in certain circumstances.

7.6.1 Nested loops joins

In the nested loops join, tables are processed as a series of nested loops, which are known as nested iterations. In a two-table join every row selected from the outer table (the table in the outer loop) causes the inner table (the table in the inner loop) to be accessed. This is known as a scan (not to be confused with table scan). The number of times the inner table is accessed is known as its scan count. The outer table will have a scan count of 1; the inner table will have a scan count equal to the number of rows selected in the outer table. Figure 7.9 illustrates a three-table join.

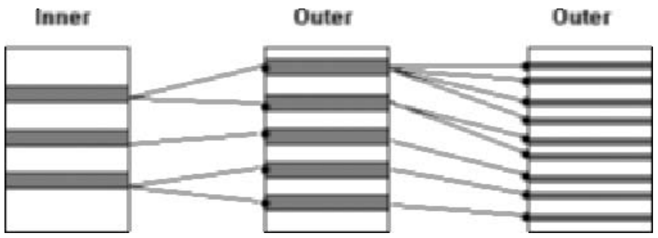


Figure 7.9 *A nested loop join of three tables*

The outer table will use indexes to restrict the rows if it can, whereas the inner table will use indexes on the join columns and potentially any other indexes that might be efficient in limiting the rows returned. However, the index on the join column is the most important index, since, without it, the inner table will be table scanned for each relevant row in the outer table. The optimizer should attempt to make the table with the smallest number of qualifying rows the inner table. It will make the attempt at least but won't necessarily succeed.

For example, consider an inner join between the Customers table and the Accounts table. There are 12,500 rows in the Customers table and 25,000 rows in the Accounts table. Suppose the Accounts table has a nonclustered index on the customer_no column. The query optimizer will choose the Customers table as the outer table (there is no index that is useful, and, besides, this is the smaller table). The Customers table will be passed through once. Its scan count will be one—that is, it is processed once, or, if you prefer, it is visited once.

There are 12,500 qualifying rows in the Customers table and so the Accounts table will be visited 12,500 times. It will have a scan count of 12,500. Luckily, there is a useful index on the customer_no column of the Accounts table so the table will not be table scanned 12,500 times!

Note: The emphasis in the above paragraph should be on the term qualifying. In other words, if you apply a WHERE clause to a table and get 1 row from 1 million, then you want to continue join execution with 1 row and not 1 million rows. So, if the second table in a join has another 1 million rows, it is obvious that much better efficiency is found by joining 1 row with 1 million rows, as opposed to 1 million with another 1 million rows. Apart from that the result of the former is 1 million rows. And the result of the latter join is 10^{12}. That's a huge difference in the amount of I/O.

Later on we will look at how we can return statistical information about the scan count and the logical read (pages requested) count. But for now just let me say that this join indeed results in the following statistics:

```
Table         Logical Read    Scan Count
Customers     736             1
Accounts      50,859          12,500
```

Note that the logical read count of the Customers table is 736, because there are 736 data pages in this table. The logical read count for the Accounts table is approximately four per scan count, indicating that for each access of the Accounts table, via the index, four logical reads were used (three for index pages and one for the data page).

The nested loop join does not depend on an equality operation relating the two tables together. The operator, for example, can be < or >.

If the outer table supplies only a few rows to the query—that is, it is a small table or is filtered by a WHERE clause and the inner table has an index on the join column—a nested loop join can be very efficient, usually more so than a merge or hash join. However, when large tables are being joined, a merge or hash join may be more efficient.

How many ways are there of joining two tables, A and B? There are in fact two ways: AB and BA. What about three tables? There are six ways: ABC, ACB, BAC, BCA, CAB, and CBA. What about four tables? The answer is four, or 1 * 2 * 3 * 4 = 24.

The number of ways, then, to join X tables is X!, or factorial X. If a query were to join 16 tables, we are talking about 20,922,789,888,000 possible ways of performing this join. A join of ten tables would have

3,628,800 possible combinations, and SQL Server can join a maximum of 256 tables in a SELECT statement!

Luckily, the query optimizer uses techniques internally to minimize the number of possible combinations, but the fact still remains that the more tables in a join the longer the query optimizer will take to work out the most efficient access strategy. Also, any inefficiency will be magnified enormously, since we are basically placing loops within loops within loops within a nested loops join.

The bottom line is: If you are going to execute a query that joins many tables, test it! Check what the final query execution plan is. Check the number of logical reads. Check the elapsed time. If you are not happy, then break the join down into parts, perhaps joining a subset of the tables into a temporary table and then joining that with the remaining tables. Or you could even use subqueries or CTE (common table expressions) and the WITH clause as opposed to a temporary table.

One useful rule of thumb is to make sure that if the number of tables in the query is N, then the number of join conditions is at least N – 1. For example, suppose we join three tables—TAB1, TAB2, and TAB3—and the join is over a column we will call C1. Suppose the query is as follows:

```
SELECT * FROM TAB1, TAB2, TAB3 WHERE TAB1.C1 = TAB2.C1
```

Applying our rule of thumb we can see that there are three tables in the join, so there should be at least two join conditions. There is only one join condition in the query, which is below the minimum number. This will result in SQL Server performing a lot of extra work joining all the rows in TAB3 with all the rows in the result set from the join of TAB1 and TAB2 or some combination of this. Depending on the form of the SELECT statement, the answer returned may be correct—for example, if a DISTINCT was used. The time taken to process the query, though, would be much greater than necessary.

Applying our rule of thumb we can see that the query should be written as follows:

```
SELECT * FROM TAB1, TAB2, TAB3
WHERE TAB1.C1 = TAB2.C1
AND TAB2.C1 = TAB3.C1
```

However, if it makes sense to add a third join condition, then do not be afraid to do so, since it will give the query optimizer more options to work with:

```
SELECT * FROM TAB1, TAB2, TAB3
WHERE TAB1.C1 = TAB2.C1
AND TAB2.C1 = TAB3.C1
AND TAB1.C1 = TAB3.C1
```

Of course, if you use the ANSI join syntax (recommended) with the ON clause you cannot miss the join condition.

7.6.2 Merge joins

Merge joins can be efficient when two large tables of similar size need to be joined and both inputs are already sorted by virtue of their indexes, or a sort operation is not expensive for sorting one or more of the inputs. The result from a merge join is sorted on the join column, and if this ordering is needed by the query, the merge join can supply it. The equality operator must be used in the query to join the tables; otherwise, a merge join cannot be used.

There are two types of merge join: a one-to-many (regular) and a many-to-many. In the case of a one-to-many, one input will contain unique join column values, whereas the other will contain zero, one, or many matching values. In the case of a many-to-many merge join, both inputs may contain duplicate join column values.

A many-to-many merge join requires that a temporary worktable is used, and this is apparent when looking at the logical read information that can be returned from a query (discussed later). In my experience, the added work required to process this worktable often means that the query optimizer uses one of the other join techniques—for example, a hash join in preference to the many-to-many merge join. If the join column from one input does contain unique values, the query optimizer will not know this unless a unique index is present on that column.

If the two join columns from the two input tables both have a clustered index created on them, the query optimizer knows that the rows are physically sorted on the join column. In this case the query optimizer does not need to perform a sort on any of the inputs. Joining these two tables will probably be performed with a merge join, especially if the merge join is a one-to-many. The presence of an ORDER BY clause on the query will increase the likelihood that a merge join is used.

If the two join columns from the two input tables both have a non-clustered index created on them, then the query optimizer knows that the rows are not physically sorted on the join column. In this case the query optimizer will need to perform a sort on the inputs. Joining these two tables with a merge join is less likely, unless an ORDER BY clause on the query is used. In this case the query optimizer will decide if a merge join is more efficient than nested loops or hash.

So how does a merge join work? Basically, the two tables being joined are visited once each. The scan count for each table is one. This is shown in Figure 7.10.

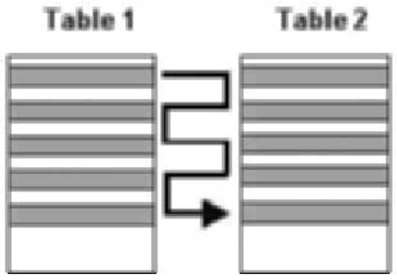


Figure 7.10 *Basic merge join algorithm*

The algorithm for a one-to-many merge join is as follows:

- Read a row from Table 1.
- Read a row from Table 2.
- If the join column values are equal, return all the matching rows.
- If the value from Table 1 is less than the value from Table 2, read the next row from Table 1.
- If the value from Table 2 is less than the value from Table 1, read the next row from Table 2.

The query optimizer carries on, stepping along each table until the processing is complete.

7.6.3 Hash joins

In my experience, hash joins are used by the query optimizer frequently in SQL Server—somewhat more, in fact, than merge joins. You would like to have fewer hash joins than nested loop joins because nested loop joins don't have the added step of building a hash key with which to search one of tables (or its index). Of course, this may not be the case with your application. With a hash join, there are two inputs: the build input and the probe input. The build input is typically the smaller table, although this may not be the table that uses fewer data pages on disk. Rather, it is the table with the least rows after selection criteria in the WHERE clause have been considered by the query optimizer. An interesting consideration with hash joins is that there need be no useful indexes on the tables to be joined. This means that the hash join mechanism can be used to join any two non-indexed inputs. This is very useful, because this is exactly the form that intermediate results in the query execution plan take. We will see examples of this later. The equality operator must be used in the query to join the tables; otherwise, a hash join cannot be used.

Assuming that the query optimizer has chosen the smaller table to be the build input; it now reads each row in turn from the table. For each row read, the value in the join column is processed by a hashing algorithm. Hashing algorithms apply some function to the input value to arrive at an output value. The important consideration is that when the same value is input to the hashing algorithm later, the value output is the same as was previously output.

In a hash join, the value returned by the hashing algorithm is used to identify a cell in memory known as a hash bucket. The row from the build input is then written into this hash bucket (at least the columns of interest to the query are). The number of hash buckets is a function of the size of the build input. It is best if the query optimizer can hold all of the hash buckets (the build input) in memory. It is not always possible to do this, and therefore several variations of the basic hash algorithm exist to facilitate the storing of hash buckets to disk. Two of these mechanisms are known as a Grace Hash Join and a Recursive Hash Join.

Once the build input has completed, the probe input phase starts. Each row in the probe input (the bigger table) is read, and the value of the join column is input to the same hash algorithm. The resulting value again identifies a hash bucket. The query optimizer then checks the hash bucket to see if there are any rows in it from the build input with the same join column value. If there are, the row is retrieved from the hash bucket and, with the row from the probe phase, returned to the query. If there is no match, the row may be discarded depending on the type of join being performed.

With a hash join, both tables are visited just once—that is, each has a scan count of one. Memory is needed for the hash buckets, so hash joins tend to be memory and CPU intensive. They typically perform better than merge joins if one table is large and one is small, and they are better than nested loops joins if both tables are large. However, because the build input is performed before any rows are returned, hash joins are not efficient when the first row of the join must be retrieved quickly.

Figure 7.11 shows a build input being processed. A row has been read with a value of 3 in the join column. This is hashed to a value of 11, and the row (relevant columns) is written into the hash bucket. Later, the probe input is processed. A row is read with a value of 3 in the join column. This is hashed to a value of 11, and the query optimizer checks to see if there is a matching row in the hash bucket from the build input. There is, so the rows are concatenated and returned to the query.

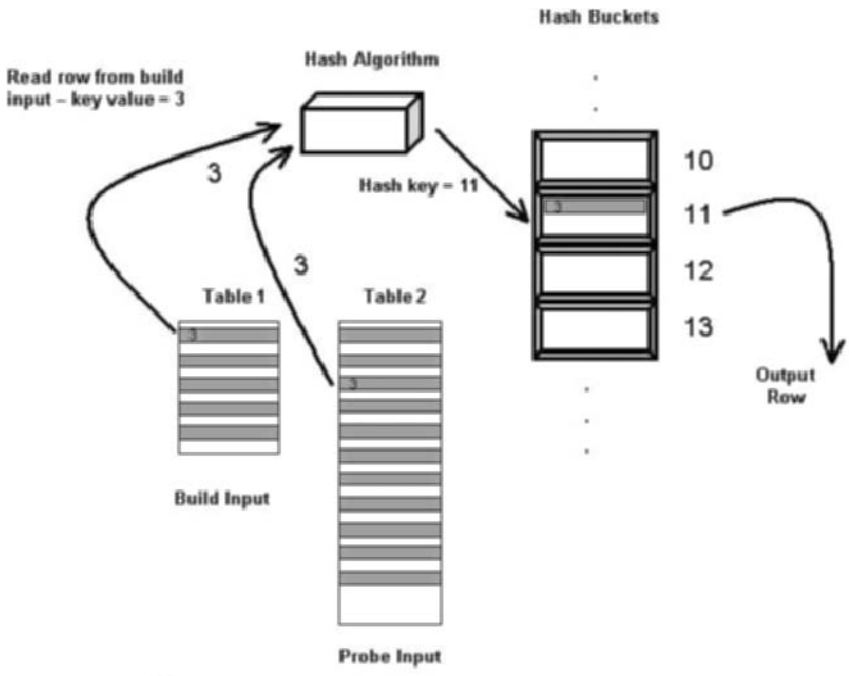


Figure 7.11 *Basic hash join algorithm*

The hash join mechanism, as previously mentioned, can be used to join non-indexed inputs. One example of this is when the query optimizer has created a plan that involves two sets of index pointers, and pointers need to be found that exist in both sets—that is, the set intersection. Hashing is also useful when the query contains an aggregate operator—for example, SUM or MAX with a GROUP BY. Using SUM as an example, suppose we want to find the sum of the bank balances for the accounts managed by each branch on a per branch basis, as follows:

```
SELECT branch_no, SUM(balance)
FROM accounts
GROUP BY branch_no
```

The query optimizer may choose to create a query execution plan using a hashing mechanism. The build input creates a set of hash buckets and then reads each row in turn. The branch number of the first account (the GROUP BY column) will be hashed, and the branch number and account balance values will be written into the appropriate hash bucket. This process will continue for each row. However, if a branch number is found to be present already in a hash bucket, the balance will be added to the value present. Finally, when all the rows have been retrieved, the hash buckets are scanned and the branch number values returned with their sums.

Note: This mechanism will produce a non-ordered output, so, as always, use an ORDER BY clause if you wish the output to be ordered.

This Page Intentionally Left Blank

Investigating and Influencing the Optimizer

In the previous chapter, we discussed the steps that the query optimizer performs during query optimization:

- Query analysis
- Index selection
- Join order selection

To facilitate performance, tuning, and optimization it is essential that we are able to see the decisions that the query optimizer has made so that we can compare the decisions with what we expect. We also need to be able to measure the work done in executing the query so we can compare the effectiveness of different indexes.

Note: You should always calculate a rough estimate of the logical reads a query should use. If the logical reads used differ significantly from the estimate, it could be that your estimate is very inaccurate or, more likely, the query execution plan is not what you expected!

There are a number of tools at our disposal for checking what the query optimizer is doing. SQL Server 2000 and before had all sorts of tools for this including the Query Analyzer, the graphical query execution plan, and the SQL Server Profiler. The SQL Server Profiler is discussed in Chapter 11. We will focus our discussion here on the graphical query execution plan, but, first, let us investigate the SET statements and options available to us. The graphical interface in SQL Server 2005 is now part and parcel of the SQL Server Management Studio, which will be examined shortly.

Before beginning with this chapter in earnest, the BankingDB database definition has been updated from when it was used previously. We have now added all primary and foreign keys. We have created indexes on foreign keys. This shows the region and country tables (all the tables are in Appendix B):

```
CREATE TABLE region
(
    region_id             INT        NOT NULL
   ,region                CHAR(40)   NOT NULL
   ,population            INT        NULL
   ,area                  INT        NULL
   ,CONSTRAINT           pk_region PRIMARY KEY
NONCLUSTERED(region_id)
)
GO
```

By default indexes for referential integrity keys are created as clustered. Indexes for tables in this book are created as non-clustered for the purposes of demonstrating optimizer behavior.

```
CREATE TABLE country
(
    country_id            INT        NOT NULL
   ,region_id             INT        NOT NULL
   ,country               CHAR(40)   NOT NULL
   ,code                  CHAR(2)    NOT NULL
   ,population            INT        NULL
   ,area                  INT        NULL
   ,fxcode                CHAR(3)    NULL
   ,currency              CHAR(40)   NULL
   ,rate                  FLOAT      NULL
   ,CONSTRAINT            pk_country PRIMARY KEY
NONCLUSTERED(country_id)
   ,CONSTRAINT fk_country_region FOREIGN KEY(region_id)
                          REFERENCES region
)
GO

CREATE NONCLUSTERED INDEX fkx_country_region ON
country(region_id)
GO
```

The most efficient relational databases will automatically create an index on a primary key because a primary has to be unique. Conversely, an index is not automatically created on a foreign key because a foreign key can be duplicated. A foreign key can even have a NULL value, which if a primary should not be NULL. Serious locking contention issues can result if indexes are not created on foreign keys. Indexing foreign keys tends to prevent gazillions of full table scans between both parent and child tables when referential integrity is checked internally.

There are no indexes to begin with. We will define indexes on the tables as we go along, showing possible improvements. Before we do some experimenting we need to indulge in few a brief words about relational databases, statistics, rule-based optimization, and cost-based optimization.

8.1 Text-based query plans and statistics

The SET SHOWPLAN and SET STATISTICS commands are used for examining query plans and execution details of a query. The difference is distinct between the SHOWPLAN and STATISTICS options. SHOWPLAN describes how the optimizer intends to execute a query (the optimizer might execute the query differently). The STATISTICS actually performs the query and then returns information about how the query executed. So, SHOWPLAN estimates before query execution, and STATISTICS provides a picture of query execution, after execution of the query has completed.

8.1.1 SET SHOWPLAN_TEXT { ON | OFF }

When SET SHOWPLAN_TEXT is set on, information is displayed pertaining to the query execution plan used. The query is not executed. This statement must be the only statement in the query batch.

Suppose we execute the following query when SET SHOWPLAN_TEXT ON has been executed:

```
SELECT * FROM accounts WHERE balance = 0
```

The following output will result:

```
StmtText

select * from accounts where balance = 0
```

```
(1 row(s) affected)

StmtText

   |--Table Scan(OBJECT:([accounts]),
WHERE:([accounts].[balance]=($0.0000)))
```

The output above has been wrapped to fit on the page.

The text of the query is repeated and then information pertaining to the query execution plan is displayed. This contains information such as the logical and physical operators used (described shortly) and other information pertinent to the plan. Since this statement is designed to be used primarily with Microsoft MS-DOS applications, such as the osql command-line utility, we will not spend any more time on it.

8.1.2 SET SHOWPLAN_ALL { ON | OFF }

When SET SHOWPLAN_ALL is set on, detailed information is displayed pertaining to the query execution plan used. The query is not executed. This statement must be the only statement in the query batch.

Suppose we execute the following query when SET SHOWPLAN_ALL ON has been executed:

```
SELECT * FROM customers C INNER JOIN accounts A
   ON C.customer_no = A.customer_no
WHERE balance = 100
```

The output will be returned in the form of a rowset that can be accessed by programs. There is too much information returned to display it across the page, so we will break it down into its constituent parts. Rather than use the previous SQL statement, we will use a slightly more complex one involving an inner join of the Customers and Accounts tables. We are not too concerned with the reason a particular plan was chosen here—the goal of this example is merely to show the output from this SET statement. Ultimately, I find the graphical query execution plan much easier to use, and I will focus on that shortly:

```
StmtText

SELECT * FROM customers C INNER JOIN accounts A ON C.customer_no = A.customer_no WHERE balance =
100
```

```
 |--Bookmark Lookup(BOOKMARK:([Bmk1000]), OBJECT:([BankingDB].[dbo].[customers] AS  [C]))
    |--Nested Loops(Inner Join, OUTER REFERENCES:([A].[customer_no]) WITH PREFETCH)
       |--Table Scan(OBJECT:([BankingDB].[dbo].[accounts] AS [A]), WHERE:([A].[balance]=100.00))
       |--Index Seek(OBJECT:([BankingDB].[dbo].[customers].[nciCustomerNo] AS [C]),
SEEK:([C].[customer_no]=
       [A].[customer_no]) ORDERED FORWARD)
```

I have wrapped the output so this StmtText column can be read completely. This is how it looks with no wrap, so it can easily be matched with the other columns I will discuss. I have had to truncate the text to fit it on the page:

```
SELECT * FROM customers C INNER JOIN accounts A ON C.customer_no = A.customer_no WHERE balance =
100
 |--Bookmark Lookup(BOOKMARK:([Bmk1000]), OBJECT:([BankingDB].[dbo].[customers] AS  [C]))
    |--Nested Loops(Inner Join, OUTER REFERENCES:([A].[customer_no]) WITH PREFETCH)
       |--Table Scan(OBJECT:([BankingDB].[dbo].[accounts] AS [A]), WHERE:([A].[balance]=100.00))
 |--Index Seek(OBJECT:([BankingDB].[dbo].[customers].[nciCustomerNo] AS [C]),
SEEK:([C].[customer_no]=
```

This StmtText column repeats the SQL statement in the first row of the column. Subsequent rows in the display, known as PLAN_ROWS, contain a description of the operation taking place. This column contains the physical operator and may or may not also contain the logical operator. So what are physical and logical operators? The physical operator describes the physical mechanism by which the operation was performed. In our example we can see physical operators such as Nested Loops, Table Scan, and Index Seek. Logical operators describe the relational operation being performed—in our example, an Inner Join. Often, there is no separate logical operator, since the logical operation results in a number of steps—each representing physical operations. In our example, there is no logical operator mentioned in the line that represents the Table Scan physical operation.

Other information is also often present in the StmtText column. In our example, we can see that the row containing the Index Seek physical operator also names the index in question—nciCustomerNo—and the column used in the predicate—customer_no—as well as the table name. The row containing the Nested Loops physical operator also specifies WITH PREFETCH, which means that asynchronous read ahead is being utilized (see Chapter 9). The information in the StmtText column is also repeated in other columns, as we shall now see.

Note that the output is in the form of a hierarchical tree with the SQL statement itself at the top of the tree. I find that decoding the hierarchy can sometimes be confusing, but, again, as we shall see, the graphical query execution plan will help us here. It is often best, however, to start looking at the deepest level in the hierarchy. This represents the basic operations against tables and indexes, which together form the basic building blocks of the query execution plan. Other steps will utilize these basic steps until the result of the query is returned. To assist in understanding the hierarchy, the next set of columns lend a helping hand.

```
StmtId   NodeId   Parent   PhysicalOp        LogicalOp

17       1        0        NULL              NULL
17       3        1        Bookmark Lookup   Bookmark Lookup
17       5        3        Nested Loops      Inner Join
17       7        5        Table Scan        Table Scan
17       8        5        Index Seek        Index Seek
```

The StmtId is a number that identifies the statement in the batch of SQL statements if there is more than one SQL statement in the batch. This groups all the steps together for the one statement. The NodeId is a number that identifies the step in the query execution plan, and the Parent is the node ID of the parent step. Using these numbers, the position of a step in the hierarchical tree can be ascertained. The PhysicalOp and LogicalOp columns contain the physical and logical operators as described above.

```
Argument

BOOKMARK:([Bmk1000]), OBJECT:([BankingDB].[dbo].[customers] AS [C])
OUTER REFERENCES:([A].[customer_no]) WITH PREFETCH
OBJECT:([BankingDB].[dbo].[accounts] AS [A]), WHERE:([A].[balance]=100.00)
OBJECT:([BankingDB].[dbo].[customers].[nciCustomerNo] AS [C]),
OBJECT:([BankingDB].[dbo].[customers].[nciCustomerNo] AS [C]),
SEEK:([C].[customer_no]=[A].[customer_no])
ORDERED FORWARD
```

This column displays extra information concerning the operation, as described previously.

The next set of columns includes the values used by the operator; they are typically columns from a SELECT list or WHERE clause. Internal values may also be represented here. In our example, the * has been expanded to the actual list of columns.

```
DefinedValues
```

```
NULL
[C].[customer_no], [C].[customer_fname],
[C].[customer_lname], [C].[customer_notes]
NULL
[A].[account_no], [A].[customer_no], [A].[branch_no],
[A].[balance], [A].[account_notes]
[Bmk1000]
```

Next we see columns that are concerned with the estimated cost of the query:

```
EstimateRows   EstimateIO     EstimateCPU     AvgRowSize   TotalSubtreeCost

1988.1769      NULL           NULL            NULL         1.4232613
1988.1769      6.2500001E-3   2.1869945E-3    886          1.4230624
1988.1769      0.0            8.3105788E-3    445          1.4146254
1988.1769      0.60027075     0.01378925      435          1.22812
1.0            6.3284999E-3   7.9603E-5       19           0.1661949
```

The EstimateRows column contains the number of rows the query optimizer expects the operator to return. In our example, we are looking at 1,988 rows estimated for all the operators except the Index Seek. The 1,988 estimate comes from the fact that the query optimizer estimates that this number of Account table rows will have a balance of 100. The value of 1 from the index seek indicates that the query optimizer knows that for each row from the Accounts table a maximum of one row can be returned from the Customers table (it has a unique index on the customer_no column).

How many rows are actually returned? How many customer accounts have a balance of exactly 100? The answer in our database is, in fact, zero! The query optimizer estimate is very inaccurate. Why? We shall see shortly!

The EstimateIO column contains the estimated I/O cost for the operator. In our example, the cost estimates are small numbers, so what do the values represent? The numbers are weighted by some undocumented weighting factor. Microsoft does not publish the weighting factor, since they want the ability to adjust it to their heart's desire. This means that it is practically impossible to translate the EstimateIO value into logical reads. However, it is possible to compare these numbers with one another, and we know the lower the number the lower the cost.

The EstimateCPU column contains the estimated CPU cost for the operator. In our example, the cost estimates are again small numbers, and, again, the numbers are weighted by some undocumented weighting factor.

This means that it is not possible to translate the EstimateCPU value into CPU milliseconds. Again, it is possible to compare these numbers with one another, and, again, the lower the number the lower the cost. Using these two estimates we can easily see the most expensive operation in terms of I/O and CPU in a query.

The AvgRowSize is the estimated average row size (in bytes) passing through the operator. In our example, rows from the Accounts table are estimated to be 435 bytes in length. The output of the Index Seek operator is an index entry (key plus pointer) of 19 bytes. Once the Customers table row has been retrieved from the data page (the Index Lookup) and joined with the Accounts table row, the combined size is estimated at 886 bytes.

The TotalSubtreeCost column contains the estimated total cost of the operator and the cost of all its children. This is derived from the EstimateIO and EstimateCPU columns, and, again, some mystery weighting factor is used. This number, though, represents a cost value that combines the I/O and CPU costs and is very useful when looking for the operation in a query that is using the lion's share of the query resource. The OutputList column represents a list of the columns that will be displayed by the query.

```
OutputList

NULL
[C].[customer_no], [C].[customer_fname], [C].[customer_lname], [C].[customer_notes],
[A].[account_no],
[A].[customer_no], [A].[branch_no], [A].[balance], [A].[account_notes]
[Bmk1000], [A].[account_no], [A].[customer_no], [A].[branch_no], [A].[balance],
[A].[account_notes]
[A].[account_no], [A].[customer_no], [A].[branch_no], [A].[balance],
[A].[account_notes]

[Bmk1000]
Warnings                          Type          Parallel      EstimateExecutions

NULL                              SELECT        0             NULL
NULL                              PLAN_ROW      0             1.0
NULL                              PLAN_ROW      0             1.0
NO STATS:([accounts] [customer_no],  PLAN_ROW   0             1.0
 [accounts].[balance])
NULL                              PLAN_ROW      0             1988.1769
```

The Warnings column contains any warning messages issued by the query optimizer for the operation. In our example, the only operation to be associated with a warning is the Table Scan operation, where the

Accounts table is being scanned looking for rows with a balance of 100. We shall look at this warning in the graphical query execution plan shortly, but for now let us just say that the query optimizer is telling us why the estimate of the number of rows returned is so inaccurate—can you guess what the warning means?

The Type column merely flags a row as being the parent row for the query—a SELECT, INSERT, UPDATE, or DELETE, for example, or a row representing an element of the query execution plan—PLAN_ROW.

The Parallel column contains a value of 0 or 1 specifying whether the operator can execute in parallel (1) or not (0).

The EstimateExecutions column is the estimated number of times the operator will execute during the query. In our example, the Table Scan operator will execute once. However, for each row in the Accounts table being scanned, the Customer table will be accessed (it is the inner table in a nested loops join). For this reason, the EstimateExecutions column for the Index Seek operator contains the value 1988.1769.

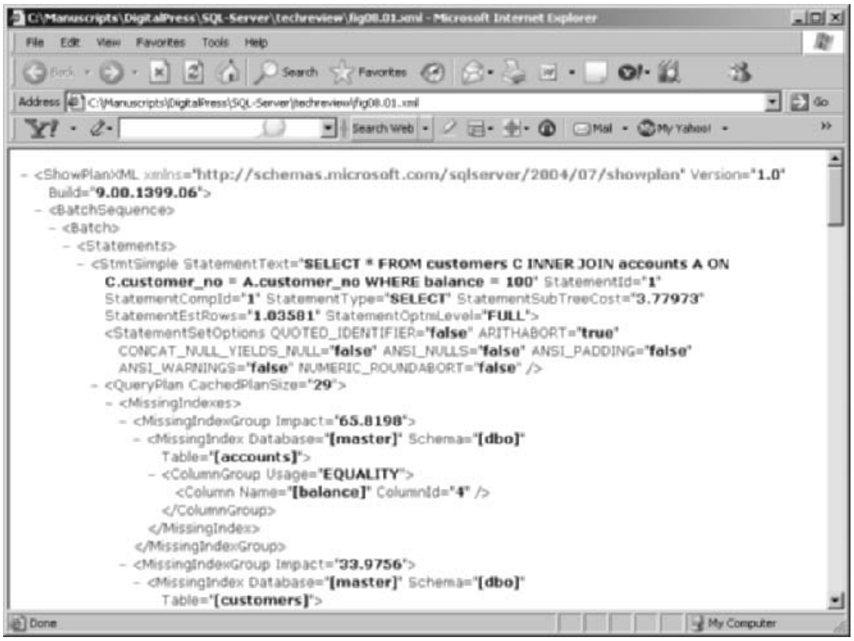
So, as we have seen, the SET SHOWPLAN_ALL statement produces a large amount of information concerning the query execution plan. As I've hinted at a number of times now, I feel this information is best displayed through the graphical query execution plan. Before we take a look at this there are more SET statements that are useful—so let's have a look at them.

8.1.3 SET SHOWPLAN_XML { ON | OFF }

The XML option is new to SQL Server 2005. The only difference with the XML is that the output is returned as a properly structured and formed XML document. So, you execute SET SHOWPLAN_XML ON and then against execute the same query as before:

```
SELECT * FROM customers C INNER JOIN accounts A
   ON C.customer_no = A.customer_no
WHERE balance = 100
```

The query plan output is returned as an XML document, as shown in Figure 8.1.



```
- <ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="1.0"
    Build="9.00.1399.06">
  - <BatchSequence>
    - <Batch>
      - <Statements>
        - <StmtSimple StatementText="SELECT * FROM customers C INNER JOIN accounts A ON
            C.customer_no = A.customer_no WHERE balance = 100" StatementId="1"
            StatementCompId="1" StatementType="SELECT" StatementSubTreeCost="3.77973"
            StatementEstRows="1.03581" StatementOptmLevel="FULL">
            <StatementSetOptions QUOTED_IDENTIFIER="false" ARITHABORT="true"
              CONCAT_NULL_YIELDS_NULL="false" ANSI_NULLS="false" ANSI_PADDING="false"
              ANSI_WARNINGS="false" NUMERIC_ROUNDABORT="false" />
          - <QueryPlan CachedPlanSize="29">
            - <MissingIndexes>
              - <MissingIndexGroup Impact="65.8198">
                - <MissingIndex Database="[master]" Schema="[dbo]"
                    Table="[accounts]">
                  - <ColumnGroup Usage="EQUALITY">
                      <Column Name="[balance]" ColumnId="4" />
                    </ColumnGroup>
                  </MissingIndex>
                </MissingIndexGroup>
              - <MissingIndexGroup Impact="33.9756">
                - <MissingIndex Database="[master]" Schema="[dbo]"
                    Table="[customers]">
```

Figure 8.1
An XML formatted query plan

8.1.4 SET STATISTICS PROFILE { ON | OFF }

The SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL statements we have just looked at both display information concerning the query execution plan adopted by the query optimizer. Neither statement actually allows the query to execute. This has a number of ramifications. Consider the following stored procedure:

```
CREATE PROCEDURE usp_testplan
AS
CREATE TABLE #t1 (c1 int)
SELECT c1 from #t1
RETURN
```

Suppose we now issue a SET SHOWPLAN_ALL ON and execute the stored procedure, as follows:

```
EXEC usp_testplan

Server: Msg 208, Level 16, State 1, Procedure
usp_testplan, Line 4
Invalid object name '#t1'.
```

Because the SET statement suppresses the execution of the stored procedure, the temporary table #t1 is not created, and it is therefore not possible to display plan information for the SELECT statement.

Another problem caused by the SET statement suppressing query execution is that we cannot produce information about the logical reads actually used by the query, nor can we see how many rows pass through an operator as opposed to an estimated number.

Enter SET STATISTICS PROFILE. This statement does not suppress the execution of the query. As well as returning the same information as SET SHOWPLAN_ALL, it also displays two extra columns—Rows and Executes—which contain the actual number of rows returned and the actual number of times the operator executed during the query. In other words, the equivalent of the EstimateRows column and the EstimateExecutions column, respectively.

8.1.5 SET STATISTICS IO { ON | OFF }

Another SET statement that is useful when investigating different query optimizer strategies is SET STATISTICS IO. This displays the count of table accesses (scans), logical and physical reads, and read ahead reads for each Transact-SQL statement, as follows:

```
SET STATISTICS IO ON
SELECT C.customer_lname, A.account_no, A.balance
FROM customers C INNER JOIN accounts A
ON C.customer_no = A.customer_no
WHERE balance BETWEEN 100 AND 120

customer_lname    account_no    balance

Burrows           107540        118.0400
 :
 :

(56 row(s) affected)

Table 'customers'. Scan count 56, logical reads 181,
physical reads 0, read-ahead reads 0.
Table 'accounts'. Scan count 1, logical reads 1569,
physical reads 0, read-ahead reads 0.
```

In the above example, the Accounts table experienced a scan count of 1, and the Customers table experienced a scan count of 56. The phrase scan count has nothing to do with the use of table scans; it merely states how many times the table was accessed in the query. In our example, the Accounts table is processed as the outer table of the (nested loops) join and is therefore accessed only once. For each qualifying row in the Accounts table, the Customers table is accessed. In this example there are 56 qualifying rows in the Accounts table, so the scan count of the Customers table is 56.

There are 1,569 pages in the Accounts table. As this is table scanned, SQL Server 2000 must retrieve every page—hence, the logical read value of 1,569. The Customers table experiences 181 logical reads, approximately three per scan. This is because the index is two levels deep, so two index pages and one data page will be fetched on each scan.

Since the data and index pages are already cached in memory, the physical reads counter is zero. A physical read represents a database page request that is not found in cache, so SQL Server 2000 has to fetch it from disk. Read-ahead reads will be discussed in Chapter 9.

Note: The scan count may sometimes be larger than you expect. For example, you may expect the scan count for a table to be one. However, the query optimizer has created a parallel execution plan and two parallel threads access the table—hence, it has a scan count of two.

8.1.6 SET STATISTICS TIME { ON | OFF }

The SET STATISTICS TIME ON statement displays the time (in milliseconds) that SQL Server took to parse the statement, compile the query optimizer strategy, and execute the statement, as follows:

```
SELECT C.customer_lname, A.account_no, A.balance
FROM customers C INNER JOIN accounts A
ON C.customer_no = A.customer_no
WHERE balance BETWEEN 100 AND 120

SQL Server parse and compile time:
   CPU time = 10 ms, elapsed time = 10 ms.

customer_lname   account_no     balance
                               ------
```

```
Burrows          107540        118.0400
 :
 :

(56 row(s) affected)

SQL Server Execution Times:
   CPU time = 29 ms,  elapsed time = 29 ms.
```

I personally do not use this statement. Whereas logical reads is a constant and will be the same for a given access strategy at any time irrespective of other work on the server, this is not true for the statistics time. For that reason I do not find it very useful. If I really want to compare the elapsed times of queries, I often use my own statements, as follows:

```
DECLARE
  @time_msg CHAR(255),
  @start_time DATETIME
SELECT @start_time = GETDATE()
-- Execute the query we wish to test
SELECT C.customer_lname, A.account_no, A.balance
FROM customers C INNER JOIN accounts A
ON C.customer_no = A.customer_no
WHERE balance BETWEEN 100 AND 120

-- Calculate the query duration

SELECT @time_msg = 'Query time (minutes:seconds) ' +
  CONVERT(CHAR(2),
  DATEDIFF(ss,@start_time,GETDATE())/60) +
  ':' +
  CONVERT(CHAR(2),
  DATEDIFF(ss,@start_time,GETDATE())%60)

print @time_msg

customer_lname   account_no   balance

Burrows          107540       118.0400
 :
 :
```

```
(56 row(s) affected)
Query time (minutes:seconds) 0 :16
```

8.1.7 SET STATISTICS XML { ON | OFF }

Once again, the XML option is new to SQL Server 2005. The only difference with the XML is that the output is returned as a properly structured and formed XML document. The result will be very similar to that shown in Figure 8.1.

As a final note with regard to the SET statements, the Query Analyzer, now part and parcel of SQL Server Management Studio. The Management Studio tends to put all the administration and tuning functionality into a single graphical user interface (GUI). That GUI is intended to make all databases, servers, and associated hardware and software accessible throughout an organization.

8.2 Query plans in Management Studio

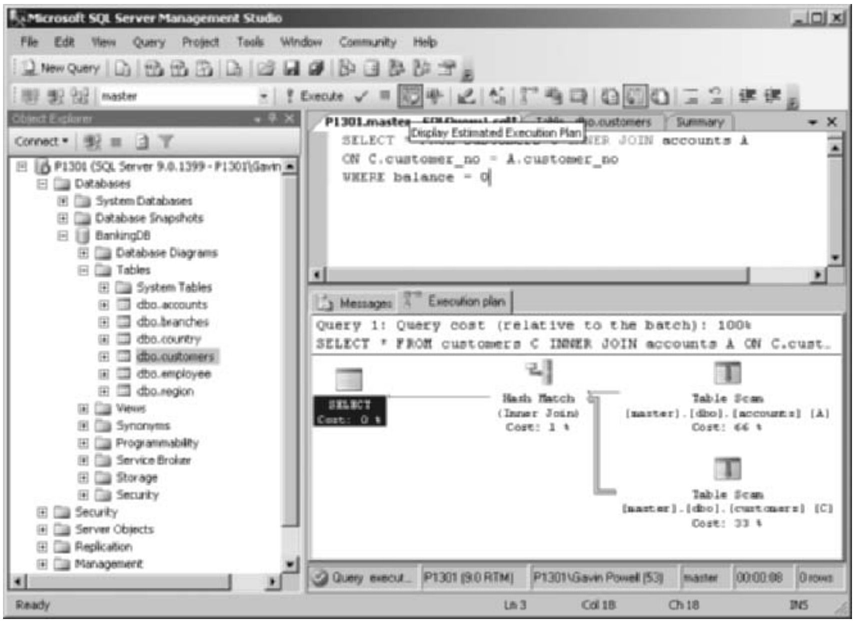
The SQL Server Management Studio in SQL Server 2005 replaces much of the interactive GUI tools in previous versions of SQL Server. Essentially, everything is more or less the same, except that now it's all wrapped up nice and neatly, inside the Management Studio. In short, all tools are accessible from a single interface.

We have been discussing SET statements so far in this chapter that allow us to check the query execution plan that the query optimizer has created. As mentioned on a number of occasions, I find this easier to do with the graphical query execution plan, and this will now be our focus. As with SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL, displaying the estimated execution plan does not cause the query to execute. However, as with SET STATISTICS PROFILE, it is possible to execute the query and view the query execution plan afterwards.

To display the estimated execution plan the keyboard shortcut CTRL+L can be used, or choose Display Estimated Execution Plan from the Query menu. Alternatively, just click the Display Estimated Execution Plan button on the toolbar. Let us take a trip around the graphical display, and then we will look at the graphical query execution plans we might encounter when analyzing our queries. We'll use the inner join we previously used for the SET SHOWPLAN_ALL statement, as follows:

```
SELECT * FROM customers C INNER JOIN accounts A
   ON C.customer_no = A.customer_no
   WHERE balance = 100
```

The estimated execution plan for this statement is shown in Figure 8.2.



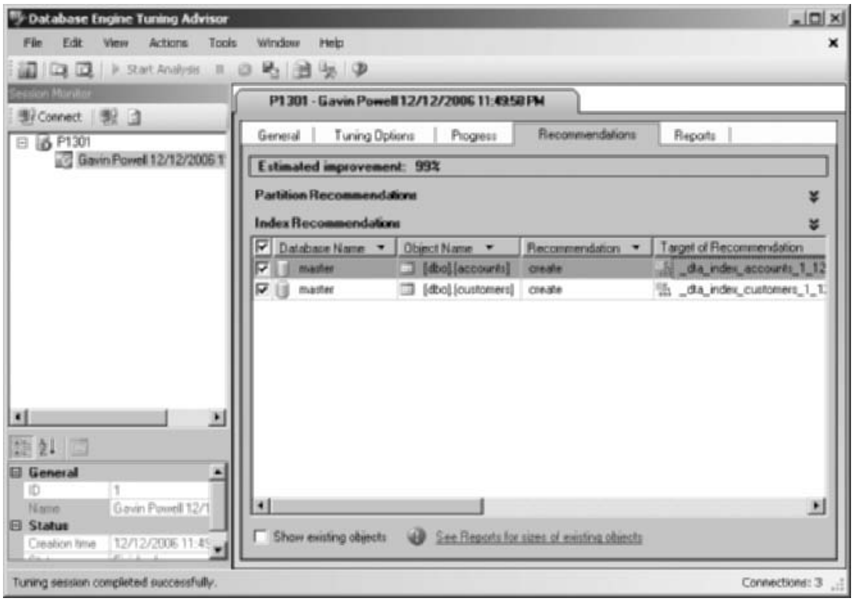
```
SELECT * FROM customers C INNER JOIN accounts A
ON C.customer_no = A.customer_no
WHERE balance = 0
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM customers C INNER JOIN accounts A ON C.cust…
```

Figure 8.2
An estimated execution plan in the Management Studio

Various other options are available across the top of the Management Studio window, at the right of the Display Estimated Execution Plan Option, shown by the rollover yellow text box toward the top of Figure 8.2. These other options include Analyze Query in Database Engine Tuning Advisor, Include Actual Execution Plan, Include Client Statistics, and so on. For example, the query plan shown in Figure 8.2 executes a hash join between the two tables because appropriate indexes are not present. So, Figure 8.3, using the Analyze Query in the Database Engine Tuning Advisor option, shows the obvious recommendation that these indexes should be created.

So, let's now assume the indexes are properly constructed – particularly on primary and foreign keys. The execution plan changes somewhat, as shown in Figure 8.4. The query execution plan is read from right to left. We can see the operators that were rows in the SET SHOWPLAN_ALL output. The hierarchical tree is displayed on its side with the top of the tree on


Database Engine Tuning Advisor
File Edit View Actions Tools Window Help
Start Analysis
Session Monitor
Connect
P1301
Gavin Powell 12/12/2006 1
P1301 - Gavin Powell 12/12/2006 11:49:58 PM
General
Tuning Options
Progress
Recommendations
Reports
Partition Recommendations
Index Recommendations
Show existing objects
See Reports for sizes of existing objects

Estimated improvement: 99%

✓	Database Name	Object Name	Recommendation	Target of Recommendation
✓	master	[dbo].[accounts]	create	_dta_index_accounts_1_12
✓	master	[dbo].[customers]	create	_dta_index_customers_1_1

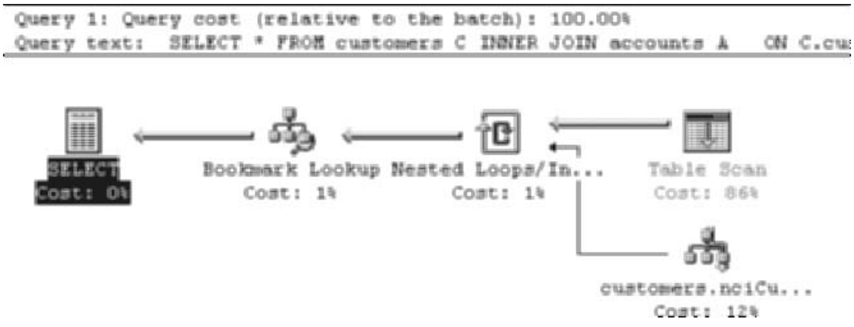
General
ID: 1
Name: Gavin Powell 12/1
Status
Creation time: 12/12/2006 11:4

Tuning session completed successfully.

Connections: 3

Figure 8.3
The Database Engine Tuning Advisor can analyze a query

the left—the SELECT statement. On the far right of the display the children at the lowest branches of the tree are displayed. The children at the same level are displayed vertically above one another. The flow of rows, index pointers, and so on is illustrated by the arrows joining the operators. Note that these arrows vary in width. This variation is proportional to the number of rows passed to the next operator up the tree.



```
Query 1: Query cost (relative to the batch): 100.00%
Query text:  SELECT * FROM customers C INNER JOIN accounts A   ON C.cu
```

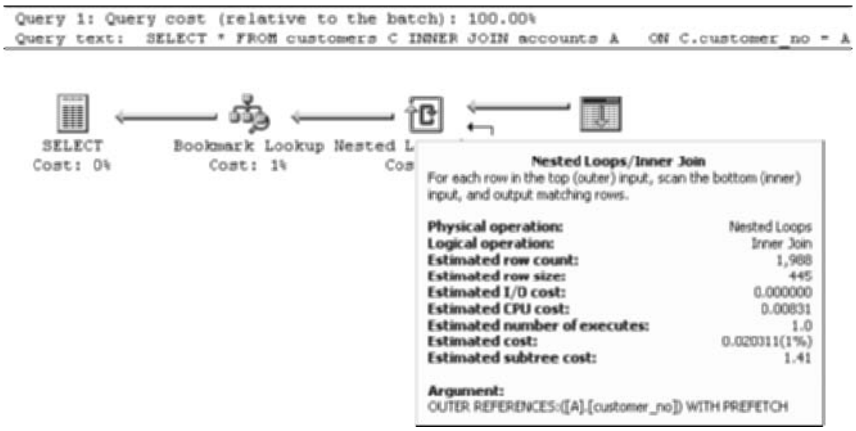
Figure 8.4
A graphical estimated execution plan

At the top of the display shown in Figure 8.4 is a heading specifying that this is query 1. We only have one query in our query batch, but if there were more than one query these would be labeled accordingly. The query text is also displayed. More interestingly, the query optimizer has also estimated the cost of the query relative to the cost of the batch. This is useful when you want to see which query in the batch is the one that is the most expensive. In our example, having only one query, the cost is 100 percent of the batch.

Different operations within a query are also cost checked relative to one another. In our example the cost of the Table Scan is 86 percent of the cost of the query. Clearly, this operator is worthy of some investigation if the query is performing badly.

The operators are named and represented by an icon. In the case of the nested loops join, the icon represents a nested loop, and the name of the physical and logical operators are displayed in the format physical/logical. In the case of the table scan, the physical operator Table Scan is displayed. There is no logical operator as such; therefore it takes the same name as the physical operator and just physical is displayed. In the case of the indexed access of the Customers table, the icon representing an Index Seek is displayed, and the index name in the format table.index is displayed underneath. How do we know what these icons represent? In the background of the display, if an icon or arrow is right-clicked, a menu appears. If Help is chosen, a list of operators appears. Click on an operator and an explanation is displayed. You will also notice that the displayed menu contains options for zooming, fonts, and managing indexes and statistics.

What about the detailed information that was produced by SET SHOWPLAN_ALL? Can the graphical execution plan produce this information also? It can and all we have to do is move the mouse pointer over the operator we are interested in—no click is needed. This is shown in Figure 8.5.



```
Query 1: Query cost (relative to the batch): 100.00%
Query text:  SELECT * FROM customers C INNER JOIN accounts A   ON C.customer_no = A
```

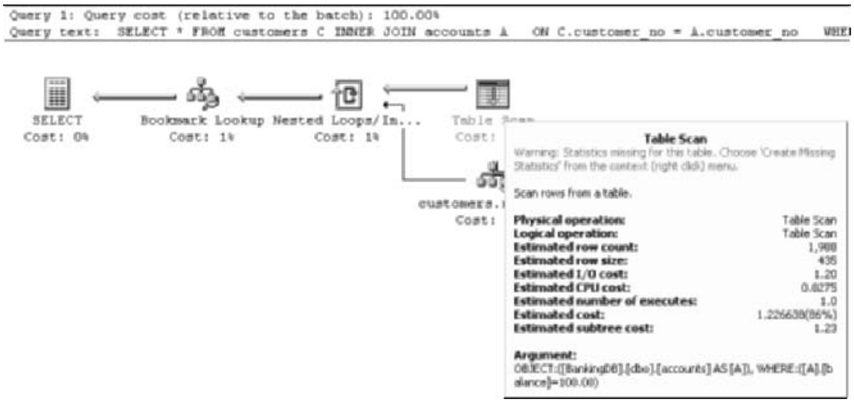
Nested Loops/Inner Join
For each row in the top (outer) input, scan the bottom (inner) input, and output matching rows.

Physical operation:	Nested Loops
Logical operation:	Inner Join
Estimated row count:	1,988
Estimated row size:	445
Estimated I/O cost:	0.000000
Estimated CPU cost:	0.00831
Estimated number of executes:	1.0
Estimated cost:	0.020311(1%)
Estimated subtree cost:	1.41

Argument:
OUTER REFERENCES:([A].[customer_no]) WITH PREFETCH

Figure 8.5
Placing the pointer over an operator

As can be seen, lots of information pertaining to the operator is displayed. Pretty much all the cost information and any other text that was displayed in the SET SHOWPLAN_ALL are displayed in this window. Note what happens when we move the mouse pointer over the Table Scan operator. This is shown in Figure 8.6.



```
Query 1: Query cost (relative to the batch): 100.00%
Query text:  SELECT * FROM customers C INNER JOIN accounts A   ON C.customer_no = A.customer_no   WHE
```

Table Scan

Warning: Statistics missing for this table. Choose 'Create Missing Statistics' from the context (right click) menu.

Scan rows from a table.

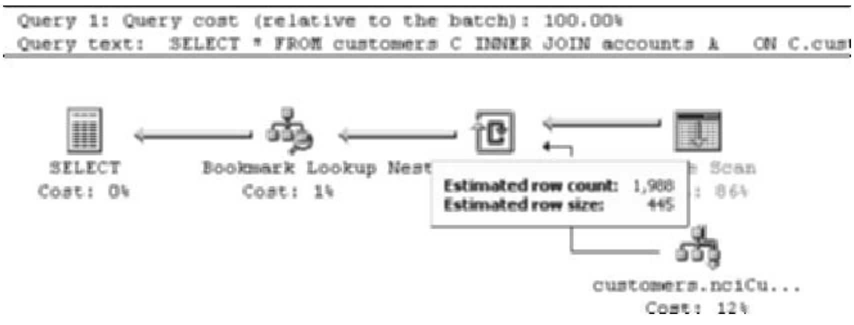
Physical operation:	Table Scan
Logical operation:	Table Scan
Estimated row count:	1,988
Estimated row size:	435
Estimated I/O cost:	1.20
Estimated CPU cost:	0.0275
Estimated number of executes:	1.0
Estimated cost:	1.226638(86%)
Estimated subtree cost:	1.23

Argument:
OBJECT:([BankingDB].[dbo].[accounts] AS [A]), WHERE:([A].[balance]=100.00)

Figure 8.6
Placing the pointer over the Table Scan operator

A warning message is displayed (in red—but you can't see that because this book is printed in black and white so the red looks just a little grayish!) telling us that statistics are missing from the table. If we recall, the SET SHOWPLAN_ALL output also had a warning in the Warnings column of its output for this operator. We'll look at what the warning means shortly, but for now let us just register that the graphical query execution plan displays warnings and, in this case, suggests a course of action. Again, we can't see this, but on the graphical display shown in Figure 8.4, the Table Scan Cost 86 is percent—also displayed in red to draw our attention to the fact that this operator has warnings associated with it.

If the mouse pointer is placed over an arrow, a window pops up—as shown in Figure 8.7.



```
Query 1: Query cost (relative to the batch): 100.00%
Query text:  SELECT * FROM customers C INNER JOIN accounts A   ON C.cus
```

Estimated row count:	1,988
Estimated row size:	445

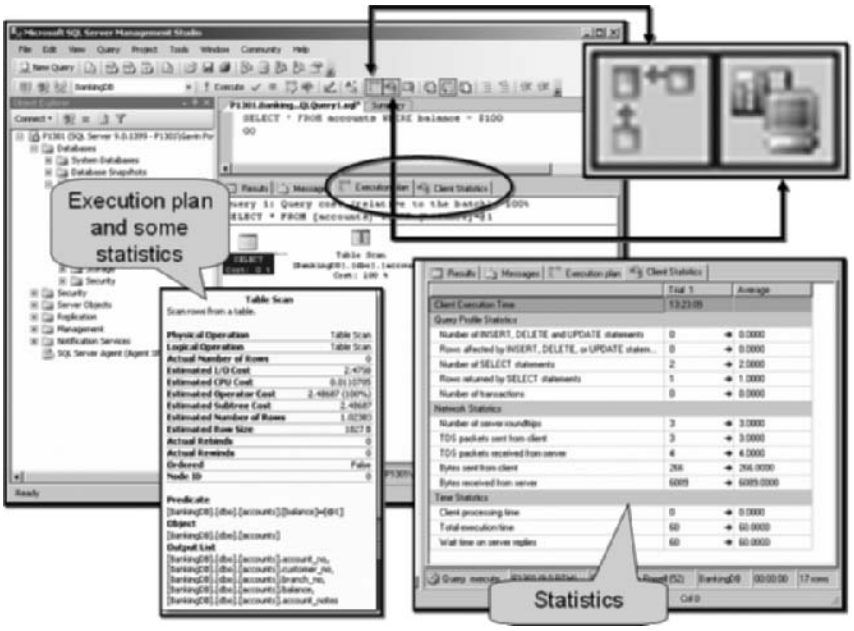
Figure 8.7
Placing the pointer over an arrow

This window displays information about the estimated number of rows being passed to the next operator and the estimated row size.

Now that we know the format of the Display Estimated Execution Plan window, we can investigate some query optimizer strategies. These strategies will be examples of the query optimizer and index behavior we have discussed in this and the previous chapter. We will start with simple examples and then move to more complex examples.

8.2.1 Statistics and cost-based optimization

In general, in SQL Server 2005, the statistical picture is quite different to that of SQL Server 2000. In Figure 8.8, statistics are already present—and I did nothing to create them, but simply created tables and added rows using simple INSERT statements. In fact, this was also the case in SQL Server 2000, except that in SQL Server 2005, it appears to be much more difficult to completely switch off the automation of statistics generation, if not impossible.


Microsoft SQL Server Management Studio
Execution plan and some statistics
Statistics

```
SELECT * FROM accounts WHERE balance = $100
GO
```

Table Scan
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Number of Rows	0
Estimated I/O Cost	2.4758
Estimated CPU Cost	0.0110795
Estimated Operator Cost	2.48687 (100%)
Estimated Subtree Cost	2.48687
Estimated Number of Rows	1.02383
Estimated Row Size	1827 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Predicate
[BankingDB].[dbo].[accounts].[balance]=[@1]
Object
[BankingDB].[dbo].[accounts]
Output List
[BankingDB].[dbo].[accounts].account_no,
[BankingDB].[dbo].[accounts].customer_no,
[BankingDB].[dbo].[accounts].branch_no,
[BankingDB].[dbo].[accounts].balance,
[BankingDB].[dbo].[accounts].account_notes

	Trial 1	Average
Client Execution Time	13:23:09	
Query Profile Statistics		
Number of INSERT, DELETE and UPDATE statements	0	0.0000
Rows affected by INSERT, DELETE, or UPDATE statem...	0	0.0000
Number of SELECT statements	2	2.0000
Rows returned by SELECT statements	1	1.0000
Number of transactions	0	0.0000
Network Statistics		
Number of server roundtrips	3	3.0000
TDS packets sent from client	3	3.0000
TDS packets received from server	4	4.0000
Bytes sent from client	266	266.0000
Bytes received from server	6009	6009.0000
Time Statistics		
Client processing time	0	0.0000
Total execution time	60	60.0000
Wait time on server replies	60	60.0000

Figure 8.8
SQL Server 2005 execution plans with statistics

As shown in Figure 8.8, statistics and execution plan information are readily and very easily available in the SQL Server 2005 Management Studio interface. The two buttons at the top of the main window display execution plan and/or statistics—either or both selected by the buttons on the out frame of the main window (the buttons marked Execution Plan and Client Statistics).

In SQL Server 2000, the existence of statistics was somewhat dependent on the existence of indexes. In the database that produced the output shown in Figure 8.8, there are no indexes on the accounts table. There are actually no indexes in the BankingDB database at all (at this point). Statistics are not dependent on indexes in SQL Server 2005. This is because optimization will often read an entire table without even reading any indexes at all. Sometimes reading a table and ignoring its indexes is more efficient. This can occur when the entire table is read or the table is very small.

This leads to a current trend in relational database technology, the replacement of rule-based optimization with that of cost-based optimization. Optimization makes a best guess at the best way to execute a query.

Note: Some may say that an optimizer is in reality an expert system for query analysis. It uses a lot of information, including statistics, in order to formulate access plans. Then again, an expert system is rule-based. Thus a optimization is not an expert system because statistics are sampled measurements of reality, rather than coded rules based on the knowledge of an expert.

An optimizer is a program running inside a relational database. Rule-based optimization does not use statistics but a set of rules. In some environments rule-based guessing at the best way to execute a query can function well—but most often rule-based optimization is pretty hopeless. The most recent versions of relational databases allow only cost-based optimization. Rule-based optimization has been discarded. Cost-based optimization uses statistical measurements of actual data spaces in a database, to guess at efficient methods of getting at that data—at any particular time. So, a cost-based optimization guess may not always be the same for a particular query—depending on all sorts of things, such as database size, current activity, and so on.

Note: Rule based optimization is a thing of the past in SQL Server database, as with many other relational database vendors. Why? Because cost-based optimization is so much more efficient and is thus much better for performance.

So, you can't really switch statistics off completely. It is also inadvisable, even for the purposes of demonstration. This is because cost based optimization is based purely on statistics. Even when statistics are not present, they will be generated on-the-fly as sampling of larger sets—aiding in cost based optimization.

Now let's go through some example queries to demonstrate graphical query plans and statistics used in the SQL Server 2005 Management Studio. One way to do this is to go back to Chapter 6 and show query plans for queries (or similar queries) as described. Perhaps some improvements can be made with indexes and otherwise.

In this first query we execute a simple query that finds all fields and records in the accounts table:

```
SELECT * FROM accounts
```

Before executing the above query, let's briefly change the primary key from a non-clustered index to the clustered index form:

```
ALTER TABLE accounts DROP CONSTRAINT pk_accounts
GO
ALTER TABLE accounts ADD CONSTRAINT pk_accounts
    PRIMARY KEY CLUSTERED(account_no)
GO
```

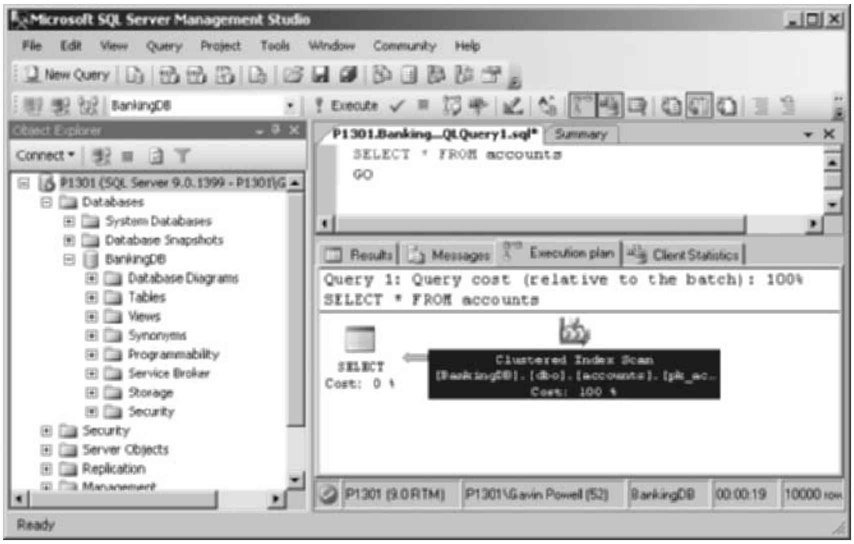
The above change does not need to be scripted. It can also be executed in the Management Studio.

Note: The default for primary key indexes in SQL Server Management Studio is creation as a clustered index. All primary keys have indexes created as non-clustered indexes at this point in this book (see Appendix B).

The result in Figure 8.9 is interesting because the clustered primary key index is read, as opposed to the table as a full scan. The reason for this is simple: a clustered index is made up of all the columns in a table by default.

Now let's change the primary key from a clustered index back to a non-clustered index:

```
ALTER TABLE accounts DROP CONSTRAINT pk_accounts
GO
ALTER TABLE accounts ADD CONSTRAINT pk_accounts
   PRIMARY KEY NONCLUSTERED(account_no)
GO
```

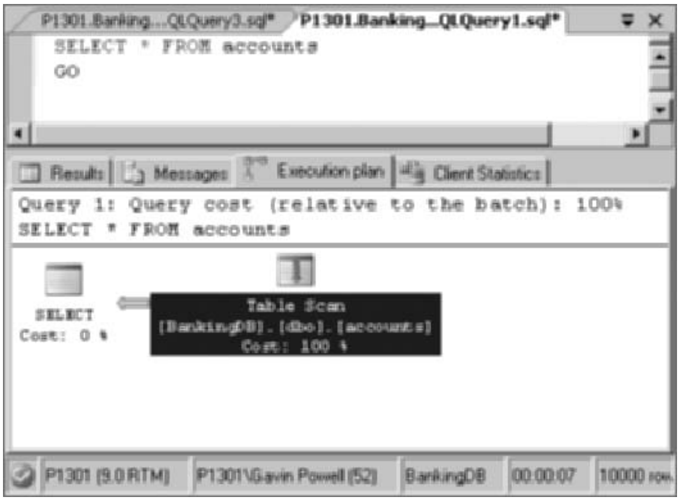


```
SELECT * FROM accounts
GO
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM accounts
```

Figure 8.9 *Reading everything from a clustered index*

Figure 8.10 shows the obvious result, which is a full scan of the entire table. There is no other alternative for the optimizer to choose.



```
SELECT * FROM accounts
GO
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM accounts
```

Figure 8.10 *A non-clustered primary key forces a full table scan*

Now let's try another query:

```
SELECT account_no, customer_no, branch_no,
    balance, account_notes FROM accounts
```

The result is shown in Figure 8.11. We substitute the individual column names for the asterisk. There is no difference in performance between using the star character (*) or explicitly naming the column. Technically speaking, explicitly naming columns might speed up the query a little because it removes the need for the query to search through metadata in order to find field names. Perhaps more importantly, explicitly listing field names might help to avoid unnecessary reading of large objects, which are not contained within the regular field structure of a table, such as externally stored images.

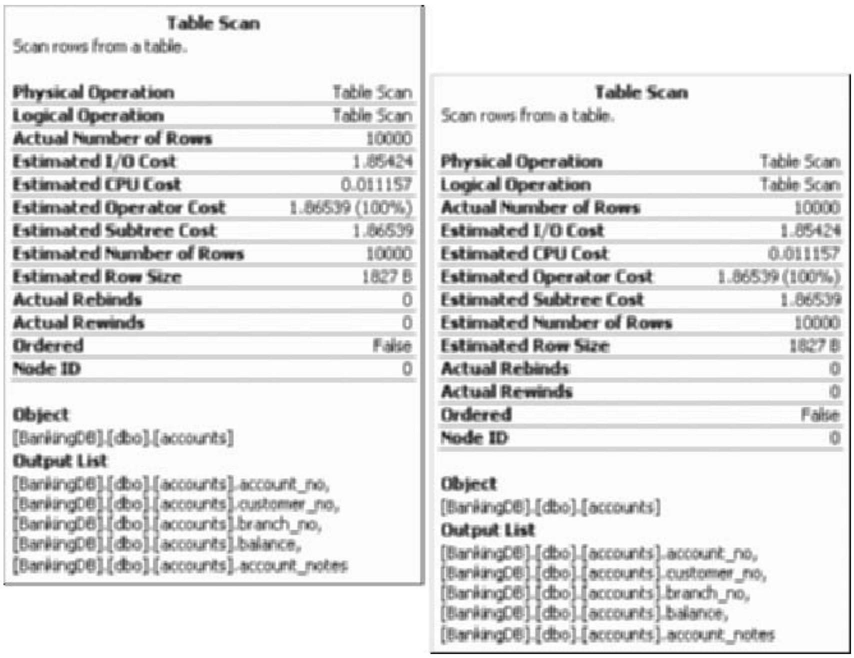


Table Scan
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Number of Rows	10000
Estimated I/O Cost	1.85424
Estimated CPU Cost	0.011157
Estimated Operator Cost	1.86539 (100%)
Estimated Subtree Cost	1.86539
Estimated Number of Rows	10000
Estimated Row Size	1827 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Object
[BankingDB].[dbo].[accounts]
Output List
[BankingDB].[dbo].[accounts].account_no,
[BankingDB].[dbo].[accounts].customer_no,
[BankingDB].[dbo].[accounts].branch_no,
[BankingDB].[dbo].[accounts].balance,
[BankingDB].[dbo].[accounts].account_notes

Table Scan
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Number of Rows	10000
Estimated I/O Cost	1.85424
Estimated CPU Cost	0.011157
Estimated Operator Cost	1.86539 (100%)
Estimated Subtree Cost	1.86539
Estimated Number of Rows	10000
Estimated Row Size	1827 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

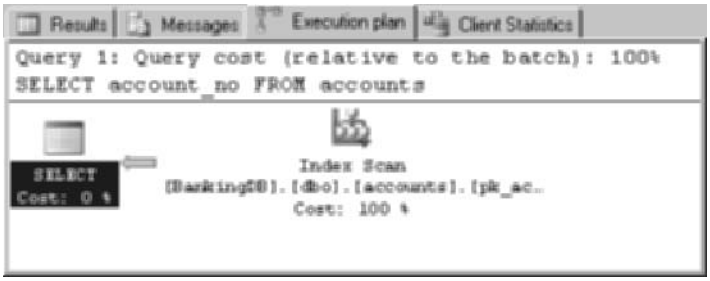
Object
[BankingDB].[dbo].[accounts]
Output List
[BankingDB].[dbo].[accounts].account_no,
[BankingDB].[dbo].[accounts].customer_no,
[BankingDB].[dbo].[accounts].branch_no,
[BankingDB].[dbo].[accounts].balance,
[BankingDB].[dbo].[accounts].account_notes

Figure 8.11 *Reading all fields explicitly can be faster, but is not apparent*

In the next query, I select the account_no column only, which also happens to be the non-clustered primary key:

```
SELECT account_no FROM accounts
```

As shown in Figure 8.12, the optimizer selects to read the primary key index and ignores the table. It is more efficient to read the index because only the indexed column is read. Of course, because of previous queries all the records in the table, and may all be loaded into RAM, the table could still be read as a full table scan—but not in this situation.

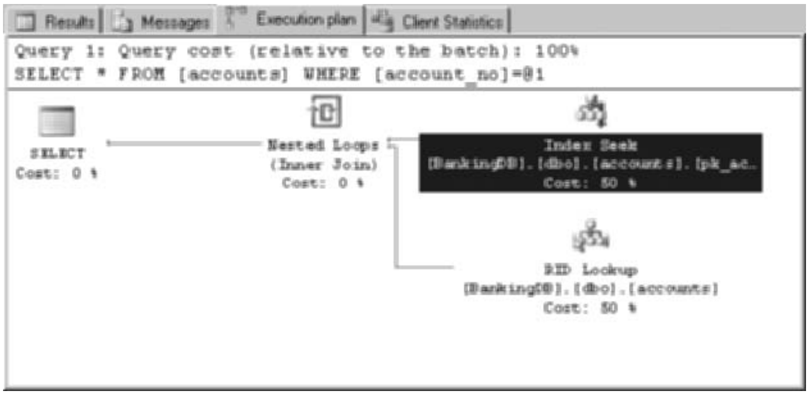


```
Query 1: Query cost (relative to the batch): 100%
SELECT account_no FROM accounts
```

Figure 8.12
Reading a single indexed column read only the index

In the next query all the columns are selected but a single row is retrieved using a WHERE clause. The result is shown in Figure 8.13:

```
SELECT * FROM accounts WHERE account_no = 1
```



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [accounts] WHERE [account_no]=@1
```

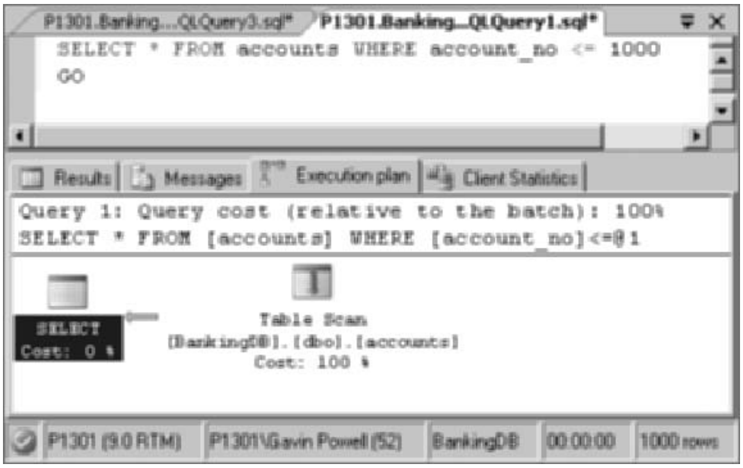
Figure 8.13
A small number of filtered rows reads the index and table

The next query pulls more than 1 row but still much less than the entire table. The result is the same as for the previous query, reading index and table as shown in Figure 8.13:

```
SELECT * FROM accounts WHERE account_no <= 100
```

The next query defaults to a full table scan, avoiding the index. This is because the optimizer thinks that reading the entire table is faster than reading the index and table, for that many rows. The result is shown in Figure 8.14:

```
SELECT * from accounts WHERE account_no <= 1000
```



```
SELECT * FROM accounts WHERE account_no <= 1000
GO
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [accounts] WHERE [account_no]<=@1
```

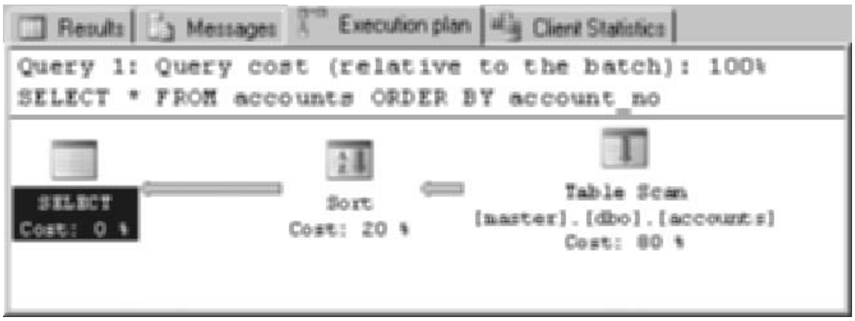
Figure 8.14 *Increase row count and index + table scan becomes full table scan*

So, based on previous queries, it is highly likely that the following types of queries will all read the accounts table, ignoring the non-clustered index:

```
SELECT * FROM accounts WHERE account_no BETWEEN 1 AND 3000
SELECT * FROM accounts WHERE account_no != 100
SELECT * FROM customers WHERE name LIKE '%a%'
SELECT * FROM accounts WHERE account_no NOT LIKE '%a%'
```

The next example uses an ORDER BY clause to sort rows. Figure 8.15 shows a sort applied after the full table scan:

```
SELECT * FROM accounts ORDER BY account_no
```

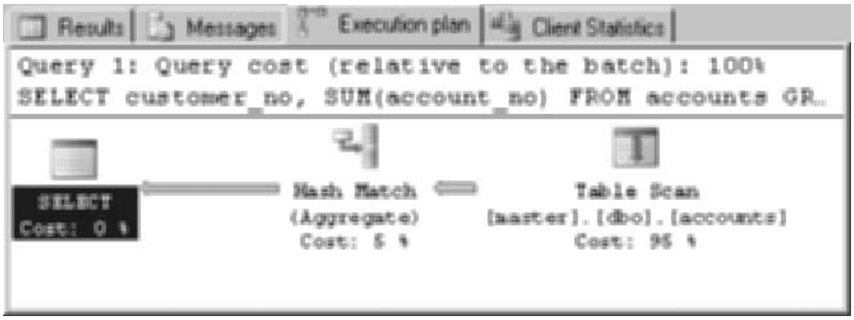


```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM accounts ORDER BY account_no
```

Figure 8.15 *The ORDER BY clause forces a sort after reading the table*

The next query finds all records. It then uses a GROUP BY clause to merge rows into aggregates, based on the customer-no. This, of course, assumes that customers have more than one account with a single branch. The result is show in Figure 8.16:

```
SELECT customer_no, SUM(account_no)
FROM accounts GROUP BY customer_no
```



```
Query 1: Query cost (relative to the batch): 100%
SELECT customer_no, SUM(account_no) FROM accounts GR...
```

Figure 8.16 *GROUP BY matches common values into multiple groups*

8.3 Hinting to the optimizer

As we have already seen, the query optimizer is a sophisticated piece of software that can consider multiple factors and create the most efficient query plan. However, there will be situations when you may wish to force the query optimizer to create a plan that it would not otherwise have chosen. Perhaps what it considers the most efficient plan is not really the case in some specific situations that you understand well. As we shall now see, it is possible to override the query optimizer, but this should be considered only as a last resort. Perhaps rewriting the query or changing the index design strategy might be a better long-term option.

The query optimizer can be overridden by using a query optimizer hint. These hints can be grouped into four categories:

- Join hints
- Table hints
- View hints
- Query hints

8.3.1 Join hints

Join hints are used to force the query optimizer to create a query plan that adopts a particular join technique when joining two tables. We already know that there are three join techniques available; these are nested loops, merge, and hash. We can specify a join hint, which will force two tables to be joined using one of these techniques. A fourth join hint, REMOTE, can also be specified to dictate on which server a join is to be performed in a distributed join query.

The join hint syntax is simple to use; the join type is specified as part of the join, as follows:

```
SELECT * FROM accounts INNER HASH JOIN customers
ON accounts.customer_no = customers.customer_no
WHERE balance > 9990
```

In the above example, a hash join technique is forced.

The REMOTE join hint dictates that the join operation is performed on the server hosting the right table. This is useful when the left table is a local table with few rows and the right table is a remote table that has many rows, since this may avoid a lot of data being shipped to the local server.

8.3.2 Table and index hints

Table hints are very useful, since they dictate the access method to use when retrieving data from a table. This can be a table scan, a single index, or multiple indexes. An example of the syntax used is as follows:

```
SELECT * FROM accounts WITH (INDEX (nciBalance))
 WHERE balance BETWEEN 100 AND 200
 AND customer_no BETWEEN 1000 AND 2000
```

The above example forces the query optimizer to adopt a plan that uses the non-clustered index nciBalance to access the Accounts table. The following example forces the query optimizer to adopt a plan that uses the non-clustered indexes nciBalance and nciCustomerNo to access the Accounts table—in other words, to perform an index intersection:

```
SELECT * FROM accounts WITH (INDEX (nciBalance,
nciCustomerNo))
```

```
 WHERE balance BETWEEN 100 AND 200
 AND customer_no BETWEEN 1000 AND 2000
```

Suppose a table scan must be forced. The following syntax forces the query optimizer to adopt a plan that uses a table scan if there is no clustered index present on the table or that uses a clustered index scan if there is:

```
SELECT * FROM accounts WITH (INDEX (0))
 WHERE balance BETWEEN 100 AND 200
 AND customer_no BETWEEN 1000 AND 2000
```

If there is a clustered index present on the table, a clustered index scan or seek can be forced, as shown in the following example:

```
SELECT * FROM accounts WITH (INDEX (1))
 WHERE balance BETWEEN 100 AND 200
 AND customer_no BETWEEN 1000 AND 2000
```

Another table hint that we have briefly discussed is FASTFIRSTROW. As mentioned in our previous discussion concerning ORDER BY, in the case of a table scan and sort no rows will be returned until the result set has been sorted. If the non-clustered index is chosen by the query optimizer, the first row can be returned immediately. This behavior can be forced with the FASTFIRSTROW query optimizer hint, as follows:

```
SELECT customer_no, balance FROM accounts
   WITH (FASTFIRSTROW)
   ORDER BY customer_no
```

8.3.3 View hints

View hints are similar to table hints but are used with indexed views. The only view hint is NOEXPAND, which forces the query optimizer to process the view like a table with a clustered index. The index on the view which should be used may be specified. An example of the syntax used is as follows:

```
SELECT * FROM BranchTotalFunds
   WITH (NOEXPAND,INDEX (nciTotalBal))
   WHERE TotalBalance > $1350000.00
```

A query hint, described in the following section, can be used to expand the indexed view.

8.3.4 Query hints

A query hint is used throughout the whole query. Query hints can be used to specify many plan behaviors. For example, the following query hint forces the query optimizer to use hashing when calculating an aggregate:

```
SELECT branch_no, SUM(balance)
FROM accounts
GROUP BY branch_no
OPTION (HASH GROUP)
```

If an ordering (sorting) rather than a hashing technique should be used, then this can be forced as follows:

```
SELECT branch_no, SUM(balance)
FROM accounts
GROUP BY branch_no
OPTION (ORDER GROUP)
```

A query hint can be used to force the query optimizer to adopt different techniques when performing UNION operations. The following example will force the use of a Concatenation operator to perform the union, and thus a Sort/Distinct operator will subsequently eliminate the duplicate rows if any:

```
SELECT * FROM AccountsEurope WHERE balance > 9990
UNION ALL
SELECT * FROM AccountsUSA WHERE balance > 9990
OPTION (CONCAT UNION)
```

The following example will force the use of a Hash/Union operator to perform the union, and thus a Sort/Distinct operator will not be needed to eliminate the duplicate rows:

```
SELECT * FROM AccountsEurope WHERE balance > 9990
UNION ALL
SELECT * FROM AccountsUSA WHERE balance > 9990
OPTION (HASH UNION)
```

Finally, the following example will force the use of a Merge/Union operator to perform the union, and thus a Sort/Distinct operator will not be needed to eliminate the duplicate rows. Normally, the Merge/Union operator would exploit the sorted order of the inputs in a manner similar to a merge join, as follows:

```
SELECT * FROM AccountsEurope WHERE balance > 9990
UNION ALL
SELECT * FROM AccountsUSA WHERE balance > 9990
OPTION (MERGE UNION)
```

We have already seen that a JOIN clause can include a join hint. The join hint is relevant to the two tables being joined by that particular join operator. The type of join may also be specified as a query hint, in which case the join type will be applied to all the joins in the query, as follows:

```
SELECT * FROM accounts
   INNER JOIN customers
   ON accounts.customer_no = customers.customer_no
   INNER JOIN branches
   ON accounts.branch_no = branches.branch_no
   WHERE balance > 9990
   OPTION (HASH JOIN)
```

Note that a join hint will override the query hint.

To force a query plan to deliver the first rows quickly, perhaps at the expense of the whole query, the FAST query hint can be used, as in the following example:

```
SELECT customers.customer_no, customer_lname, balance
   FROM customers INNER JOIN accounts
   ON customers.customer_no = accounts.account_no
   WHERE customers.customer_no > 12400
   ORDER BY customers.customer_no
   OPTION (FAST 10)
```

This query hint will force the query optimizer to create a plan that will be optimized to return the first ten rows.

Perhaps a more practical hint is one that can force the query optimizer to change the join order to that specified by the query syntax, as follows:

```
SELECT customers.customer_no, customer_lname, balance
   FROM customers INNER JOIN accounts
   ON customers.customer_no = accounts.account_no
   WHERE accounts.balance BETWEEN 100 AND 200
   OPTION (FORCE ORDER)
```

In the above example, the outer table will become the Customers table even though it is the Accounts table that is filtered.

The next query hint is used to specify the number of CPUs used to parallelize the query on a multiprocessor computer. If there is only one processor, this hint is ignored. The following hint limits the number of CPUs that can be used for parallelism to two:

```
SELECT branch_no, SUM(balance)
FROM accounts
GROUP BY branch_no
OPTION (MAXDOP 2)
```

If MAXDOP is set to 1, parallel query plan is suppressed.

The KEEP PLAN and KEEPFIXED PLAN options are similar in that they control when query plans are recompiled. This is discussed later in the chapter. The KEEPFIXED PLAN option ensures that the query optimizer does not recompile a query due to changes in statistics or to the indexed column. A query will only be recompiled if the table schema changes or sp_recompile is executed specifying the table. The KEEPPLAN option is used to reduce the recompilation thresholds, which determine how many inserts, deletes, and index column updates cause a query to be recompiled. The recompilation thresholds used for querying temporary tables in a stored procedure are less than those for a permanent table, and therefore this option is useful when it is necessary to reduce stored procedure recompilations for stored procedures that use temporary tables. This is discussed later in the chapter.

The EXPAND VIEWS option is used with indexed views. This option effectively ensures that the indexes on an indexed view are not used. The view is expanded into its definition, which is the traditional behavior with non-indexed views, as follows:

```
SELECT * FROM BranchTotalFunds
   WHERE TotalBalance > $1350000.00
   OPTION (EXPAND VIEWS)
```

The ROBUST PLAN option ensures that the query plan will not fail due to size limitations when the maximum row sizes are used in the query. For example, plan A may be more efficient than plan B. However, due to the fact that plan A uses intermediate tables to store intermediate results, if any of the variable-length rows used in the query are at their maximum size, the use of the intermediate tables will fail due to size limitations. The ROBUST PLAN option will ignore plan A and choose plan B, which, although less efficient, will not have the same potential problems due to the way the plan executes—perhaps it does not use intermediate storage of results.

As a final note on hints, hints are not always the most prudent option. It is always best to attempt to get as much out of a query as possible, using just the query. Leave the hint for later. Then again, utilizing a hint can be a useful short-cut to resolve a specific performance issue with a query. The only danger with using hints is that in the past, hints were provided in database engines such as SQL Server and Oracle, merely as suggestions. In other words, the optimizer had the option of ignoring hints. The general trend in relational database technology is that hints are now more a forced rather than a suggested influence on an optimizer. SQL Server appears to be no exception to this general trend. So, use hints as a last resort. I myself have never used hints when tuning queries. I do sometimes find it useful to use hints in order to test assumptions, such as why a specific index is not used as I would expect. Also, if you take a very close look at the buttons at the top of the Management Studio, you may notice one of those little rollover text boxes states *Include Actual Execution Plan*. The use of the word *Actual* is significant. Why? Because there is s proposed or estimated execution plan, which is the one the optimizer figures out from things like statistics, appropriate indexes, hints, and a host of other things. However, the actual execution might be very different to the estimated execution plan. In actuality, hints could be ignored, statistics could be ignored, and indexes could be completely useless. What happens in reality is not necessarily what you might always expect, or even what the optimizer tells you to expect. In short, hints can be very useful in trying to predict the expected, because by hinting at the query to do one thing, and something else happens entirely, then you are made clearly aware that something is odd with your query. Keep an open mind because the unexpected can happen.

8.4 Stored procedures and the query optimizer

Stored procedures are found everywhere in SQL Server. There are many system-stored procedures, and a typical SQL Server development department will also create and use many stored procedures. There are a number of benefits to using stored procedures, such as the following:

- Function encapsulation
- Security
- Performance

By function encapsulation I mean that complex logic can be placed into a stored procedure and hidden from the client software, which then only has to call the stored procedure, passing appropriate parameters. The stored procedure logic can be changed, perhaps to encompass a database modification, without having to change client application software or at least minimizing any change. We can say that stored procedures insulate the client application software from the database structure.

Many sites take a stance that updates to database data can only be made through stored procedures and cannot be made directly to the tables by the client issuing Transact-SQL statements. This model of processing is shown in Figure 8.17.

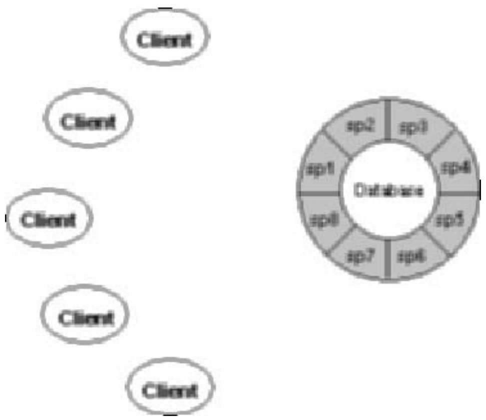


Figure 8.17 *Insulating clients from the database structure via stored procedures*

Even without the use of stored procedures, many clients are insulated from the database, or at least the database is insulated from clients, because custom applications execute SQL code from the applications. Do you really want 1 million and 1 Internet users running all sorts of things directly onto your database? Even with the potential performance nightmare, the lack of security of sensitive data makes this impractical.

This brings us to the second benefit of stored procedures: security. Taking the model shown in Figure 8.17, we can see that in order to implement it, we need a security mechanism that allows us to prohibit client software from directly accessing tables and other objects but allows indirect access in a way that we can define and control. Stored procedures provide this benefit by means of ownership chains.

As long as the owner of the stored procedure is the owner of all the objects referenced by the stored procedure, then execute access on that stored procedure can be granted to database users. They can perform all of the actions defined in the stored procedure even though they have no direct access to the underlying objects. For example, a database user may be granted execute access to a stored procedure that deletes from one table and inserts into another. As long as the ownership of the stored procedure and tables is the same, the database user needs no permissions on the tables.

The most important benefit of stored procedures from the perspective of this book is performance, and it is this aspect of stored procedures on which we will now concentrate. Generally speaking, stored procedures save us the time and effort spent syntax checking Transact SQL and optimizing it. They reduce network load because they minimize the amount of traffic sent to and from the server.

The stages in stored procedure processing are shown in Figure 8.18. This figure can be compared with Figure 7.1, which shows the stages in query processing. The principal difference is that when a Transact-SQL query is submitted, all the above phases are performed. If the query is submitted 100 times, these phases are performed for each submission unless the query plan for the statement can be reused. We will discuss plan reuse for statements later.

With a stored procedure, the query plan is generally always reused—with a few exceptions. When the stored procedure is initially created, the syntax is checked, and, if correct, the stored procedure code is stored in the syscomments system table, which is resident in every database. Also, the stored procedure name is stored in the sysobjects system table, which is also resident in every database.

When a stored procedure is first executed after SQL Server starts (in other words it is not cached in any way), the stored procedure is retrieved

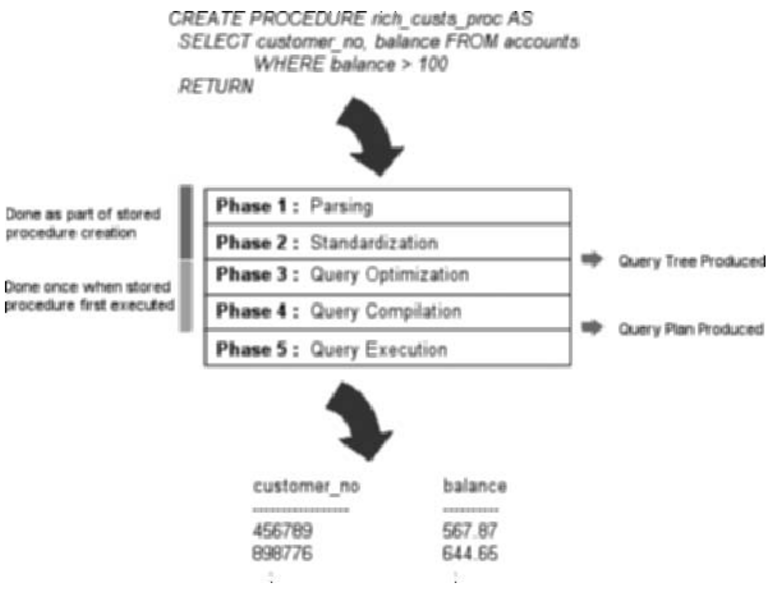

CREATE PROCEDURE rich_custs_proc AS
SELECT customer_no, balance FROM accounts
WHERE balance > 100
RETURN
Done as part of stored procedure creation
Done once when stored procedure first executed
Phase 1 : Parsing
Phase 2 : Standardization
Phase 3 : Query Optimization
Phase 4 : Query Compilation
Phase 5 : Query Execution
Query Tree Produced
Query Plan Produced
customer_no
balance
456789
567.87
898776
644.65

Figure 8.18 *Stages in stored procedure processing*

from syscomments. We can see that we immediately have a performance gain, since we do not have to perform the syntax checking, which, for a large stored procedure, may be nontrivial. Existence checking must be performed at this point, since SQL Server allows us to create the stored procedure even if tables and views, which are referenced in it, do not exist at creation time. This is known as delayed name resolution.

Assuming all the objects referenced exist, the query optimizer creates a query plan for the Transact SQL in the stored procedure and compiles it into executable code. Once the query plan has been created and compiled, it is cached in an area of memory known as the procedure cache. It is then available for the next user.

If another user wishes to execute the stored procedure, SQL Server can now skip the above phases, since the query plan is ready and waiting in the cache. This can increase the performance benefit of the stored procedure quite substantially. How useful the performance advantage of skipping these phases is depends on how long it takes to perform these phases relative to the execution time of the stored procedure and how often the stored procedure is executed. For a complex stored procedure, which is frequently executed, the performance advantage is significant.

Microsoft recommends that the owner name be specified when a stored procedure is executed. This enables SQL Server to access the execution plan for the specific procedure more efficiently. Therefore, it is better to issue:

```
EXEC dbo.usp_GetAuthors
```

than:

```
exec usp_GetAuthors
```

The query plan of a stored procedure can be utilized by many users at the same time. The stored procedure is effectively split into a read only section, which many users can share, and multiple sections, which are private to a user. They are reusable but cannot be shared simultaneously between users. These sections can be used, for example, to hold a user's read/write variables. This is known as an execution context. This approach means that the bulk of the stored procedure plan, the executable code, is held in cache as a single copy. Actually, even this is not quite true. Two copies of the plan may be held on a multiprocessor computer: a nonparallel plan and a parallel plan.

8.4.1 A stored procedure challenge

There is one disadvantage to the stored procedure mechanism compared with executing Transact-SQL queries outside of a stored procedure. Suppose we execute the following query outside of a stored procedure, assuming that there is a non-clustered index on the balance column:

```
SELECT account_no, balance FROM accounts
   WHERE balance BETWEEN 8000 AND 8100
```

What strategy will the query optimizer choose? The non-clustered index on the balance column is, in fact, chosen. This is a reasonable plan given what we already know. If we execute the query, the following Show Stats IO output is displayed after 239 rows have been returned:

```
Table 'accounts'. Scan count 1, logical reads 241,
physical reads 0, read-ahead reads 0.
```

The query optimizer has chosen to use a non-clustered index to access the data and has taken 241 logical reads to do so. Now suppose we execute the following query:

```
SELECT account_no, balance FROM accounts
```

```
      WHERE balance BETWEEN 8000 AND 9000
```

What strategy will the query optimizer now choose? As we might expect, the query optimizer has decided to use a table scan. Again, this is a reasonable plan given what we already know. As the number of rows returned increases, it becomes more efficient to execute a table scan rather than use the non-clustered index. If we execute the query, the following Show Stats IO output is displayed after 2,426 rows have been returned:

```
Table 'accounts'. Scan count 1, logical reads 1570,
physical reads 0, read-ahead reads 0.
```

So, the query optimizer has now chosen to use a table scan, taking 1,570 logical reads to do so.

Now let us place the query in a stored procedure, as follows:

```
CREATE PROCEDURE dbo.usp_accounts_per_range (@minbal
MONEY, @maxbal MONEY)
AS
SET STATISTICS IO ON
SELECT account_no, balance FROM accounts

      WHERE balance BETWEEN @minbal AND @maxbal
RETURN
```

Let us execute it with the following EXEC statement:

```
EXEC dbo.usp_accounts_per_range @minbal=8000, @maxbal =
8100

account_no  balance
7880        8000.43
12053       8000.43
 :
 :

Table: accounts  scan count 1,  logical reads: 241,
physical reads: 0,  read ahead reads: 0
```

This is exactly the same number of logical reads as before. The query optimizer has chosen a query plan that uses the non-clustered index as it did for the stand-alone query.

Now let us execute the stored procedure with the following EXEC statement:

```
EXEC dbo.usp_accounts_per_range @minbal=8000, @maxbal =
9000

account_no balance

7880       8000.43
12053      8000.43
 :
 :

Table: accounts  scan count 1,  logical reads: 2433,
physical reads: 0,  read ahead reads: 0
```

The number of logical reads has increased from 1,570 executing the query as a stand-alone statement to 2,433 executing the query in a stored procedure. Why is this?

The problem is caused by the fact that the query plan was created and loaded into cache by the first execution. The query optimizer created the query plan based on the parameters passed to it, so in this case it created a query plan for the SELECT statement, as follows:

```
SELECT account_no, balance FROM accounts
   WHERE balance BETWEEN 8000 AND 8100
```

The next time the stored procedure was executed no query optimization was done, and the query plan utilizing the non-clustered index was used. This is not the most efficient query plan for the range, as can be seen from the logical reads.

In its worst manifestation we can imagine that the first stored procedure execution happens to use a query plan that is not efficient for all subsequent stored procedure executions. So how can we deal with this situation?

One mechanism available to us is to make sure that the stored procedure always creates and uses a new query plan. We can force a stored procedure to create and use a new query plan, but there are also times when a stored procedure is automatically recompiled.

We will look shortly at how we can force a stored procedure to create and use a new query plan, but first let us look at some of the situations that result in SQL Server automatically recompiling a plan. To check if a stored procedure plan is recompiled the SP:Recompile SQL Server Profiler event can be monitored.

8.4.1.1 *Changes to the table structure*

If the structure of a table referenced by the stored procedure is changed, typically by the use of an ALTER TABLE statement, the schema_ver and base_schema_ver columns in the sysobjects system table are incremented. This informs SQL Server that it needs to recompile the stored procedure plan the next time the stored procedure executes. Examples of structure changes are the addition and deletion of columns and constraints.

8.4.1.2 *Changes to indexes*

If indexes are created and dropped, the schema_ver and base_schema_ver columns are incremented. This will cause a stored procedure recompilation even if the indexes are not relevant to the queries in the stored procedure.

8.4.1.3 *Executing update statistics*

If UPDATE STATISTICS is run against a table referenced by the stored procedure, stored procedure recompilation will take place the next time the stored procedure is executed. Running UPDATE STATISTICS increments the base_schema_ver and stats_schema_ver columns.

8.4.1.4 *Aging the stored procedure out of cache*

We will discuss how stored procedures are aged out of cache later. If this happens, then the next time the stored procedure executes it must be compiled and cached again.

8.4.1.5 *Table data modifications*

SQL Server will detect that a certain fraction of the data in a table has changed since the original plan was compiled. Once this threshold has been crossed a recompilation will occur. To keep track of the changes to the table data, the rowmodctr column in the sysindexes system table is incremented whenever one of the following conditions occurs to the table in question:

- A row is inserted.
- A row is deleted.
- An indexed column is updated.
- When a predefined threshold has been crossed, the statistics for the table will be automatically updated when it is accessed next, assuming the database option Auto update statistics is set to on. This automatic updating of statistics will reset the rowmodctr column. This threshold tends to depend on the size of the table.

So the stored procedure is recompiled when the threshold is crossed. As was mentioned earlier, the SP:Recompile SQL Server Profiler event can be monitored to check for recompilations; however, trace flag 205 can also be used to output information about when a statistics-dependent stored procedure is being recompiled. I tend to set this in the Startup Parameters section of the General Tab in Server Properties in the SQL Server Enterprise Manager together with trace flag 3605 to ensure logging of trace messages to the error log. A typical pair of messages logged follows:

```
Recompile issued : ProcName: usp_GetAccts   LineNo:2
StmtNo: 3

Schema Change: Tbl Dbid: 7 Objid: 1993058136 RowModCnt:
25000.000000 RowModCntMax: 0 RowModLimit: 22000
```

The first message specifies the stored procedure. The second message holds the table name in the form of its object ID. The item RowModCnt is the total number of modifications to the table, and RowModLimit is the threshold, which, when exceeded, results in the statistics being updated for the table and the stored procedure being recompiled. It is possible to ensure that the query optimizer does not recompile a query due to changes in statistics or to the indexed column by using the KEEPFIXED PLAN query option. In this case a query will only be recompiled if the table schema changes or sp_recompile is executed specifying the table.

8.4.1.6 Mixing data definition language and data manipulation language statements

If Data Definition Language (DDL) statements and Data Manipulation Language (DML) statements are mixed together in a stored procedure, the stored procedure will be recompiled when the DML statements are executed. The following example displays a stored procedure:

```
CREATE PROC dbo.usp_DDL_DML
AS
CREATE TABLE #table1 (c1 INT)
SELECT COUNT(*) FROM #table1
CREATE INDEX i1 ON #table1(c1)
SELECT COUNT(*) FROM #table1

CREATE TABLE #table2 (c1 INT)
SELECT COUNT(*) FROM #table2
CREATE INDEX i2 ON #table2(c1)
SELECT COUNT(*) FROM #table2
RETURN
```

This will result in four stored procedure recompilations. When the stored procedure compilation takes place the first time around, the temporary tables—#table1 and #table2 have not yet been created. The stored procedure must execute for this to happen. The SELECT statements that access #table1 and #table2 are not yet able to have a plan created. When the stored procedure executes, #table1 is created and then accessed by the first SELECT statement. Since a plan does not exist for this query, the stored procedure is recompiled in order to create a plan for this query.

The index is then created for #table1. A SELECT statement is then executed against #table1, but, as we have previously mentioned, this is treated as a schema change and therefore the stored procedure is recompiled again. The same recompilations occur because of #table2, and thus four recompilations are performed. It would have been better to place all the DDL statements at the beginning of the stored procedure and then execute the DML statements. Doing this results in one stored procedure recompilation.

8.4.2 Temporary tables

Another reason that stored procedures may be recompiled concerns the use of temporary tables. SQL Server will recompile a stored procedure if a few changes have been made to a temporary table created in the stored procedure. At the time of writing, only six changes to the temporary table have to be made inside the stored procedure before it is recompiled. This means that changes to a temporary table will result in recompilation far more frequently than in the case of a permanent table, as previously discussed. If you wish to apply the same recompilation thresholds to temporary tables as were applied to permanent tables, use the KEEP PLAN query option on any query that uses the temporary table.

8.4.3 Forcing recompilation

How can we manually cause a stored procedure to be recompiled? There are a number of mechanisms. One is the sp_recompile stored procedure. CREATE PROCEDURE WITH RECOMPILE.EXECUTE WITH RECOMPILE.

The sp_recompile system stored procedure ensures that each stored procedure and trigger that uses the specified table are recompiled the next time the stored procedure and triggers are run:

```
EXEC sp_recompile accounts

Object 'accounts' was successfully marked for
recompilation.
```

It is also possible to specify a stored procedure name instead of a table name, in which case only that stored procedure will be recompiled the next time it is run.

The sp_recompile system stored procedure actually increments the schema_ver and base_schema_ver column in the sysobjects system table. Note that triggers are also affected. Triggers are just a special kind of stored procedure that are automatically executed when inserts, updates, and deletes happen to a table. As such, they have their query plans stored in cache like any other stored procedure.

When we create a procedure, we can use the WITH RECOMPILE option. This means that every execution of a stored procedure causes a new query plan to be created. Using this option means that we do not have the problem of a query plan resident in cache that is inefficient for various parameter combinations. However, because we generate a new query plan for each execution of the stored procedure, the performance benefit of stored procedures is negated.

A less-severe option is to execute a stored procedure with the WITH RECOMPILE option. This causes a new query plan to be created for just that execution.

These options will help us avoid the problem described previously with an inefficient query plan loaded into procedure cache, but they do mean that new query plans get created. Another option is to break up the stored procedure into smaller pieces:

```
CREATE PROC dbo.usp_few_accounts_per_range (@minbal
            MONEY, @maxbal MONEY) AS
            SET STATISTICS IO ON
            SELECT account_no, balance FROM accounts
               WHERE balance BETWEEN @minbal AND @maxbal
            RETURN
GO

CREATE PROC dbo.usp_many_accounts_per_range (@minbal
            MONEY, @maxbal MONEY) AS
SET STATISTICS IO ON
            SELECT account_no, balance FROM accounts
               WHERE balance BETWEEN @minbal AND @maxbal
            RETURN
GO

CREATE PROC dbo.usp_accounts_per_range (@minbal MONEY,
@maxbal MONEY) AS
            IF (@maxbal - @minbal) <= 100
               EXEC dbo.usp_few_accounts_per_range @minbal,
@maxbal
            ELSE
               EXEC dbo.usp_many_accounts_per_range @minbal,
@maxbal
            RETURN
GO
```

The stored procedure usp_accounts_per_range is executed passing the minimum and maximum balance. It tests to see if the difference between the minimum and maximum balance is less than or equal to 100, and, if it is, it executes the stored procedure usp_few_accounts_per_range. If the difference is greater than 100, it executes the stored procedure usp_many_accounts_per_range. In this way the two stored procedures that access the data are compiled with their own execution plan. In this example the stored procedure usp_few_accounts_per_range gets a query plan that uses a nonclustered index, whereas the query plan for usp_many_accounts_per_range uses a table scan.

This method can work well, but it did require the developer writing the stored procedures to know that a balance range greater than 100 was best dealt with by a table scan, and, of course, this distribution can change over time. Another approach is to recompile not the whole stored procedure but only the troublesome statement. This can be brought about by using the EXECUTE statement with a character string:

```
CREATE PROC dbo.usp_example_proc (@bal MONEY) AS
        DECLARE @balstr VARCHAR(10)
        SELECT @balstr = CONVERT(VARCHAR(10), @bal)

          :

        EXECUTE ('SELECT account_no, balance FROM accounts
        WHERE balance > ' + @balstr)

          :

        RETURN
```

The Transact-SQL statement inside the EXECUTE statement goes through the same phases that any standalone Transact-SQL statement goes through—that is, parsing through to query compilation. This does not happen until the EXECUTE statement is performed. Other Transact-SQL statements in the stored procedure are compiled just once. To see the plan used for the Transact-SQL statement in the EXECUTE you need to look at the query plan after the stored procedure has been executed. In other words, choose Show Execution Plan from the Query menu in the Query Analyzer.

Another possibility is to use query optimizer hints. We have already seen optimizer hints and how they can be used to force the query optimizer to use a particular index. Optimizer hints can also be used with queries in stored procedures to ensure that a particular query plan is always used.

8.4.4 Aging stored procedures from cache

Versions of SQL Server prior to SQL Server 7.0 used two areas of cache—one for stored procedure plans and one for database pages, in particular data and index pages. SQL Server 7.0 and SQL Server 2000 use a single unified cache for database pages, stored procedure plans, and the query plans of SQL statements that are not stored procedures. The cache can grow and shrink dynamically as the memory allocated to SQL Server grows and shrinks.

Different stored procedures will require different amounts of effort to compile. Some will be simple and cheap to compile, and some will be complex and expensive to compile. To ensure that a stored procedure plan that is expensive to compile is not as easily aged out of cache as a simple stored procedure, the cost of the stored procedure compilation is stored with the plan.

If memory is tight, a component of SQL Server known as the lazywriter frees up cache pages. It does this by looking at the buffers in cache and checking the cost value associated with them. The lazywriter will decrement the cost of a buffer page by one. If the lazywriter finds that the cost of a page is zero, it will be freed. Conversely, if a stored procedure plan is used, the cost is set to the initial creation cost. This means that a frequently used stored procedure will not have its cost decremented over time to zero by the lazywriter. Also, a stored procedure that was expensive to compile and therefore has an associated large cost will take a long time to have its cost decremented to zero. Therefore, a stored procedure that is expensive to compile but not used frequently may stay in cache, as would a stored procedure that is cheap to compile but is used frequently.

8.5 Non-stored procedure plans

If you wish to ensure that a query plan is created and stored in cache, then placing the query inside a stored procedure will guarantee this. However, SQL Server does not only place stored procedure plans in cache. It will store the plans of SQL statements that are not part of a stored procedure in cache and attempt to reuse them.

SQL Server distinguishes between RPC events and SQL language events. RPC events are parameterized in some way. If the developer has used sp_executesql to submit the query or has used the prepare/execute model from the database API, it is an RPC event. Parameterization is typically used by a developer who wishes to submit a SQL statement for multiple execution, and in this case it makes sense to try to keep the query plan of the SQL statement.

A SQL language event is typically a SQL statement that is sent direct to the server. It has not been prepared and has not been submitted using sp_executesql. In this case the developer probably does not intend that the SQL statement be resubmitted multiple times.

Note: The SQL Server Profiler distinguishes between these events—for example, RPC:Starting, Prepare SQL, and SQL:StmtStarting.

When an RPC statement is received by SQL Server, the query plan is created and placed into cache. So that the query plan can be retrieved for a subsequent statement, some mechanism must be used to allow the plan to be identified. In the case of a stored procedure this was not necessary, since the stored procedure has a unique name. In the case of a SQL statement, which has no such name, the statement text is hashed to produce a hash key, which identifies it in cache. The hash key seems to be particularly sensitive to the statement text. The following two statements will have different keys even though the only difference is the case of the WHERE keyword (the server is case insensitive):

```
SELECT account_no FROM accounts where balance=100

SELECT account_no FROM accounts WHERE balance=100
```

Even the number of spaces in the statement is significant when hashing the statement text. Different plans will also be stored for identical statements that have different connection settings. Suppose two connections both execute the following SQL statement.:

```
SELECT account_no FROM accounts WHERE balance=100
```

Suppose one connection has its ANSI_NULL setting set to TRUE, and one connection has it set to FALSE. There will be two plans cached.

For nonparameterized (ad hoc) SQL language statements, the query optimizer may actually attempt to change a hard-coded value into a parameter marker in order to facilitate reuse of the query plan. This is known as autoparameterization. However, the query optimizer is very conservative, and few statements will undergo this process. The reason for this is the same as our previous discussion of stored procedure plans. A plan that is efficient for one parameter value may be extremely inefficient for another value. At least with stored procedures, the developer is in control and can use one of the techniques suggested earlier to avoid this problem. This is not the case with non-stored procedure statements, so the responsibility falls with SQL Server to avoid using inefficient plans.

To achieve this, it only autoparameterizes when it knows it is safe to do so. A typical case would be the following statement.:

```
SELECT balance FROM accounts WHERE account_no = 1000
```

There is a unique non-clustered index on the account_no column. An obvious efficient plan is to use this non-clustered index. Since this index is unique, a maximum of one row only can be returned.

Now consider the following statement.:

```
SELECT account_no FROM accounts
WHERE balance between 100 and 120
```

It would be very risky to replace the values 100 and 120 by parameter markers. Two different values from a subsequent query such as 50 and 5,000 would probably benefit from an entirely different plan.

It's worth it at this point to mention the system-stored procedure sp_executesql. This allows the developer to build a Transact-SQL statement that can be executed dynamically. Unlike the EXECUTE statement though, sp_executesql allows the setting of parameter values separately from the Transact-SQL string. This means that sp_executesql can be used instead of stored procedures to execute a Transact-SQL statement a number of times when only the parameters change. Because the Transact-SQL statement itself does not change—rather, the parameter values change—it is highly probable that the query optimizer will reuse the query plan it creates and saves for the first execution. Again, it is up to the developer, being familiar with the data, to decide whether reusing plans is a good strategy for a particular statement.

Here is an example of using sp_executesql:

```
DECLARE @MoneyVariable MONEY
DECLARE @SQLString NVARCHAR(500)
DECLARE @ParameterDefinition NVARCHAR(500)
```

```
-- Create the SQL String – only need to do this once
SET @SQLString =

     N'SELECT account_no FROM accounts WHERE balance =

@bal'

SET @ParameterDefinition = N'@bal MONEY'

-- Execute the string with the first parameter value

SET @MoneyVariable = 100

EXECUTE sp_executesql @SQLString, @ParameterDefinition,

              @bal = @MoneyVariable

-- Execute the string with the next parameter value

SET @MoneyVariable = 200

EXECUTE sp_executesql @SQLString, @ParameterDefinition,

              @bal = @MoneyVariable
```

Note that the query plans of the non-stored procedure SQL statements are placed in the cache and aged in a manner similar to stored procedures, described previously. Ad hoc statements that are not autoparameterized, however, will be placed in the cache with a cost of zero, so their plans will be removed from cache as soon as memory becomes short.

8.6 The syscacheobjects system table

To check for plans in cache the system table syscacheobjects can be queried. Here is a fragment of the output of syscacheobjects:

```
SELECT cacheobjtype, objtype, sql FROM syscacheobjects

cacheobjtype objtype          sql
```

```
Compiled Plan Adhoc          SELECT account_no FROM accounts
where balance=100

Compiled Plan Adoc           SELECT account_no FROM accounts
WHERE balance=100

Compiled Plan Adhoc          SELECT account_no FROM accounts
WHERE balance between
100 and 120

Executable Plan Prepared     (@1 smallint)SELECT
[balance]=[balance] FROM [accounts]
WHERE [account_no]=@1

Compiled Plan Prepared       (@1 smallint)SELECT
[balance]=[balance] FROM [accounts]
WHERE [account_no]=@1

Executable Plan Prepared     (@bal MONEY)SELECT account_no
FROM accounts
WHERE balance = @bal

Compiled Plan Prepared       (@bal MONEY)SELECT account_no
FROM accounts

WHERE balance = @bal

Executable Plan Proc         usp_accounts_per_range

Compiled Plan Proc           usp_accounts_per_range
```

The column sql holds the statement text. The column cacheobjtype represents the type of object in the cache. We can see that the two statements previously mentioned that have their WHERE keyword in different case are represented by separate plans. The statement that was too dangerous to autoparameterize with the balance between 100 and 120 values is held as a separate plan. All three statements are held as ad hoc objects in the objtype column. This column holds the type of object.

One of our statements was autoparameterized:

```
SELECT balance FROM accounts WHERE account_no = 1000
```

This is held as a prepared object, as is the statement that was submitted through sp_executesql. Finally, we can see that a stored procedure is also held in cache. Because different users will usually have different parameter values when executing stored procedures and prepared statements, they must also be given an execution context as well as a completely shared plan.

9

SQL Server and Windows

This chapter discusses SQL Server performance with respect to the CPU, memory, and disk resources found on a Windows server.

9.1 SQL Server and CPU

The first resource on a Windows server that is usually monitored is the CPU. CPUs have been gaining in power dramatically over the last few years, and Windows supports multiprocessor systems.

Although a multiprocessor system may not reduce CPU bottlenecks when a single threaded process is consuming the CPU, multithreaded processes such as SQL Server will benefit greatly.

CPU is a system resource. The more CPU power available the better the system is likely to perform. Windows schedules CPU time to the threads of a process, and, if more threads require CPU time than there is CPU time available, a queue of waiting threads will develop. Sometimes a processor bottleneck is actually masking another bottleneck, such as memory, so it is important to look at CPU use in conjunction with other resource use on the system. This first part of the chapter provides an overview of CPU usage and looks at how SQL Server makes use of the CPU. It then looks at how CPU bottlenecks can be observed.

9.1.1 An overview of Windows and CPU utilization

To understand the way that Windows uses the CPU we first of all need to consider the difference between a process and a thread. A process can be considered to be an object containing executable code and data; an address space, which is a set of virtual addresses; and any other resources allocated to the code as it runs. It also must contain a minimum of one thread of execution.

A thread is the item inside a process that is scheduled to run, not the process itself as in some older operating systems. A Windows process can contain any number of threads, and a process that contains more than one thread is known as a multithreaded process. Windows is able to simultaneously schedule a number of threads across multiple CPUs. These can be threads belonging to many processes or threads belonging to just one process.

Each running instance of SQL Server is a multithreaded process, and so it is able to schedule a number of threads simultaneously across multiple processors to perform a multitude of functions. SQL Server may have threads concurrently executing across multiple processors with one servicing a user connection, one performing a backup, and one writing pages from cache to disk. Also, SQL Server is able to perform queries in parallel as well as various database operations in parallel, such as index creation. Although SQL Server can be parallelizing operations across multiple processors, it can be restricted to only using a subset of the available processors on the server.

The order in which threads are scheduled is governed by a priority associated with those threads. Windows always schedules the highest-priority thread waiting for processor time to run first in order to make sure that the highest-priority work gets done first. Each process is allocated to one of four base priority classes:

- Idle
- Normal
- High
- Real time

The base priority of a process can change within its base priority class. The base priority of a process thread varies within the base priority of its parent process. As a general rule, the base priority of a thread varies only within a range of two greater than or two less than the base priority of its process. The dynamic priority of a thread governs when it will be scheduled. The dynamic priority of a thread is constantly being adjusted by Windows. For example, the dynamic priority of a thread is typically increased when an I/O operation it has been waiting for completes and the thread now needs processor time. The dynamic priority of a thread can equal or grow beyond its base priority, but it can never drop below it.

SQL Server also has the concept of fibers. Normally, SQL Server executes work using Windows threads. Work is allocated to threads. The Windows operating system code that manages threads runs in kernel mode. Switching threads requires switches between the user mode of the application code and the kernel mode of the thread manager. This context switching can be expensive on systems with multiple CPUs that are very busy. For that reason, SQL Server can be configured to use fibers by means of the lightweight pooling server configuration option. Setting this option can be accomplished using sp_configure or setting the option on the Processor tab of the SQL Server Properties (Configure) window in the SQL Server Enterprise Manager.

Lightweight pooling allows SQL Server to manage scheduling within the normal Windows thread structures. Fibers are managed by code running in user mode, and switching fibers does not require the user-mode to kernel-mode context switch needed to switch threads. Each Windows thread can support multiple fibers, and SQL Server performs the scheduling of these fibers. For most SQL Server systems, using lightweight pooling is unlikely to produce any noticeable benefit.

9.1.2 How SQL Server uses CPU

There are various ways that SQL Server can be configured with respect to how it makes use of the CPU. These can be grouped into the following categories:

- Priority
- Use of symmetric multiprocessing systems
- Thread use
- Query parallelism

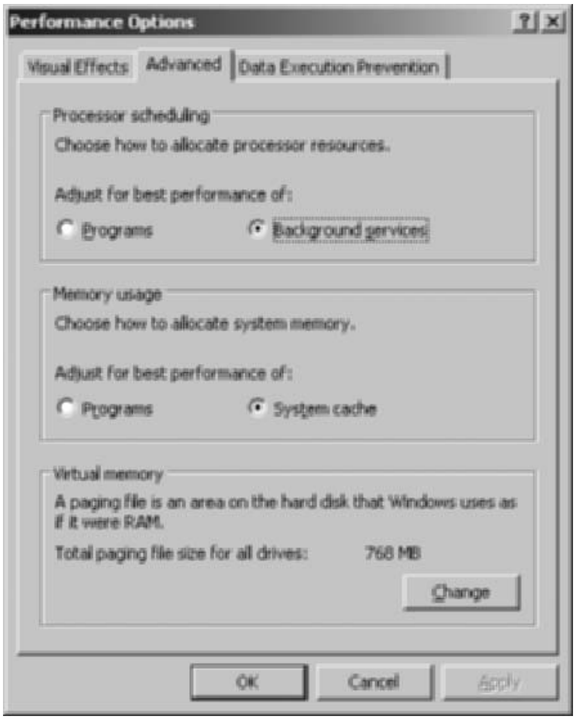
Let us consider each of the above categories in turn.

9.1.2.1 Priority

On the Windows Server running SQL Server it is likely that little interactive use will take place. The server will communicate with client workstations. Usually, when there is interactive use made of a workstation, it is preferable to increase the priority of the foreground application—that is, the application running in the window that is currently displayed at the top of the other windows.

By default, Windows Server has longer, fixed quanta with no priority boost for foreground applications, allowing background services to run more efficiently. Windows Professional, however, defines short, variable quanta for applications and gives a foreground application a priority boost (a quantum is the maximum amount of time a thread can run before the system checks for another thread of the same priority to run).

Whether a priority boost for foreground applications occurs or not can be overridden. This can be done using the System icon in the Control Panel, choosing the Advanced tab, and mouse-clicking the Performance Options button. This is shown in Figure 9.1.


Performance Options

Visual Effects | Advanced | Data Execution Prevention

Processor scheduling

Choose how to allocate processor resources.

Adjust for best performance of:

Programs [✓ Background services]

Memory usage

Choose how to allocate system memory.

Adjust for best performance of:

Programs [✓ System cache]

Virtual memory

A paging file is an area on the hard disk that Windows uses as if it were RAM.

Total paging file size for all drives: 768 MB

Change

OK Cancel Apply

Figure 9.1
The advanced performance options window

SQL Server is never a foreground application, and so, on the server, the performance should be optimized for Background services. On the client workstation, however, boosting the foreground priority by optimizing for Applications makes sense. Again, the choice of the Windows platform will likely accomplish this by default. Of course, using the Query Analyzer, for example, on the server directly will not benefit from any priority boost, so you might find that you do not get great performance. This does not mean that SQL Server is running slowly; it means that the Query Analyzer is not priority boosted and so will be contending equally with it for the CPU.

Another method of changing the priority of SQL Server is to change the advanced server configuration option priority boost. This governs whether or not SQL Server should run at a higher priority than other processes on the same server. Setting this option can be accomplished using sp_configure or setting the option on the Processor tab of the SQL Server Properties (Configure) window in the SQL Server Enterprise Manager.

Setting priority boost to 1 causes SQL Server to execute at a higher priority and to be scheduled more often. In fact, its priority will be changed from Windows base priority 7 to base priority 13. This will probably have a negative impact on other applications running on the server (including other instances of SQL Server), and therefore this parameter should be used with care unless the server has been designated as being dedicated to SQL Server (in which case why bother setting it anyway!). To use our previous example, executing the Query Analyzer locally on a server that has priority boost set to 1 would result in degraded Query Analyzer performance.

9.1.2.2 Use of symmetric multiprocessing systems

With respect to multiprocessor systems, the edition of SQL Server and the operating system platform on which it is running governs the maximum number of processors that can be supported.

For query parallelism, described shortly, the maximum number of processors that can be used to execute a query can be specified as a server configuration option, max degree of parallelism. Setting this option can be accomplished using sp_configure or setting the option on the Processor tab of the SQL Server Properties (Configure) window in the SQL Server Enterprise Manager. This also limits the degree of parallelism for utility execution such as DBCC CHECKDB.

Which processors on a multiprocessor system can SQL Server use? Generally, Windows does not guarantee that any thread in a process will run on a given processor. However, it uses a soft affinity algorithm, which tries to run a thread on the last processor that serviced it. A thread may still migrate from processor to processor if the favored processor is busy, which causes reloading of the processor's cache. Under heavy system loads, this is likely to degrade performance. Specifying the processors that should and should not run SQL Server threads can boost performance by reducing the reloading of processor cache. This is only likely to make a difference with four or more processors under load. By specifying the processors manually a hard affinity algorithm is used.

The association between a processor and a thread is called processor affinity. SQL Server enables a processor affinity mask to be specified as a server configuration option. By setting bits in the mask, the system administrator can decide on which processors SQL Server will run. The number of the bit set represents the processor. For example, setting the mask to the value 126 (hexadecimal 0x7E) sets the bits 01111110, or 1, 2, 3, 4, 5, and 6. This means that SQL Server threads should run on processors 1, 2, 3, 4, 5, and 6. On an eight-processor system this means that SQL Server threads should not run on processors 0 and 7.

In the SQL Server Enterprise Manager, the CPU affinity can be set in the Processor control section on the Processor tab of the SQL Server Properties (Configure) window.

Note: It is also possible to use the Set Affinity option in the Task Manager to allocate a process to specific CPUs.

For most database administrators, using a hard affinity option is unlikely to be an option that gains much in the way of performance.

9.1.2.3 Thread use

When a SQL Server client executes a request, the network handler places the command in a queue and the next usable thread from the worker pool of threads acquires the request and handles it. If no free worker thread is available when a request arrives, SQL Server creates a new thread dynamically, until it reaches the server configuration option maximum worker threads.

The default value for maximum worker threads is 255, which will often be greater than the number of users connected to the server. However, when there are a large number of connections (typically hundreds), using a thread for every user connection may deplete operating system resources. To avoid this, SQL Server can use a technique called thread pooling. With thread pooling a pool of worker threads will handle a larger number of user connections.

If the maximum worker threads value has not been exceeded, a new thread is created for each user connection. Once the maximum worker threads value has been exceeded, user connections will share the pool of worker threads. A new client request will be handled by the first thread in the pool that becomes free.

9.1.2.4 Query parallelism

In SQL Server, a single query can execute in parallel over multiple CPUs. For workloads that have a small number of complex queries running on SMP computers, this should bring a performance boost. For OLTP workloads, which consist of many small transactions, parallelism is unlikely to enhance performance.

Parallel query processing is aimed at improving the performance of single, complex queries. The query optimizer decides if a query plan can be executed in parallel based on various criteria. If it can, the query plan will contain extra operators, known as exchange operators, which will enable the query plan to be executed in parallel. At run time, SQL Server will decide, again based on various criteria, how many processors the query will use—that is, how many threads will be used. This is known as the Degree of Parallelism (DOP).

Parallel query processing is pretty much out of the box. There are, however, two server configuration options that affect parallel query processing:

- Max degree of parallelism
- Cost threshold for parallelism

The max degree of parallelism option controls the number of CPUs SQL Server can use for parallel queries—that is, the maximum number of threads a query can use. The cost threshold for parallelism controls the threshold over which the query optimizer will generate a parallel query plan. If a query is short, such as an OLTP query, the overhead of setting up a parallel query is not worth the gain.

The query optimizer will not generate a parallel query plan if the computer is only a single processor. Before the query starts to execute, SQL Server uses its knowledge of CPU use and the available memory to decide the degree of parallelism for the query. It may be that SQL Server decides not to run the query in parallel at all.

If the estimated cost of executing the query is less than the cost threshold for parallelism, the query optimizer will not generate a parallel plan. This is also true if the query optimizer determines that only a few rows will be returned. To summarize, the query optimizer will only generate a parallel query plan if it considers that it is worth doing so, and at run time the query will only be executed in parallel if SQL Server decides that there are sufficient free resources to do so.

There are SQL statements that will not be executed with a parallel query plan. INSERT, UPDATE, and DELETE statements will use a serial plan, but their WHERE clause may use a parallel plan. Static and keyset cursors can use a parallel plan but not dynamic cursors.

To control parallel query execution, as previously mentioned, the maximum number of processors that can be used to execute a query can be specified as a server configuration option, max degree of parallelism. Setting this option can be accomplished using sp_configure or setting the option on the Processor tab of the SQL Server Properties (Configure) window in the SQL Server Enterprise Manager. The default is to use all the processors.

The cost threshold for parallelism server configuration can be specified using sp_configure or setting the *Minimum query plan threshold . . .* value on the Processor tab of the SQL Server Properties (Configure) window in the SQL Server Enterprise Manager. The default is five seconds.

There is also a query optimizer hint, which can be used to influence parallel query execution. The MAXDOP query hint allows the max degree of parallelism to be set on a statement-by-statement basis. However, this is not supported for CREATE INDEX statements.

The CREATE INDEX in SQL Server can be executed in parallel. Assuming that the max degree of parallelism option is sufficiently high, and the workload on the server is not great, the CREATE INDEX statement can be executed across all the CPUs. To give each CPU an equal portion of work to do, a fast, random initial scan is performed to check on the data value distribution of the table column that will be used for the index column. This initial thread then dispatches the number of threads determined by the max degree of parallelism option. Each thread builds its own index structure based on the range of data it is working with. The initial thread then combines these smaller index structures into a single index structure.

Let us now look at how we can detect processor bottlenecks.

9.1.3 Investigating CPU bottlenecks

The tools used to observe CPU bottlenecks are typically the System Monitor and the Task Manager. We will focus on using the System Monitor in this section, although the Processes and Performance tabs in the Task Man-

Table 9.1 *Selected counters for the System, Processor, and Process Objects*

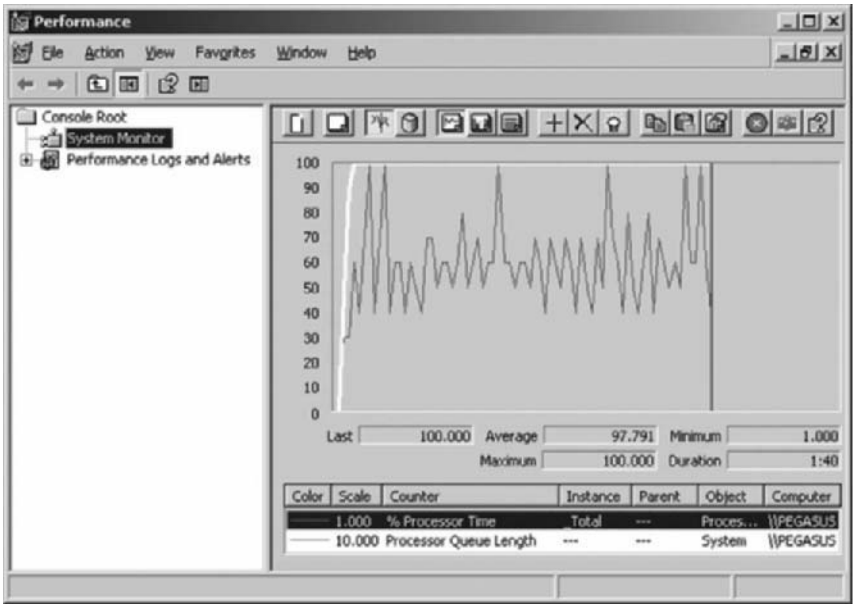
CPU-Related Counters	Explanation
System: Processor Queue Length	The number of threads that need CPU time but have to wait. This counts only ready threads, not those being handled. This counter belongs to the system object, because there is only one queue even when there are multiple processors on the server.
Processor: % Processor Time	This is the percentage that a processor is busy. There is an instance of this counter for every processor on the server. The _Total instance can be used to display the value of total processor utilization system-wide.
Processor: % User Time	This is the percentage that a processor is busy in user mode. User mode means application code and subsystem code.
Processor: % Privileged Time	This is the percentage that a processor is busy in privileged mode. Privileged mode means operating system services.
Process: % Processor Time	This is the percentage of CPU time that a process is busy.

ager are also quite useful. These are shown later in Figures 9.14 and 9.15 when we investigate memory. The System, Processor, and Process objects are a useful place to start and it's worth a look at some of their counters, as shown in Table 9.1.

In Figure 9.2 the System Monitor is being used to monitor the following counters:

- Processor: % Processor Time
- System: Processor Queue Length

The counter Processor: % Processor Time is highlighted (in white). We can see that the processor appears to be 100 percent utilized. This in itself is not necessarily going to cause a bottleneck; however, we can see that the Processor Queue Length is quite high. It averages around six (note the scale factor of ten so it can be seen on the display) and peaks at around ten. To check the average and maximum, this counter was selected instead of the counter Processor: % Processor Time counter. This means that on average,


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
90
80
70
60
50
40
30
20
10
0

Last 100.000 Average 97.791 Minimum 1.000
Maximum 100.000 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
—	1.000	% Processor Time	_Total	---	Proces...	\\PEGASUS
—	10.000	Processor Queue Length	---	---	System	\\PEGASUS

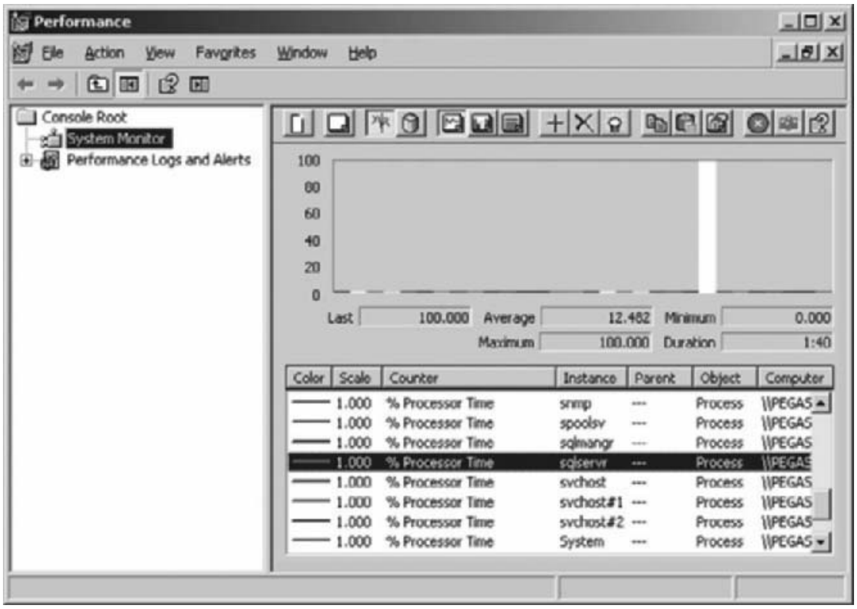
Figure 9.2
A busy processor

six threads are waiting for the CPU; this is a clear indication that we have a processor bottleneck.

The lows and highs in the Processor Queue Length counter display are caused by the randomness that ready tasks are being generated. This is not uncommon. Queues usually appear when the processor is very busy, but they can appear when CPU utilization not high. This can happen if requests for the processor's time arrive randomly and if threads demand irregular amounts of time from the processor.

So what is causing the bottleneck? Is it one process or many processes? We can monitor the processor use of each process to get a feel for the answer. In Figure 9.3 the System Monitor is being used to monitor the Process: % Processor Time counter.

We have selected the Histogram display to make it easier to look at the processes using the processor. It is pretty clear that one process is monopolizing the processor. This is the highlighted process and we can see that it is SQL Server. The only problem is that we do not know which SQL Server! We may have many instances of SQL Server running, and in each case the instance will be named sqlservr in the System Monitor. There are various approaches to finding out which instance is which. One approach I find useful is to create a System Monitor report showing the Process: % Proces-

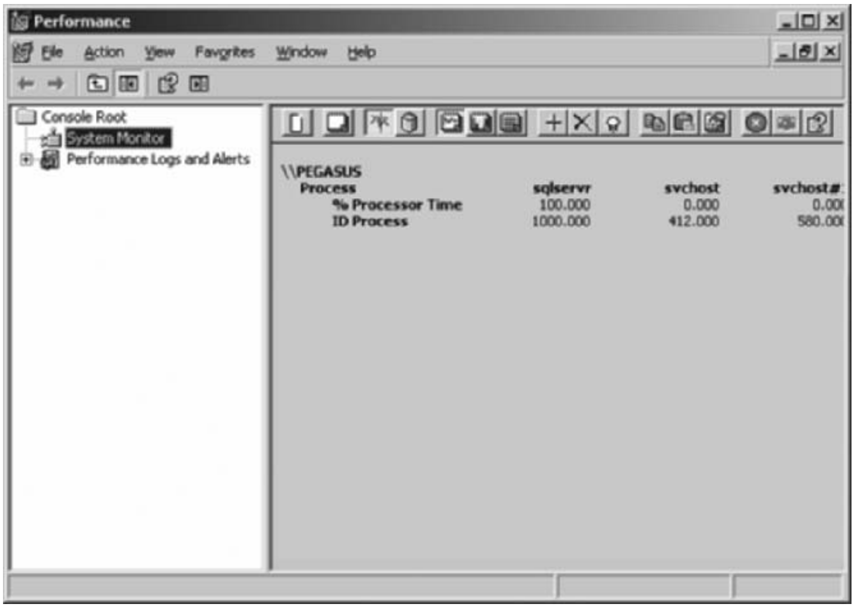

Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
80
60
40
20
0

```
Last    100.000   Average   12.482   Minimum    0.000
                  Maximum  100.000   Duration    1:40
```

Color	Scale	Counter	Instance	Parent	Object	Computer
——	1.000	% Processor Time	snmp	---	Process	\\PEGAS
——	1.000	% Processor Time	spoolsv	---	Process	\\PEGAS
——	1.000	% Processor Time	sqlmangr	---	Process	\\PEGAS
——	1.000	% Processor Time	sqlservr	---	Process	\\PEGAS
——	1.000	% Processor Time	svchost	---	Process	\\PEGAS
——	1.000	% Processor Time	svchost#1	---	Process	\\PEGAS
——	1.000	% Processor Time	svchost#2	---	Process	\\PEGAS
——	1.000	% Processor Time	System	---	Process	\\PEGAS

Figure 9.3 *Monitoring processor time for individual processes*

sor Time counter and the Process: ID Process counter. This is shown in Figure 9.4.


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts

```
\\PEGASUS
  Process                  sqlservr     svchost    svchost#
     % Processor Time       100.000       0.000       0.00
     ID Process            1000.000     412.000     580.00
```

Figure 9.4 *Checking process ID for the SQL Server instance*

We can confirm that the instance sqlservr with process ID 1000 is using up the CPU. Another way (often easier) is to check the Processes tab in the Task Manager. This is shown in Figure 9.5.

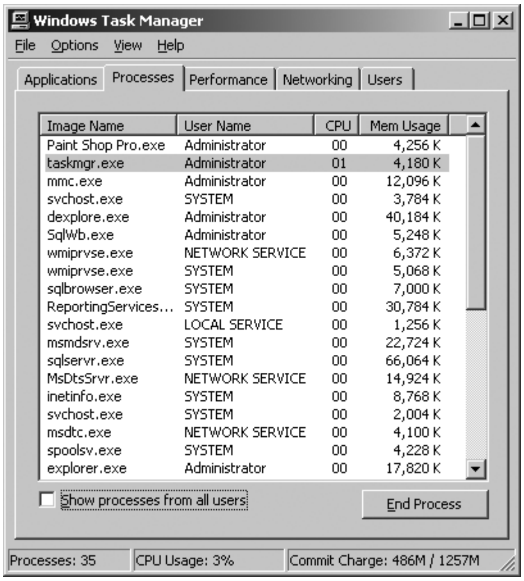


Figure 9.5
The Task Manager processes tab

Windows Task Manager

File Options View Help

Applications | Processes | Performance | Networking | Users

Image Name	User Name	CPU	Mem Usage
Paint Shop Pro.exe	Administrator	00	4,256 K
taskmgr.exe	Administrator	01	4,180 K
mmc.exe	Administrator	00	12,096 K
svchost.exe	SYSTEM	00	3,784 K
dexplore.exe	Administrator	00	40,184 K
SqlWb.exe	Administrator	00	5,248 K
wmiprvse.exe	NETWORK SERVICE	00	6,372 K
wmiprvse.exe	SYSTEM	00	5,068 K
sqlbrowser.exe	SYSTEM	00	7,000 K
ReportingServices...	SYSTEM	00	30,784 K
svchost.exe	LOCAL SERVICE	00	1,256 K
msmdsrv.exe	SYSTEM	00	22,724 K
sqlservr.exe	SYSTEM	00	66,064 K
MsDtsSrvr.exe	NETWORK SERVICE	00	14,924 K
inetinfo.exe	SYSTEM	00	8,768 K
svchost.exe	SYSTEM	00	2,004 K
msdtc.exe	NETWORK SERVICE	00	4,100 K
spoolsv.exe	SYSTEM	00	4,228 K
explorer.exe	Administrator	00	17,820 K

☐ Show processes from all users

End Process

Processes: 35 | CPU Usage: 3% | Commit Charge: 486M / 1257M

If we click on the CPU column heading, the display will be sorted with the process using most of the CPU displayed first. We can easily read off the process ID from the PID column.

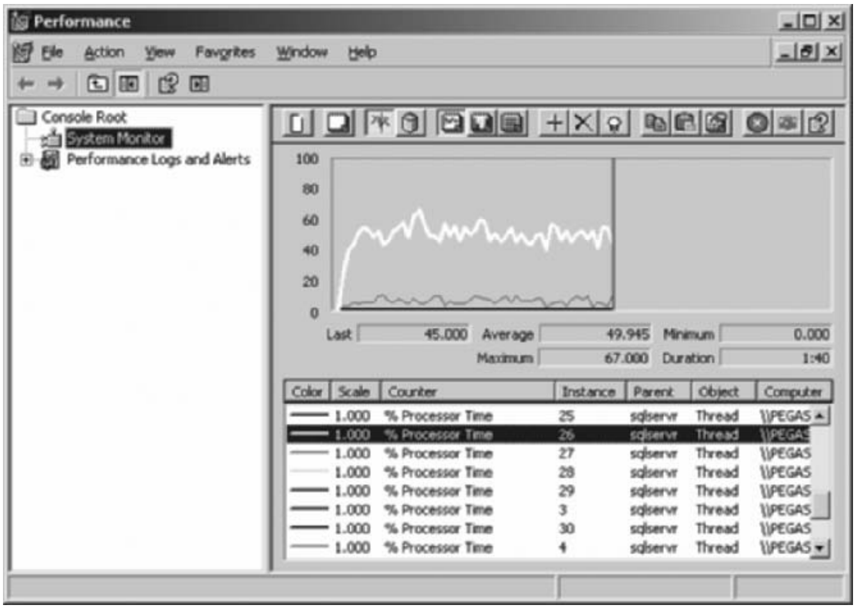
Whichever method we use to find the process ID, once we have obtained it we now need to translate it into a SQL Server instance. An easy way to do this is to connect to the SQL Server instance you suspect in the Query Analyzer and execute the following statement.

```
SELECT SERVERPROPERTY('ProcessID')
```

This will return the process ID. If it is not correct, connect to the next instance and check that. Most servers will not be running more than a few instances.

Once we have established the SQL Server instance that is monopolizing the processor, we need to further investigate why this is so, and, if it is not a database or application design problem, perhaps consider moving the instance of SQL Server onto its own server. If no process stands out in this display, this might be an indication that the processor is just too slow.

Can we drill down further into SQL Server? We can look at the individual threads. In Figure 9.6 the System Monitor is being used to monitor the Thread: % Processor Time counter for all the SQLSERVR process's threads. We can clearly see that one thread with thread instance number 26 is using most of the CPU.



Last 45.000 Average 49.945 Minimum 0.000
Maximum 67.000 Duration 1:40

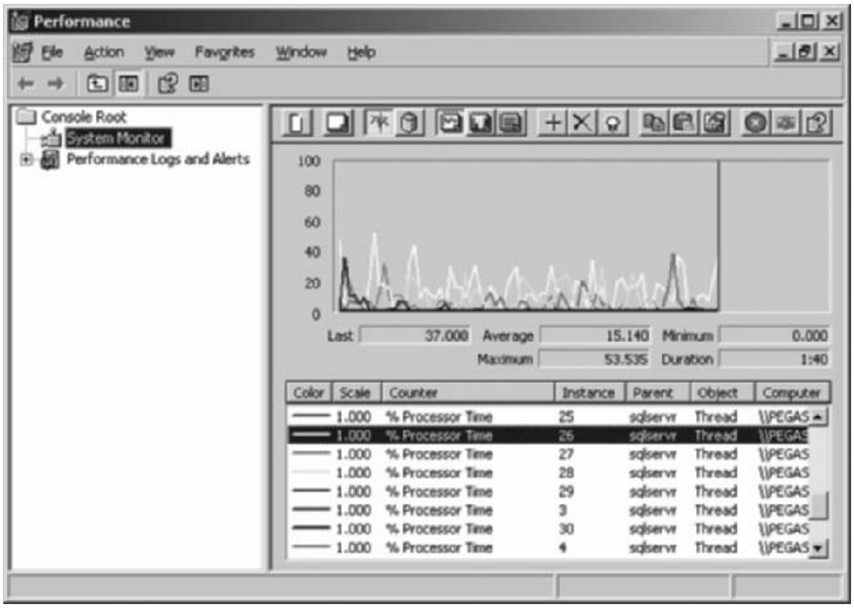
Color	Scale	Counter	Instance	Parent	Object	Computer
	1.000	% Processor Time	25	sqlservr	Thread	\\PEGAS
	1.000	% Processor Time	26	sqlservr	Thread	\\PEGAS
	1.000	% Processor Time	27	sqlservr	Thread	\\PEGAS
	1.000	% Processor Time	28	sqlservr	Thread	\\PEGAS
	1.000	% Processor Time	29	sqlservr	Thread	\\PEGAS
	1.000	% Processor Time	3	sqlservr	Thread	\\PEGAS
	1.000	% Processor Time	30	sqlservr	Thread	\\PEGAS
	1.000	% Processor Time	4	sqlservr	Thread	\\PEGAS

Figure 9.6 *A single SQL Server thread monopolizing the CPU*

Compare this with Figure 9.7. Here we see that many SQL Server threads are running the CPU. So looking at the Thread: % Processor Time counter can be useful to help distinguish between the case of one busy connection versus many busy connections. However, I find that at this point I really want to start using the SQL Profiler because it provides information more appropriate to SQL Server, rather than the operating system as a whole.

Chapter 12 discusses the SQL Profiler in detail. We wish to check for connections that are using a large proportion of the CPU and which SQL statements on those connections are using the most CPU.

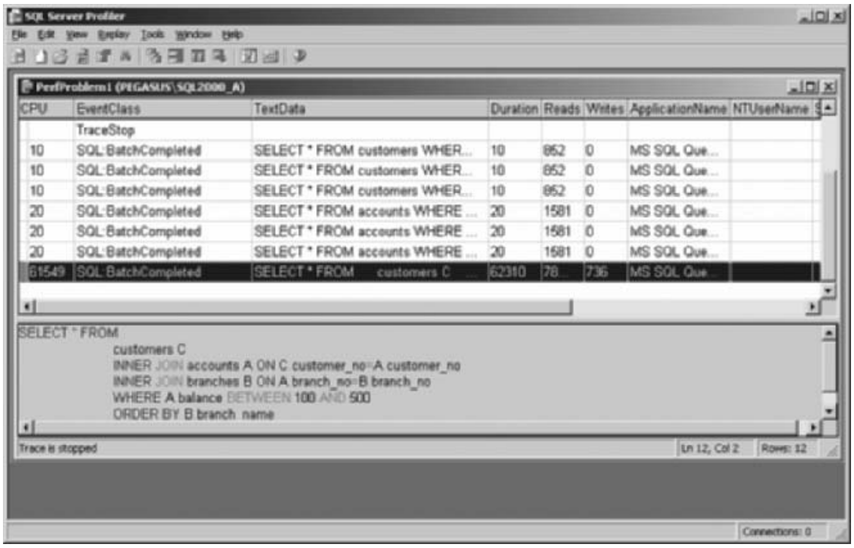
For our requirement we can create a trace with the SQLServerProfiler-Standard template. The default events are sufficient, since they include the events that we need. We can choose to filter out low CPU use events, but we must be careful not to filter out information that might prove useful in our investigation. In Figure 9.8, a graphic SQL Profiler display is shown.


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
80
60
40
20
0

Last 37.000 Average 15.140 Minimum 0.000
Maximum 53.535 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
—	1.000	% Processor Time	25	sqlservr	Thread	\\PEGAS
—	1.000	% Processor Time	26	sqlservr	Thread	\\PEGAS
—	1.000	% Processor Time	27	sqlservr	Thread	\\PEGAS
—	1.000	% Processor Time	28	sqlservr	Thread	\\PEGAS
—	1.000	% Processor Time	29	sqlservr	Thread	\\PEGAS
—	1.000	% Processor Time	3	sqlservr	Thread	\\PEGAS
—	1.000	% Processor Time	30	sqlservr	Thread	\\PEGAS
—	1.000	% Processor Time	4	sqlservr	Thread	\\PEGAS

Figure 9.7 *Many SQL Server threads using the CPU*


SQL Server Profiler
File Edit View Replay Tools Window Help
PerfProblem1 (PEGASUS\SQL2000_A)

CPU	EventClass	TextData	Duration	Reads	Writes	ApplicationName	NTUserName
	TraceStop						
10	SQL:BatchCompleted	SELECT * FROM customers WHER...	10	852	0	MS SQL Que...	
10	SQL:BatchCompleted	SELECT * FROM customers WHER...	10	852	0	MS SQL Que...	
10	SQL:BatchCompleted	SELECT * FROM customers WHER...	10	852	0	MS SQL Que...	
20	SQL:BatchCompleted	SELECT * FROM accounts WHERE ...	20	1581	0	MS SQL Que...	
20	SQL:BatchCompleted	SELECT * FROM accounts WHERE ...	20	1581	0	MS SQL Que...	
20	SQL:BatchCompleted	SELECT * FROM accounts WHERE ...	20	1581	0	MS SQL Que...	
61549	SQL:BatchCompleted	SELECT * FROM customers C ...	62310	78...	736	MS SQL Que...	

```
SELECT * FROM
            customers C
            INNER JOIN accounts A ON C.customer_no=A.customer_no
            INNER JOIN branches B ON A.branch_no=B.branch_no
            WHERE A.balance BETWEEN 100 AND 500
            ORDER BY B.branch_name
```

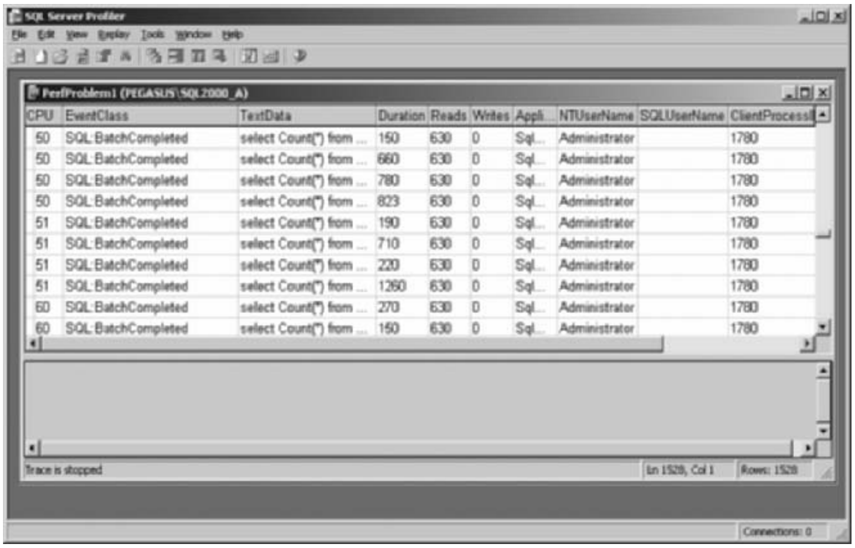
Trace is stopped   Ln 12, Col 2   Rows: 12
Connections: 0

Figure 9.8 *The SQL Profiler showing a single thread monopolizing the CPU*

The data columns have been grouped by the data column CPU, and we can immediately see that although many queries are using between 10 and 20 milliseconds of CPU, one query is using nearly 62 seconds of CPU. We can see that the duration of this query is about 62 seconds also. In fact, virtually this entire query is CPU. The SQL Profiler identifies the query syntax, application name, and so on so we can easily identify the problem query in our application. We can then, of course, investigate the query plan using the Query Analyzer and hopefully improve it.

We could have saved the trace into a table and then searched the table for events taking, for example, greater than one second of CPU. In practice, I find myself taking this approach most of the time.

In Figure 9.9, many queries are using between 50 and 60 seconds of CPU. No one query stands out. If the queries have a duration, reads, writes, and a CPU use that is expected, then it may be that the queries are efficient. If the processor is constantly busy and there is a significant queue, it may be the case that the CPU is just not powerful enough.


SQL Server Profiler
File Edit View Replay Tools Window Help
PerfProblem1 (PEGASUS\SQL2000_A)

CPU	EventClass	TextData	Duration	Reads	Writes	Appli...	NTUserName	SQLUserName	ClientProcessI
50	SQL:BatchCompleted	select Count(*) from ...	150	630	0	Sql...	Administrator		1780
50	SQL:BatchCompleted	select Count(*) from ...	660	630	0	Sql...	Administrator		1780
50	SQL:BatchCompleted	select Count(*) from ...	780	630	0	Sql...	Administrator		1780
50	SQL:BatchCompleted	select Count(*) from ...	823	630	0	Sql...	Administrator		1780
51	SQL:BatchCompleted	select Count(*) from ...	190	630	0	Sql...	Administrator		1780
51	SQL:BatchCompleted	select Count(*) from ...	710	630	0	Sql...	Administrator		1780
51	SQL:BatchCompleted	select Count(*) from ...	220	630	0	Sql...	Administrator		1780
51	SQL:BatchCompleted	select Count(*) from ...	1260	630	0	Sql...	Administrator		1780
60	SQL:BatchCompleted	select Count(*) from ...	270	630	0	Sql...	Administrator		1780
60	SQL:BatchCompleted	select Count(*) from ...	150	630	0	Sql...	Administrator		1780

Trace is stopped   Ln 1528, Col 1   Rows: 1528

Connections: 0

Figure 9.9 *The SQL Profiler showing many threads using the CPU*

9.1.4 Solving problems with CPU

Having determined that there is indeed a CPU bottleneck and that there is a queue of threads waiting for processor time, the next step is to find out what is using up the CPU. Other bottlenecks should be investigated, such as memory, to ensure that they are not manifesting themselves as a CPU bottleneck. If there is no particular candidate process to home in on, then the CPU is probably too slow and either a faster CPU can be purchased or an additional CPU. If it is obvious which application is monopolizing the CPU and it is not SQL Server, then it might be an idea to move that application to another server. Moving SQL Server off a Domain Controller may help if that is where it is installed.

If SQL Server is monopolizing the CPU, then it should be possible to track down a query that is inefficient and using too much CPU. If there is no particular candidate query to home in on, then the CPU is probably too slow and an additional CPU might be the most cost-effective solution.

Another consumer of CPU is the network interface card. Better network cards will save some CPU. Network interface cards that use bus-mastering direct memory access (DMA) are less of a burden on the CPU.

If SQL Server does not seem to be the main consumer of the CPU, it is always worth checking the counters System: Context Switches/sec and Processor: Interrupts/sec. The System: Context Switches/sec counter measures the average rate per second at which context switches among threads on the computer occur. On a multiprocessor system experiencing processor bottlenecks, high context switches may be reduced by using fibers, which can be enabled by setting the lightweight pooling server configuration option.

The Processor: Interrupts/sec counter measures the average rate per second at which the processor handles interrupts from applications or hardware devices. High activity rates can indicate hardware problems. Expect to see interrupts in the range upward from 1,000 per second for computers running Windows Server and upward from 100 per second for computers running Windows Professional.

One very important factor to consider is the processor cache. Use the largest processor cache that is practical. Typically, choose from 512 KB to 2 MB (or greater) for the L2 cache. Benchmarks have shown that upgrading to a faster processor but with a smaller cache usually results in poorer performance.

Multiprocessors need some further consideration. Adding extra processors to the server may well increase performance if SQL Server is bottlenecking on CPU. It is recommended that the addition of CPUs be accompanied by the addition of other resources such as memory and disk. It is recommended to scale memory with processors. For example, if a single-processor system requires 512 MB of memory and a second processor is added to increase the throughput, double the memory to 1,024 MB. The simple reason for this is that more processing power needs more memory, apart from the fact that coordinating two processors needs extra memory as well.

Because of the extra processors, the acceptable queue length will be longer. If the CPUs are mostly utilized, a queue value equal to about three per processor is not unreasonable. A four-processor server, for example, might have a queue length of 12.

If SQL Server is running in lightweight pooling mode—that is, using fibers, the queue length should not exceed one, because there is a single thread on each processor in which fibers are scheduled.

9.2 SQL Server and memory

Another important resource on a Windows server is memory. Over the last few years the amount of memory found on servers and workstations has rapidly increased.

Having large amounts of physical memory is not enough in itself. The software running on the server must be able to benefit from it, and it is therefore vital that the server operating system manages memory in an efficient and intelligent fashion. Windows employs a virtual memory manager to do just that, and it can provide excellent memory management on a wide range of memory configurations with multiple users.

SQL Server uses the virtual memory management features of Windows to enable it and other processes to share the physical memory on the server and to hold memory pages on disk in a page file.

Physical memory is a system resource. The more physical memory the better the system is likely to perform. If there is not enough physical memory on the server, then performance will be degraded as processes fight for memory. This section provides an overview of the Windows virtual memory model and looks at how SQL Server uses memory. It then looks at how memory bottlenecks can be observed.

9.2.1 An overview of Windows virtual memory management

Similar to a number of modern server operating systems, Windows uses a flat, linear memory model. Each process is able to address 4 GB of virtual memory. The upper 2 GB of virtual memory are reserved for system code and data, which are accessible to the process only when it is running in privileged mode. The lower 2 GB are available to the process when it is running in user mode. However, SQL Server Enterprise Edition provides support for using Windows Address Windowing Extensions (AWEs).

Information held in physical memory can usually be categorized as either code or data. The pages given to a Windows process by the virtual memory manager are known as the working set of the process, and this working set holds pages containing any code and data recently used by the process. The working set of a process can grow or shrink as the virtual memory manager transfers pages of code and data between hard disk and physical memory. This is known as paging. All virtual memory operating systems page, and the secret is to make sure that the amount of physical memory and the memory requirements of processes are such that paging does not become a burden on the system. In this situation, paging can cause disk bottlenecks and start to consume the processor.

If a page of code or data is required by a process, and it is not present in the working set of the process, a page fault results. The page is then brought into its working set. Whether the working set of the process then grows is determined by the availability of free memory on the server. If there is an abundance of free memory, the working set of the process will grow as the new page is added. If there is a lack of free memory, pages in the working set that have not been used for a while will be removed. This is known as working set trimming. If pages are continually being taken out of the working set of a process to make room for new pages, it is likely that the removed pages will be needed again soon. The process will again page fault and the cycle will be repeated.

We can see that if memory is running low, code and data pages will be continually removed from, and added to, the working set of the process, resulting in many page faults. This can lead to a disk bottleneck and wasted CPU, since the system spends more time paging than doing useful work on behalf of the user.

There are two types of page fault. A hard page fault happens when the code or data page needs to be retrieved from disk. A soft page fault happens when it is discovered elsewhere in physical memory. Soft faults use CPU, but hard faults cause disk reads and writes to occur.

When a page is removed from the working set, it may need to be written to disk if it has been changed. If it has not been changed, this need not happen. The area on disk that pages are read from and written to is known as the page file. The file name of the page file is pagefile.sys, and its default size is equal to 1.5 times the amount of physical memory. If memory is committed to a process (known as committed memory), space will be reserved for it in the page file.

9.2.2 How SQL Server uses memory

An instance of SQL Server is a single Windows process as is an instance of the SQL agent process that manages components such as the replication and alert subsystems. The amount of memory you can give to SQL Server really depends upon the amount of memory available on your Windows server, and this is a function of the amount of physical memory on the server and the memory requirements of other processes running on the server. Ideally, if it is possible, dedicate a single Windows Server to run a single instance of SQL Server, and then SQL Server will not compete for memory resources with anything else. Of course, it can compete with Windows 2003 itself for memory, but this will degrade performance and so the dynamic memory configuration in SQL Server leaves free memory for the operating system. If you decide to configure the memory requirements of SQL Server manually, you are advised to leave ample memory for the operating system.

Remember that multiple instances of SQL Server can run on one Windows server—a default instance with up to 16 named instances. Each of these instances will compete for memory.

So what is memory used for in an instance of SQL Server? The short answer is lots of things. There is a pool of 8 KB buffers that are used for database pages—for example, data and index pages and also query plans. Memory is required for user connections and locks. Most importantly, memory is required for the queries themselves.

Different queries can have very diverse memory requirements. A simple query such as a single row lookup will require little memory to execute. Such queries are typically found in online transaction processing systems (OLTPs). Other queries, such as the ad hoc queries found in data warehouse type systems, may need to perform large sorts. Some queries will need to perform hash joins on large amounts of data. The queries that need to sort and hash will benefit from lots of memory. If the sort can fit into memory, or the hash buckets can fit into memory, query performance will be improved.

When the query optimizer creates a plan for a query, it calculates the minimum memory a query will need and the maximum amount of memory it would benefit from. When a query needs to be executed, it is passed to a special scheduler. This scheduler checks to see if the query indeed does perform a sort or hash operation. If it does not, it is scheduled to run immediately. Queries that have a sort or hash operation will then be scheduled based on their memory requirements. Queries with small sorts or joins will be scheduled almost immediately. Queries with large sorts or joins will be scheduled in such a way that only a few can run concurrently.

9.2.2.1 Configuring memory for SQL Server

SQL Server will dynamically configure its memory requirements. It will expand to use up the free memory on the Windows server as long as it needs memory and that amount of memory is available on the server. It will not use all the free memory, since some will be needed by the operating system—typically under 10 MB. As other processes start up and need memory, the available free memory will drop and SQL Server will then release memory.

Two server configuration options, min server memory (MB) and max server memory (MB), can be used to specify upper and lower bounds for the memory an SQL Server instance will use. When the instance is started, it takes as much memory as it needs to initialize. This may well be below the min server memory (MB) value. However, once it has crossed this value, it should not drop below it. This ensures that even if the instance is not busy, some memory will be kept ready for starting queries. This ensures that their performance is not degraded by the instance trying to suddenly acquire memory it has given up. The max server memory (MB) value places an upper limit on the memory the instance will use.

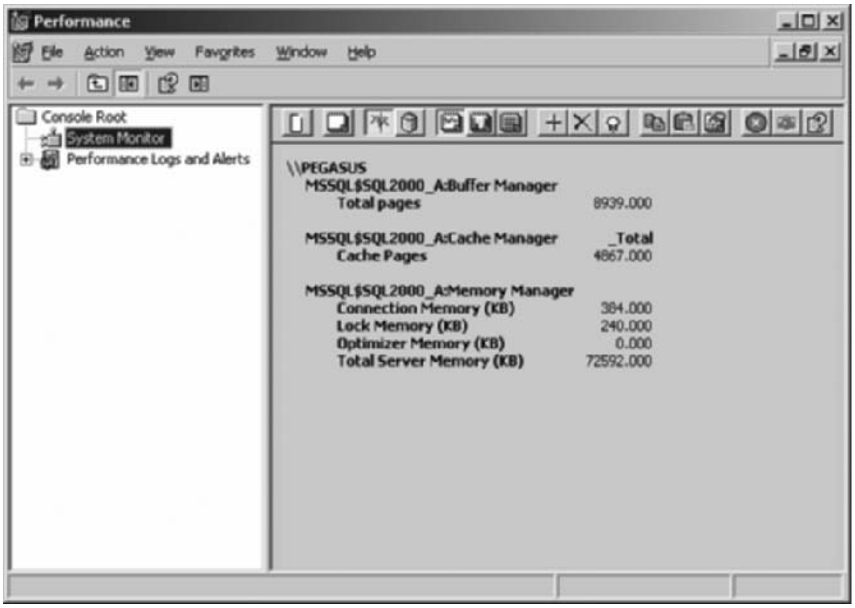
These two server options can be set so that their values are equal. In this situation, once the instance has grown its memory to that value, it should not increase or decrease it.

These server configuration options can be set with the system stored procedure sp_configure or with the SQL Server Management Studio.

The slider controls that set the min server memory (MB) and max server memory (MB) server configuration option values can be seen. These can be adjusted and are meaningful when the Dynamically configure SQL Server memory option is selected. If preferred, the Use a fixed memory size (MB) option can be selected, which effectively sets min server memory (MB) and max server memory (MB) values equal and stops dynamic configuration.

Once the server has been allocated memory, it uses it for a variety of objects—for example, user connections, locks, and the buffer pool (cache).

There are various methods to investigate the apportionment of memory. The System Monitor has a number of objects and counters to help us. Figure 9.10 shows the System Monitor in report format displaying some useful object counters.


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts

```
\\PEGASUS
  MSSQL$SQL2000_A:Buffer Manager
      Total pages                     8939.000

  MSSQL$SQL2000_A:Cache Manager          _Total
      Cache Pages                     4867.000

  MSSQL$SQL2000_A:Memory Manager
      Connection Memory (KB)           384.000
      Lock Memory (KB)                 240.000
      Optimizer Memory (KB)              0.000
      Total Server Memory (KB)       72592.000
```

Figure 9.10 *System Monitor counters useful for checking memory use of objects*

In Figure 9.10 we can see three objects—Buffer Manager, Cache Manager, and Memory Manager. They belong to the instance of SQL Server named SQL2000_A. Some useful counters belonging to these objects are displayed.

The Buffer Manager: Total Pages counter represents the total number of 8-KB pages (buffers) in the buffer pool. This holds, for example, database pages and stored procedure query plans. There are currently 8,939 buffers in the pool.

The Cache Manager: Cache Pages counter, for the _Total instance, represents the total number of 8-KB pages (buffers) in the buffer pool used by cached objects, such as stored procedure plans, trigger plans, prepared SQL plans, and ad hoc SQL plans. If required, the number of pages used by each of these cached object types can be monitored individually. There are currently 4,867 pages used for cached objects.

The Memory Manager: Connection Memory (KB) counter represents the amount of memory in kilobytes used by connections. There are currently 384 KB used by connections. Generally, a new connection will take about 24 KB depending on the network packet size. The formula for connection memory is: ((3 * the network packet size) + 12 KB), with the default network packet size being 4 KB.

The Memory Manager: Lock Memory (KB) counter represents the amount of memory in kilobytes used by locks. There are currently 240 KB used by locks. Generally, a lock will take about 96 KB.

The Memory Manager: Optimizer Memory (KB) counter represents the amount of memory in kilobytes used for query optimization. There is no query optimization being performed at the time of the monitoring.

The Memory Manager: Total Server Memory (KB) counter represents the amount of dynamic memory that the instance is currently using. We can see that if we add up the Buffer Manager: Total Pages counter (remember, each page is 8 KB) and the Memory Manager counters, the value is not far from 72,592 KB. The figure arrived at is less, because we have not monitored all consumers of dynamic memory.

Another useful tool is DBCC MEMUSAGE. This has not been documented since SQL Server 6.5, and its output has changed dramatically since then. However, if we use it with that thought in mind, we get the following output:

```
dbcc memusage (names)

Buffer Cache Top 20

Database Name     Object Name      Index Name     Buffers    Dirty


BIG               accounts         5556           0
Master            syscharsets      33             0
Master            syscomments      24             0
Master            sysmessages      14             0
BIG               accounts         UNKNOWN        11         0
:
:
```

This gives us an insight into the number of data and index pages used by the largest objects in cache.

To look at the sizes of cached objects, such as stored procedure plans, the syscacheobjects system table can be queried. Here is a fragment of output showing the pages used by different objects in cache.

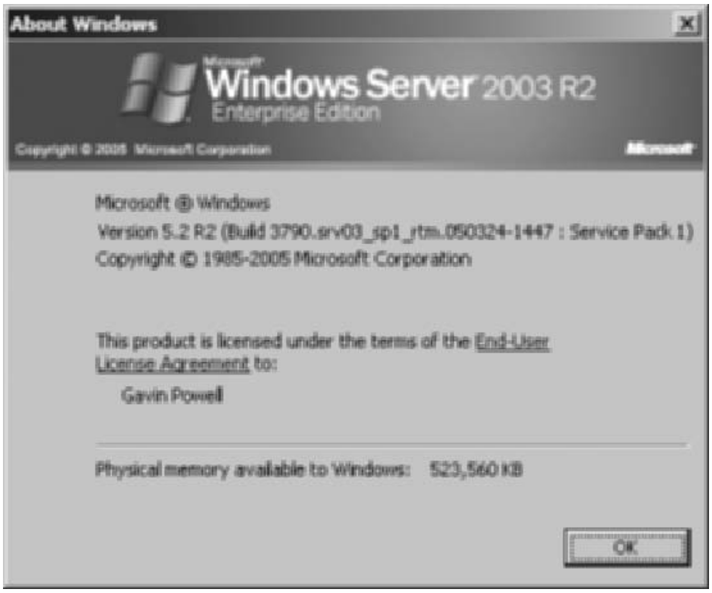
```
SELECT cacheobjtype, objtype, pagesused, sql
FROM master..syscacheobjects
```

```
ORDER BY pagesused DESC

Cacheobjtype      objtype  pagesused   sql

Executable Plan  Proc      2164   usp_test
Compiled Plan     Proc      206    usp_test
Compiled Plan     Proc  52 sp_helpdb
Executable Plan  Proc  42 sp_helpdb
Compiled Plan     Proc  31 sp_helpconstraint
```

9.2.3 Investigating memory bottlenecks

If memory starts to get tight on the server, performance will start to suffer. This is most likely to happen on a server that is running applications other than just SQL Server, since they will contend for memory.

Before we investigate memory bottlenecks, we need to look at the tools we can use to do so. The first piece of information we will want to know is likely to be how much physical memory the server has. We can easily check this by choosing About Windows from the Help menu in Windows Explorer, as shown in Figure 9.11.



About Windows

Microsoft Windows Server 2003 R2 Enterprise Edition

Copyright © 2005 Microsoft Corporation

Microsoft ® Windows
Version 5.2 R2 (Build 3790.srv03_sp1_rtm.050324-1447 : Service Pack 1)
Copyright © 1985-2005 Microsoft Corporation

This product is licensed under the terms of the End-User License Agreement to:
Gavin Powell

Physical memory available to Windows: 523,560 KB

Figure 9.11 *Memory available as shown by Windows Explorer*

Another handy tool is the Task Manager, which is present in Windows. There are a number of tabs that can be chosen, and these are Appli-

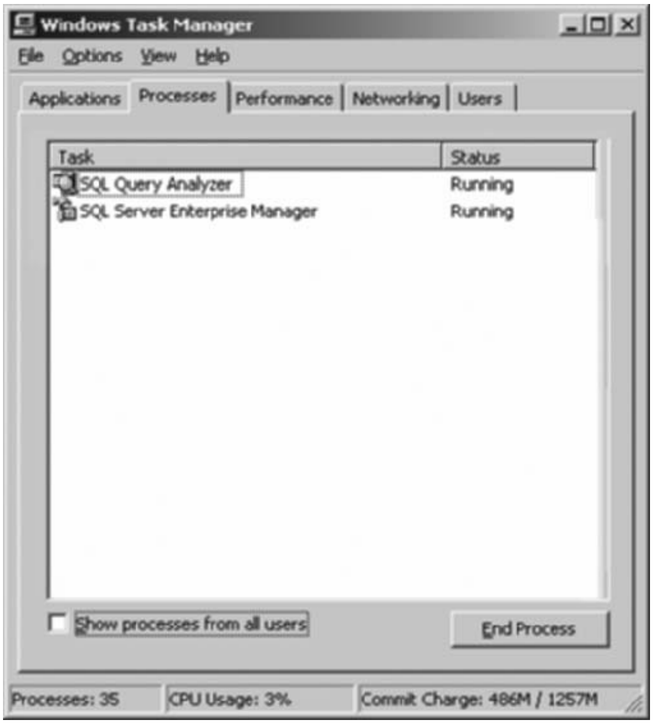

Windows Task Manager
File Options View Help
Applications Processes Performance Networking Users
Task
Status
SQL Query Analyzer
Running
SQL Server Enterprise Manager
Running
Show processes from all users
End Process
Processes: 35
CPU Usage: 3%
Commit Charge: 486M / 1257M

Figure 9.12 *The Windows 2003 Task Manager applications tab*

cations, Processes, and Performance. The Applications tab is shown in Figure 9.12.

This tab shows the status of programs that are running on the system. SQL Server is not shown, since it is running as a service. The Processes tab displays information about processes that are running on the system, as shown in Figure 9.13.

Information such as the memory usage and the page faults is shown for each process. Columns can be added or removed from this tab. The Performance tab, shown in Figure 9.14, displays a graph of CPU and memory use history as well as a textual display.

The most useful tool is the System Monitor, which we have already met. There are a number of useful System Monitor objects concerning memory, such as Memory and Process.

Let us now focus on using the System Monitor to investigate memory bottlenecks. The memory object is a useful place to start, and it is worthwhile to look at some of the memory object's counters, as shown in Table 9.2.

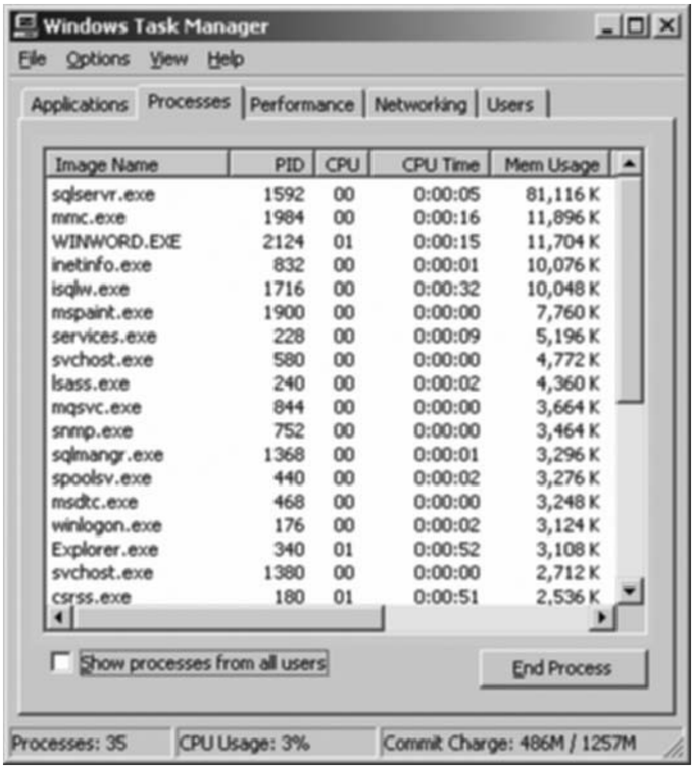


Figure 9.13
The Windows Task Manager processes tab

Image Name	PID	CPU	CPU Time	Mem Usage
sqlservr.exe	1592	00	0:00:05	81,116 K
mmc.exe	1984	00	0:00:16	11,896 K
WINWORD.EXE	2124	01	0:00:15	11,704 K
inetinfo.exe	832	00	0:00:01	10,076 K
isqlw.exe	1716	00	0:00:32	10,048 K
mspaint.exe	1900	00	0:00:00	7,760 K
services.exe	228	00	0:00:09	5,196 K
svchost.exe	580	00	0:00:00	4,772 K
lsass.exe	240	00	0:00:02	4,360 K
mqsvc.exe	844	00	0:00:00	3,664 K
snmp.exe	752	00	0:00:00	3,464 K
sqlmangr.exe	1368	00	0:00:01	3,296 K
spoolsv.exe	440	00	0:00:02	3,276 K
msdtc.exe	468	00	0:00:00	3,248 K
winlogon.exe	176	00	0:00:02	3,124 K
Explorer.exe	340	01	0:00:52	3,108 K
svchost.exe	1380	00	0:00:00	2,712 K
csrss.exe	180	01	0:00:51	2,536 K

Processes: 35 CPU Usage: 3% Commit Charge: 486M / 1257M

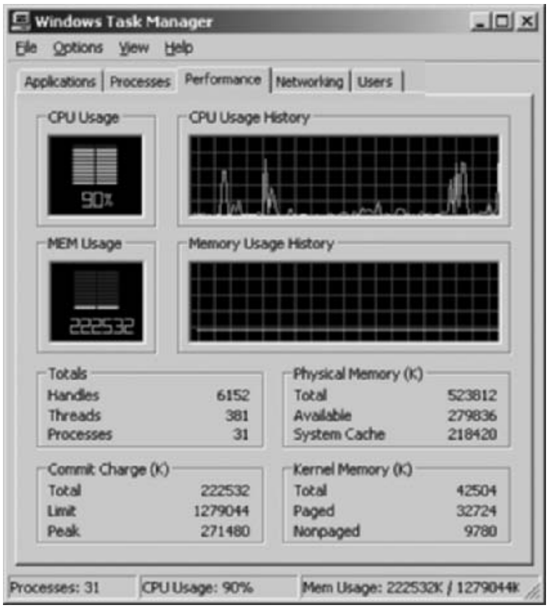


Figure 9.14
The Windows Task Manager performance tab

Totals	
Handles	6152
Threads	381
Processes	31

Physical Memory (K)	
Total	523812
Available	279836
System Cache	218420

Commit Charge (K)	
Total	222532
Limit	1279044
Peak	271480

Kernel Memory (K)	
Total	42504
Paged	32724
Nonpaged	9780

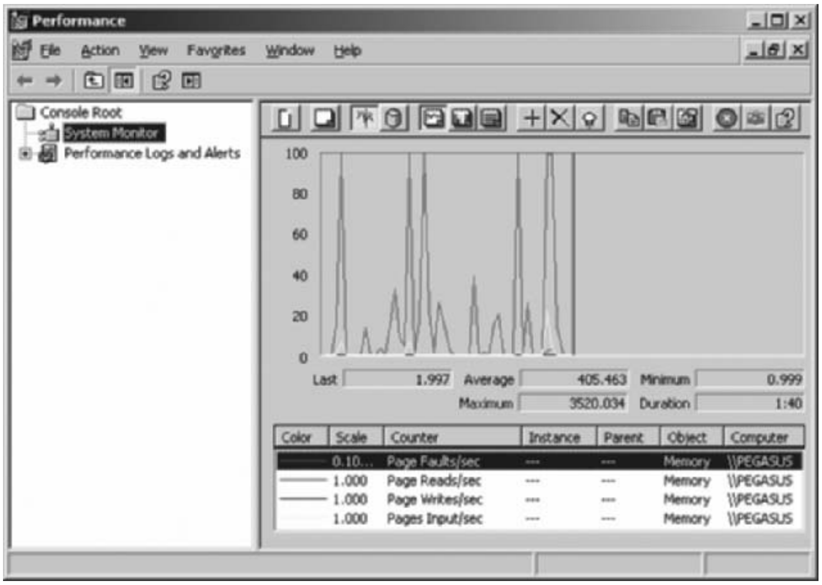
Processes: 31 CPU Usage: 90% Mem Usage: 222532K / 1279044K

Table 9.2 *Selected counters for the Memory Object*

Memory Object Counter	**Explanation**
Page Faults/sec	This counter includes both hard page faults and soft page faults. Hard page faults result in disk I/O. Soft page faults mean pages are found elsewhere in memory.
Pages Input/sec	This is a measure of the number of pages brought in from disk every second. The difference between this value and Page Faults/sec represents soft page faults.
Pages Output/sec	This is a measure of the number of pages written to disk every second to make room in the working set of the process for newly faulted pages. If the process modifies pages, they must be written out. They cannot be discarded.
Pages/sec	This is total of Pages Input/sec plus Pages Output/sec.
Page Reads/sec	This indicates the reads from disk per second to satisfy page faults. This is an important counter. As a rule of thumb, if this counter exceeds five pages per second there is a memory shortage. A single read operation can actually bring in more than one page.
Page Writes/sec	This indicates the writes to disk per second to satisfy page faults. This is another important counter, since it measures real disk I/O work being done by the system because of page faulting. A single write operation can actually write out more than one page.
Available Bytes Available KBytes	This shows how much memory remains that can be given to processes. The three counters only differ in the units used.
Available MBytes	

In Figure 9.15 the System Monitor is being used to monitor the following counters: Memory: Page Reads/sec. Memory: Page Writes/sec. Memory: Pages Input/sec. Memory: Page Faults/sec. The line that peaks the highest is Page Faults. This is to be expected, since it represents both hard and soft faults.

The averages for these counters are shown in the following chart (the averages cannot be deduced from the screenshot alone).


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
80
60
40
20
0

Last 1.997 Average 405.463 Minimum 0.999
Maximum 3520.034 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
	0.10...	Page Faults/sec	---	---	Memory	\\PEGASUS
	1.000	Page Reads/sec	---	---	Memory	\\PEGASUS
	1.000	Page Writes/sec	---	---	Memory	\\PEGASUS
	1.000	Pages Input/sec	---	---	Memory	\\PEGASUS

Figure 9.15 *Memory pages being read in from disk*

```
Counter                Average
Page Reads/sec         0.2
Pages Input/sec        0.8
Page Faults/sec        405
```

The Page Faults/sec counter represents the sum of hard and soft page faults. The Pages Input/sec counter represents hard faults, so about 0.2 percent of the faults are hard faults. The 0.8 pages that are input per second are brought in by 0.2 page reads per second, so approximately four pages are being brought in by every disk read. Although the majority of page faults are soft, 0.2 I/Os per second are hitting the disk to retrieve pages, which is trivial.

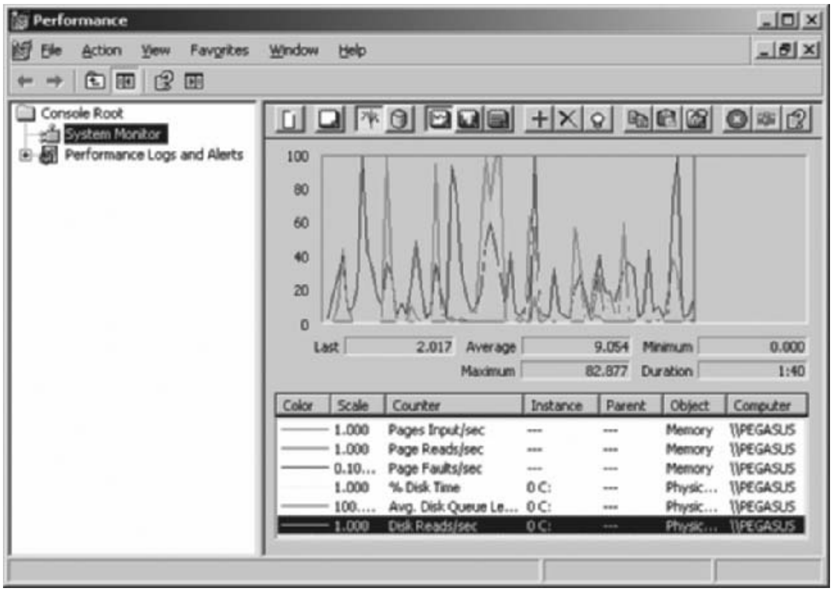
It is useful to also examine the disk activity to see how hard paging is hitting the disks. Some useful counters are as follows:

- % Disk Time
- Avg. Disk Queue Length
- Disk Reads/sec

The % Disk Time is the percentage of elapsed time that the selected disk drives are busy servicing requests. Avg. Disk Queue Length is the average number of read and write requests queued on the selected disks. Disk

Reads/sec is the rate of read operations on the disk. These are shown in Figure 9.16. The averages for these counters are shown in the following chart.

```
Counter                    Average
Page Reads/sec             0.4
Pages Input/Sec            1.6
Page Faults/sec            282
% Disk Read Time           23.8
Avg. Disk Queue Length     0.2
Disk Reads/sec             9
```


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
80
60
40
20
0

Last	2.017	Average	9.054	Minimum	0.000
		Maximum	82.877	Duration	1:40

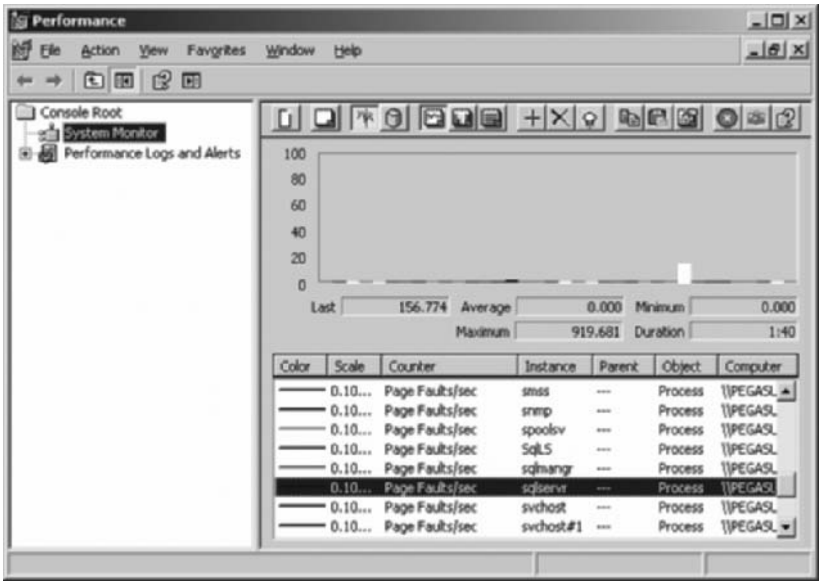
Color	Scale	Counter	Instance	Parent	Object	Computer
—	1.000	Pages Input/sec	---	---	Memory	\\PEGASUS
—	1.000	Page Reads/sec	---	---	Memory	\\PEGASUS
—	0.10...	Page Faults/sec	---	---	Memory	\\PEGASUS
	1.000	% Disk Time	0 C:	---	Physic...	\\PEGASUS
—	100....	Avg. Disk Queue Le...	0 C:	---	Physic...	\\PEGASUS
—	1.000	Disk Reads/sec	0 C:	---	Physic...	\\PEGASUS

Figure 9.16
Memory counters with disk counters

We can immediately compare Page Reads/sec with Disk Reads/sec. This shows us that only a small part of our disk activity is caused by paging. The disk is busy about 24 percent of the time. The Avg. Disk Queue Length is small: about 0.2.

A similar investigation can be performed for page writes. It is also worth looking at which individual processes are faulting heavily. This can be done by monitoring the Page Faults/sec counter on the process object for all the process instances. If this is viewed in histogram format, processes that are page faulting heavily stand out immediately, as shown in Figure 9.17.

Another area worth monitoring is the page file, to see if it is filling. Ensure that there is enough free space to let it expand if it needs to.


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
80
60
40
20
0

Last 156.774 Average 0.000 Minimum 0.000
Maximum 919.681 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
—	0.10...	Page Faults/sec	smss	---	Process	\\PEGASL
—	0.10...	Page Faults/sec	snmp	---	Process	\\PEGASL
—	0.10...	Page Faults/sec	spoolsv	---	Process	\\PEGASL
—	0.10...	Page Faults/sec	SqlLS	---	Process	\\PEGASL
—	0.10...	Page Faults/sec	sqlmangr	---	Process	\\PEGASL
	0.10...	Page Faults/sec	sqlservr	---	Process	\\PEGASL
—	0.10...	Page Faults/sec	svchost	---	Process	\\PEGASL
—	0.10...	Page Faults/sec	svchost#1	---	Process	\\PEGASL

Figure 9.17 *Monitoring page faults for individual processes*

9.2.4 Solving problems with memory

The two main approaches to solving memory problems are: making best use of available memory and adding more physical memory to the server.

To make more use of available memory, remove anything that is not needed but is consuming memory resources, for example, Windows services, drivers, and network protocols that are not used. As was mentioned earlier: if possible, dedicate the server to a single instance of SQL Server.

Increasing the size of the paging file and adding another paging file may help. The addition of extra memory should also be accompanied by an increase in paging file size and, if possible, an increase in secondary cache size. In my experience, the addition of more memory is often the simplest and quickest fix to memory problems and is often the most cost effective.

9.3 SQL Server and disk I/O

A bottleneck that is often experienced with database management systems concerns the disk subsystem. By definition a database is a shared repository of information, and, consequently, many users are likely to be reading and writing to the database. Depending on whether the database supports an online transaction processing (OLTP) system or a decision support system

(DSS), users may update small amounts of data or may perform read only queries on large amounts of data.

The disks themselves are different from most other components in the server in that they typically have moving parts. The disk surface rotates and the disk heads move in and out across the disk surface. Relative to memory access this takes a long time, and therefore SQL Server uses many techniques to help it minimize disk access. In fact, as we have seen, the query optimizer attempts to choose an access strategy that limits the number of disk I/Os performed.

Care should be taken when investigating disk I/O bottlenecks, since there can be many causes. One cause is a memory bottleneck, which results in high levels of paging to disk, as was described in the previous section.

9.3.1 An overview of Windows and disk I/O

To perform its disk I/O, SQL Server issues reads and writes to Windows and lets Windows deal with the business of reading and writing to the underlying disk subsystem. Various techniques are employed to keep the physical disk I/Os efficient. For example, Windows utilizes a technique known as scatter-gather I/O. This technique enables Windows to transfer data into or out of areas of memory, which are not contiguous, in a highly efficient fashion. Unlike Windows XP, Windows Server can also make use of asynchronous I/O, which gives SQL Server the ability to issue I/Os to disk and, instead of waiting for the I/O to complete, carry on with other work. The I/O completion can then be checked later.

To provide high levels of disk I/O throughput, Windows provides various levels of RAID (Redundant Arrays of Inexpensive Disks), and SQL Server can make use of this capability. Various vendors also provide hardware-based RAID solutions. These increase the cost of the system but tend to provide better performance and are becoming increasingly popular. For that reason, we will assume we are using hardware-based RAID arrays.

Commonly supported RAID levels are as follows:

- RAID 0—disk striping
- RAID 1—disk mirroring
- RAID 5—disk striping with parity

In a RAID 0 stripe set, data is spread across all the drives in the set. If you were to create a database file on a RAID 0 stripe set, the disk controller would actually break the file into pieces (known as chunks) as you created it. Each piece would be placed on the next disk in the set circling round when it moved off the last one. We can imagine a three-disk stripe set now providing three sets of disk heads to access the file. This is the bonus of RAID 0: performance. RAID 0 provides very good performance for both reading and writing. The downside of RAID 0 is that the loss of a single disk will affect the whole stripe set. The RAID 0 array will appear to be a single disk to Windows and SQL Server.

RAID 5 is very similar to RAID 0. However, as well as writing data onto a disk drive in the stripe set, parity information is written to another stripe set member. Not only do we stripe data, but we stripe parity information. This gives us a level of redundancy. We can lose one disk and the data information on that disk can be recreated from the parity on other disks when a request for data on the failed disk is made. The downside of RAID 5 is that although read performance is good, write performance is worse than RAID 0, since two disks must be written to. Hardware-based implementations of RAID 5 can help to absorb this write performance degradation. Again, the RAID 5 array will appear to be a single disk to Windows and SQL Server.

In RAID 1 data is duplicated on a mirror disk drive (some RAID implementations allow more than one mirror). Writes are performed to both members of the set. This configuration gains us redundancy. We can lose one of the members and still continue working with the other one. There is no performance advantage in using RAID 1 for writing; in fact, it can be slightly slower, but it may well give some performance boost to reading. A downside of RAID 1 is that twice as much disk space is necessary and, therefore, twice the cost.

It is also possible to use two disk controllers—one for each mirror set member. This means that a disk controller failure can be tolerated. This is known as duplexing. As with the other RAID configurations, the RAID 1 array will appear to be a single disk to Windows 2003 and SQL Server.

Table 9.3 summarizes the different RAID levels.

What happened to RAID levels 2, 3, and 4? Generally, these are considered to be evolutionary steps toward RAID 5 and thus are not often used with database systems.

Table 9.3 *RAID levels 0, 1, and 5*

Number of				
RAID Type	**Characteristics**	**Disks**	**Reliability**	**Performance**
RAID 0: disk striping	Data is spread over all the disks in the stripe set with no redundancy.	N	Less than a single disk.	High for read and write.
RAID 1: disk mirroring	Data duplicated on each member.	2N	Higher than RAID 0 or 5.	Good for read but less than a
			or single disk	Single disk for write.
RAID 5: disk striping with parity	Similar to RAID 0, but parity information is stored with data for redundancy.	N + 1	Higher than RAID 0 or single disk.	Similar to RAID 0 for read but less than a single disk for write.

Note: Choosing the appropriate RAID implementation is a compromise between performance, fault tolerance, and cost. Figures 9.19 and 9.20 show two common configurations.

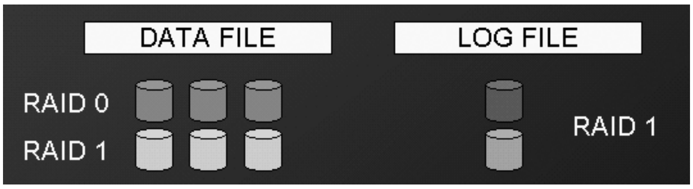


Figure 9.18 *A RAID configuration utilizing RAID 0 and 1 for the data and RAID 1 for the log*

Note: Both of the configurations store the log on a separate RAID array from the data using a separate disk controller. This means that the data file can be lost while the transaction log remains unaffected.

The configuration in Figure 9.18 places the data file on a RAID 0 array for optimum read and write performance. The RAID 0 array is mirrored to provide fault tolerance. This is often known as RAID 1+0, or RAID 10. This provides the best performance and fault tolerance but at the greatest cost. The transaction log is placed on a RAID 1 array. The transaction log is usually written to sequentially so, as long as nothing competes for disk bandwidth on this array, this configuration provides good write performance (and read). The transaction log is mirrored, since losing it may result in the loss of work.

The configuration in Figure 9.19 places the data file on a RAID 5 array. This will provide optimum read performance, but write performance will be degraded. This will be a lower-cost solution than the previous configuration. The transaction log is placed on a RAID 1 array as before.

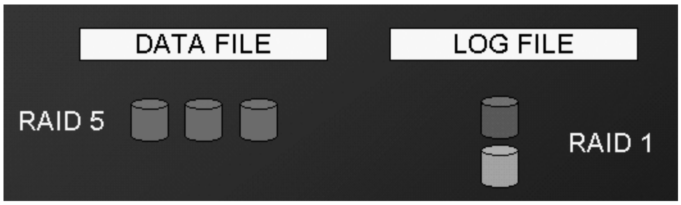


Figure 9.19 *A RAID configuration utilizing RAID 0 and 1 for data, plus RAID 1 for the log*

Suppose the size of our data was greater than the size of the RAID arrays available to us. In this case we could use multiple data files, placing each file on each RAID array. Space for our tables would be allocated from each file on each RAID array. SQL Server would be able to issue read requests simultaneously to each RAID array when the table was scanned.

9.3.2 How SQL Server uses disk I/O

We have already mentioned the fact that SQL Server maintains a pool of 8 KB buffers. This buffer pool is sometimes referred to as a unified cache, since it holds both cached objects, such as stored procedure plans, and database pages, such as data and index pages. The buffers used for cached objects are often referred to as the procedure cache, and the buffers used for database pages are referred to as the data cache.

The goal of the data cache is to minimize physical accesses to the disk subsystem. There is a single data cache for each instance of SQL Server that all the instances databases share. In this section we will look at the data cache and the various techniques used to make reading from it and writing to it more efficient.

9.3.2.1 An overview of the data cache

As we discussed earlier, a portion of SQL Server memory is used for the data cache. As long as there is enough memory available on the server to allow SQL Server to dynamically grow its memory allocation, the data cache can grow.

The idea behind the data cache is quite simple. If a user connection requests a row, SQL Server will translate this into a page request, and it will then look for the page in the data cache to see if this page has previously been retrieved from disk. This request for a page represents a logical read.

If the page cannot be found, it must be retrieved from the database on disk, and this disk access represents a physical read. The page is read into a free buffer, and the data requested by the connection obtained. The page is now in cache, and, assuming that it does not leave the cache for any reason, it will be available for any connection requesting it. The next connection requesting that page will issue a logical read, which will be satisfied from the data cache. This is a memory access, as opposed to a disk access, and is consequently much faster than the original request that brought in the page from disk.

We can envision a situation where a whole database gets brought into the cache, and this is quite feasible—the only limiting factor being the size of the data cache. In reality, 20 percent of most databases get accessed 80 percent of the time, so we find that the most accessed pages in the database find themselves in the data cache. Note that increasing the size of the data cache does not bring us a linear performance increase. Once we can hold the most accessed pages in a database or group of databases in the data cache, the allocation of more memory to the data cache brings us little gain.

An empty data cache is created when SQL Server is started. At this point most database page requests end up as physical reads. After a while a steady state is reached, with the data cache holding the most frequently used pages, as shown in Figure 9.20.

As shown in Figure 9.20, the percentage of time a requested database page is found in the data cache is known as the cache hit ratio. The cache hit ratio is defined as follows:

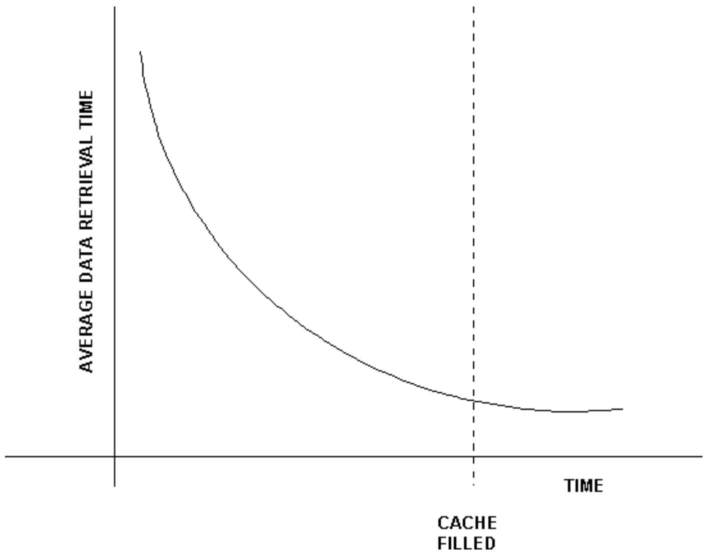


Figure 9.20
A steady state reached in the data cache

```
cache hit ratio (%) = ((logical read - physical read)/
logical read) * 100
```

What happens if we fill the data cache and then we need to read in a new page? We will discuss the mechanisms employed shortly, but SQL Server will have to make room in the data cache for the new page. If the new page has been changed by a user connection, then it is known as a dirty page and it cannot be discarded, because it reflects the latest state or version of that page. It must be written back to the database on disk. However, if the page has not been changed, it can be discarded. SQL Server keeps track of which pages have not been used for the longest length of time. This is important, because this is taken into account when SQL Server jettisons pages from the cache.

How does SQL Server find out if a page is resident in the data cache? It could look at every used buffer in the data cache, but this would be very expensive for large data caches consisting of tens of thousands of buffers. Instead, it uses an internal hashing scheme to quickly locate buffers.

What happens if we change pages in the data cache? How do they get to disk? There are a number of mechanisms involved. First of all, we need to consider the fact that usually the data cache is finite in size and eventually all the buffers in it could be used. In other words, there are no free buffers. If there are no free buffers, then SQL Server has no room to place new pages that are read in from disk. To avoid and preempt this situation, SQL

Server periodically frees up buffers in the data cache. When a buffer is freed, it is first checked to see if it is dirty. A dirty page is one where changes have not yet been written to disk and therefore the buffer cannot just be discarded. The dirty page must be written to the data file. If the page is not dirty, then its contents can be discarded and the buffer is placed into a chain of free buffers.

It would not make sense to free a buffer containing a page that was frequently accessed instead of a buffer containing a page that had not been accessed for a long time. To avoid this situation, each buffer contains a reference count, which is incremented each time the page in the buffer is accessed. The more the page is accessed, the greater the reference count. When the data cache is searched in order to find buffers that can be freed, the reference count is decremented. When a buffer is found with a reference count of zero, it is freed. This mechanism ensures that frequently accessed pages stay in the cache. Of course, if we have a large data cache and lots of memory on the server so that the data cache can expand, there is no reason to free up buffers constantly.

Note: SQL Server uses a write-ahead log algorithm. This means that the transaction log is always written to before the data file, and this ensures that a change can always be rolled back in a recovery situation.

So what writes the dirty pages to disk? There is no one process that does this. Often it is the worker threads that perform the function of scanning the buffer pool looking for pages to discard. They do this while waiting for their own disk accesses to complete. If they need to write a page, this is performed as an asynchronous I/O.

A system process known as the lazywriter also performs the same function. The lazywriter thread is activated at periodic intervals. It then scans the data cache in order to find buffers that can be freed. It basically performs the same activities at the worker threads. Because the worker threads have been freeing up buffers, the lazywriter system process is not kept busy. However, on the Windows XP platform, where asynchronous I/O is not supported, the worker threads cannot perform this function and therefore the lazywriter system process can become very busy.

Another system process that contributes is the checkpoint process. The checkpoint thread's goal in life is not to free up buffers but rather to ensure that the contents of dirty pages eventually get written to the data files on disk. It does this to keep recovery time short; otherwise, an automatic SQL

Server recovery, performed perhaps because of a power failure, would potentially take a long time rolling forward changes from the transaction log to the data files. The checkpoint thread writes the pages asynchronously to disk with what are sometimes referred to as batch writes. This is a very efficient mechanism, especially if it is used in conjunction with hardware-based RAID arrays.

To monitor the lazywriter and checkpoint processes, SQL Server provides us with a number of useful counters associated with the Buffer Manager object, as shown in Table 9.4.

Table 9.4 *Some useful counters for observing page transfers to and from disk*

Lazywrites/sec	**Number of Buffers Written per Second by the Lazywriter**
Checkpoint Pages/sec	Number of pages flushed to disk per second by a checkpoint
Page Reads/sec	Number of physical database page reads per second
Page Writes/sec	Number of physical database page writes per second
Database Pages	Number of database pages in the buffer pool
Free Pages	Number of free pages

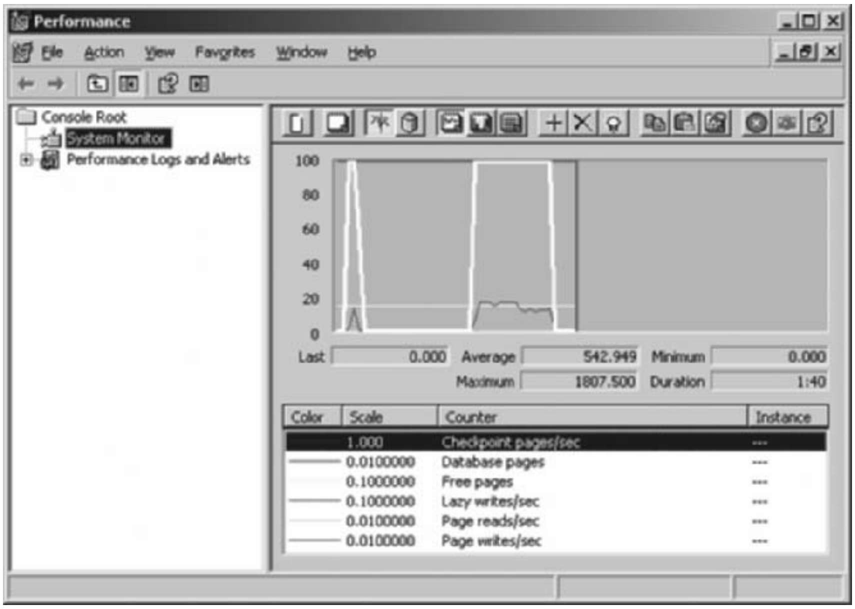
Another Buffer Manager counter that is very useful is Buffer Cache Hit Ratio. This is the cache hit ratio described previously.

Figure 9.21 shows checkpoint activity occurring on the server. The highlighted counter is the Checkpoint pages/sec counter. Notice that during the checkpoint, another counter is also active. This is the Page Writes/sec counter. In this example both counters had a maximum value of 1,807.

9.3.2.2 Keeping tables and indexes in cache

As described previously, tables and indexes that are accessed frequently stay in the data cache, while other, least used pages are flushed out first. In this way the pages that are often required are the pages that connections get fast access to. However, it is possible that fast access is required to tables and indexes that are not accessed frequently enough to keep them in the data cache.

To keep a table and its indexes in data cache the sp_tableoption system stored procedure can be used, as follows:


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
80
60
40
20
0

Last 0.000 Average 542.949 Minimum 0.000
Maximum 1807.500 Duration 1:40

Color	Scale	Counter	Instance
	1.000	Checkpoint pages/sec	---
	0.0100000	Database pages	---
	0.1000000	Free pages	---
	0.1000000	Lazy writes/sec	---
	0.0100000	Page reads/sec	---
	0.0100000	Page writes/sec	---

Figure 9.21 *Checkpoint activity observed in the System Monitor*

```
EXEC sp_tableoption 'branches', 'pintable', true
```

Note that the table name can use wildcard characters. This statement does not load pages from the table into the data cache, but once they are read into data cache by normal activity, they stay there and are not removed. This can result in little data cache being left for other tables and indexes, so table pinning should be used with care.

To turn the option off, just use the false keyword, as follows:

```
EXEC sp_tableoption 'branches', 'pintable', false
```

9.3.2.3 Read-ahead scans

Read-ahead processing is a mechanism used by SQL Server to reduce the number of stalls a thread experiences waiting for a physical read to complete. It is a concept similar to instruction prefetch in a CPU. If SQL Server realizes that a table scan or an index scan is taking place—in other words, sequential scanning of pages—it can start to prefetch pages into the data cache before the thread requests those pages. This means that when the thread requests a page, it is found in the data cache, and the thread does not stall waiting for a physical read from disk to complete.

If a read-ahead mechanism was not employed, a thread issuing many disk I/Os while executing a table scan or index scan would spend a large amount of time waiting for the disk read to complete, as shown in Figure 9.22. We know that disk I/O takes a long time relative to memory access, and this is represented by 't' in Figure 9.22.

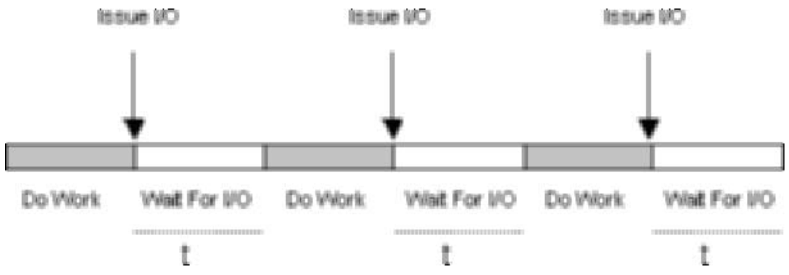


Figure 9.22
Performing a table scan with no read ahead

If we employ a read-ahead mechanism, which can read the pages into cache using other threads before the user's thread requests them, we have eliminated the stall caused by the physical read and only the data cache access is required, as shown in Figure 9.23.

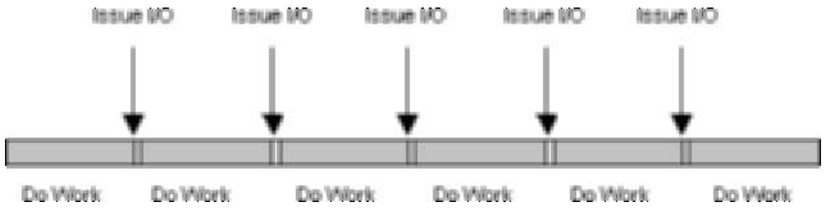


Figure 9.23
Performing a table scan with read ahead

The read-ahead mechanism also reads in units of extents, so it reads in eight pages in one disk I/O, which clearly is more efficient than reading eight pages with eight single-page reads.

So what can we benefit from the read-ahead capability? Basically, anything that performs a sequential scan of data pages, including the following:

- Table scans
- Non-clustered index leaf scans
- DBCC statements, such as DBCC CHECKDB
- Transact-SQL statements, such as UPDATE STATISTICS

How does SQL Server know, for example, that a table scan is taking place? It knows because that was the decision the query optimizer made.

How does SQL Server know which pages to read next? Because the extents in a table or index are managed by IAM pages SQL Server can easily retrieve the relevant IAM page or pages and find the extents that need to be read. A sorted list is then built of the extents to be read and this drives the read ahead. Contiguous extents can then be read very efficiently.

To observe read-ahead processing in action, the Set statistics IO option can be set in the Query Analyzer. For example, suppose we execute the following query against the Accounts table—this time increased to 400,000 rows:

```
SELECT COUNT(*) FROM accounts
```

The output from Set statistics IO is as follows:

```
Table 'accounts'. Scan count 1, logical reads 24306,
physical reads 136, read-ahead reads 24087.
```

This shows that 24,306 logical reads were required to perform the table scan but only 136 physical reads. The number of read-ahead reads performed was 24,087. This means that 24,087 pages were read into the data cache by the read-ahead mechanism. The low value of physical reads performed by this query is due to read ahead.

What happens if we immediately reissue the query:

```
Table 'accounts'. Scan count 1, logical reads 24306,
physical reads 0, read-ahead reads 0.
```

In this case the pages are already in data cache. The read-ahead mechanism is never initiated.

The System Monitor can also be used to monitor read ahead. The Buffer Manager object has an associated counter: Readahead pages/sec.

9.3.2.4 Shrinking database files

One consideration to be made when scanning the pages of a table is the utilization of the pages. If we have many pages that are only partly filled because of row deletions, perhaps made by an archive program, we are scanning more pages than should be necessary to retrieve our data. We need some way of detecting the problem and then fixing it by compacting the file.

The DBCC SHOWCONTIG statement, which we discussed in a previous chapter, can show us how densely rows are stored on pages. For example:

```
DBCC SHOWCONTIG ('accounts')
   :

- Pages Scanned............. : 1570
   :

- Avg. Page Density (full)... : 42.34%
   :
   :
```

To compact the file we can use DBCC SHRINKFILE. Previously, we noted that in the default case data rows from the pages at the end of the table would migrate to the free space in pages at the beginning of the table. Let us issue a DBCC SHRINKFILE:

```
DBCC SHRINKFILE (BankingDB_Data,10)
```

Now let us execute DBCC SHOWCONTIG again:

```
DBCC SHOWCONTIG ('accounts')

   :

- Pages Scanned............. : 782
   :

- Avg. Page Density (full)... : 84.70%
   :
   :
```

We can immediately see that the page density has increased by about a factor of two. This means we are storing twice as many rows per page and that we need half the pages to hold our data compared with what we needed previously. This is clear from the Pages Scanned value, which has changed from 1,570 to 782. So, although it may take a while to shrink a large file, you may find that subsequent scans take somewhat less time.

9.3.3 Investigating disk I/O bottlenecks

The tool used to observe disk I/O bottlenecks is typically the System Monitor. The Task Manager displays little useful information as far as disk I/O is concerned. We will focus on using the System Monitor, since it is the most comprehensive tool, and we will also introduce a useful system table-valued function, fn_virtualfilestats.

If you are using Windows, the statistics collection for the Logical-disk object is not active by default. However, the statistics collection for the Physicaldisk object is active by default.

To activate statistics collection in Windows for the Logicaldisk object, run the diskperf command and reboot Windows. To turn on statistics collection for the Logicaldisk object, type in:

```
diskperf -yv
```

To deactivate statistics collection, type in:

```
diskperf -nv
```

The Physicaldisk object uses the syntax -yd and -nd. Once the diskperf command has been run, it will not have to be run again until you want to change the statistics collection. For Windows NT the syntax is just -y and -n. Let us look at some of the more useful counters associated with disk activity.

The Logical Disk, Physical Disk, and a number of SQL Server objects are a useful place to start, and it is worth a look at some of their counters. Again, note that it often is a memory bottleneck that manifests itself as a disk bottleneck, and therefore the counters associated with the Memory object, as described earlier, should also be monitored. Some of the most useful Logical Disk and Physical Disk counters are shown in Table 9.5.

Table 9.5 *Logical and Physical Disk counters*

Logical/Physical Disk Object Counter	Explanation
% Disk Time	How busy is the disk? This is the percentage of elapsed time that the selected disk is busy handling read and write requests.

Table 9.5 *Logical and Physical Disk counters (continued)*

% Disk Read Time	This is the percentage of elapsed time that the selected disk is busy handling read requests.
% Disk Write Time	This is the percentage of elapsed time that the selected disk is busy handling write requests.
% Idle Time	This is the percentage of elapsed time that the selected disk is not processing requests.
Disk Reads/sec	The rate of read operations on the disk.
Disk Writes/sec	The rate of write operations on the disk.
Avg. Disk Queue Length	This is the average number of read and write requests for the disk in the sample interval. If disk queue length is greater than two and the %Disk Time is high, this may indicate a disk bottleneck.
Current Disk Queue Length	This is an instantaneous value at the point of sample. It includes the requests being serviced.
Avg. Disk Bytes/Read	This is the average number of bytes transferred to disk during read operations.
Avg. Disk Bytes/Write	This is the average number of bytes transferred to disk during write operations.

Useful SQL Server counters are shown in Table 9.6.

Table 9.6 *Useful SQL Server counters*

SQL Server Object Counter	Explanation
Access Methods: Forwarded Records/sec	Number of records per second fetched through forwarded record pointers
Access Methods: Full Scans/sec	Number of unrestricted table or index scans per second
Access Methods: Page Splits/sec	Number of page splits per second that occur as the result of over flowing index pages (data pages in a clustered index).

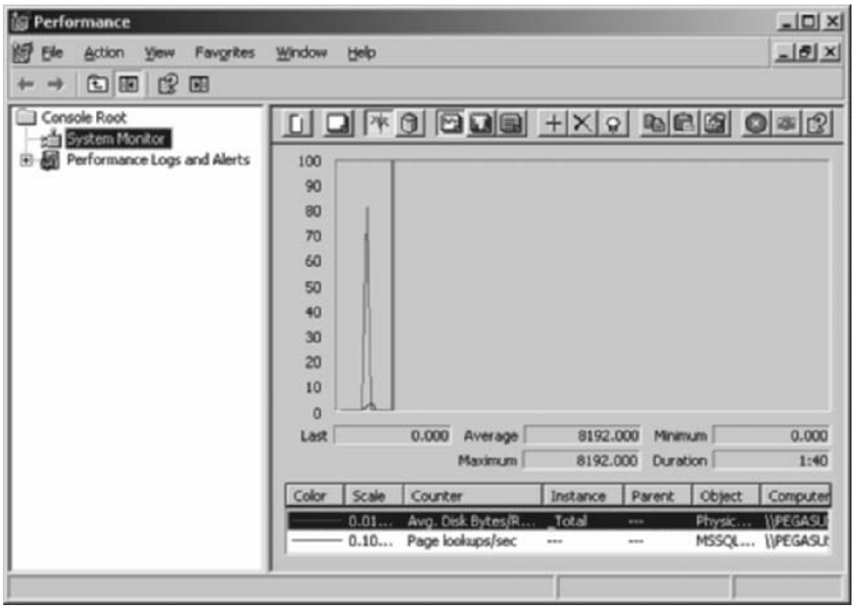
Table 9.6 *Useful SQL Server counters (continued)*

Buffer Manager: Buffer Cache Hit Ratio	The percentage of time that a page was found in the data cache. Usually 95% plus on a server in steady state with a large cache.
Buffer Manager: Checkpoint Pages/sec	Number of pages written to disk per second by a checkpoint.
Buffer Manager: Database Pages	Number of database pages in the buffer pool.
Buffer Manager: Free List Stall/sec	Number of requests per second that had to wait for a free page.
Buffer Manager: Free Pages	Total number of pages on all free lists.
Buffer Manager: Lazy Writes/sec	The number of pages written out to disk per second by the lazy-writer. This cleans buffers and returns them to the free buffer pool.
Buffer Manager: Page life Expectancy	Number of seconds a page will stay in the buffer pool without any references to it.
Buffer Manager: Page Lookups/sec	Number of requests per second to find a page in the buffer pool.
Buffer Manager: Page Reads/sec	The number of physical page reads per second. This is what we try to minimize with indexes and data cache.
Buffer Manager: Page Writes/sec	The number of physical page writes per second.
Buffer Manager: Procedure Cache Pages	Number of pages used to store compiled queries.
Buffer Manager: Readahead Pages/sec	Number of pages read in by the read-ahead mechanism.
Buffer Manager: Reserved Pages	Pages reserved in the buffer pool.
Buffer Manager: Stolen Pages	Number of pages used for miscellaneous server purposes.
Buffer Manager: Target Pages	Ideal number of pages in the buffer pool.
Buffer Manager: Total Pages	Number of pages in the buffer pool—includes database, free, and stolen pages.
Databases: Data File(s) Size (KB)	Total size of all data files in a database.
Databases: Log File(s) Size (KB)	Total size of all log files in a database.

Be aware that the % Disk Time, % Disk Read Time, % Disk Write Time, and % Idle Time counters can exaggerate. You may see values over

100 percent. It is a good idea to monitor % Idle Time with the other three counters to get an indication of whether this is happening.

In the System Monitor chart shown in Figure 9.24 we have added the PhysicalDisk: Avg. Disk Bytes/Read counter and the Buffer Manager: Page lookups/sec counter. We have executed a query that retrieves a row from the Accounts table using a non-clustered index. We can see a blip in the Buffer Manager: Page lookups/sec counter. However, note the value of the PhysicalDisk: Avg. Disk Bytes/Read counter. It is 8,192 bytes. This shows us that a single page read was performed.



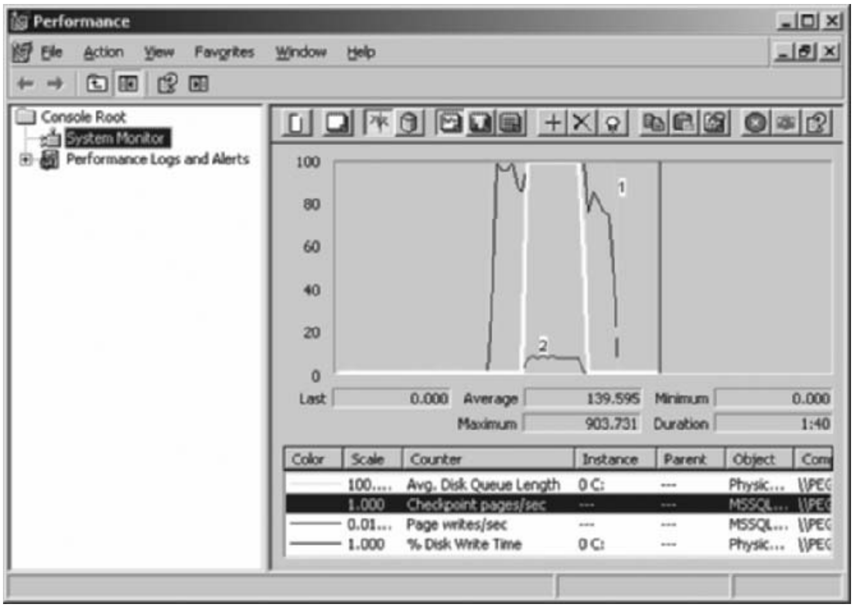
```
Last      0.000   Average  8192.000   Minimum  0.000
                  Maximum  8192.000   Duration  1:40
```

Color	Scale	Counter	Instance	Parent	Object	Computer
——	0.01...	Avg. Disk Bytes/R...	_Total	---	Physic...	\\PEGASU
——	0.10...	Page lookups/sec	---	---	MSSQL...	\\PEGASU

Figure 9.24 *Investigating disk activity – examining read size*

In the System Monitor chart shown in Figure 9.25, we have added the PhysicalDisk counters, Avg. Disk Queue Length and %Disk write time, and the Buffer Manager counters, Page writes/sec, and Checkpoint pages/sec.

We have initiated an update of a large table, resulting in many rows being changed. The Avg. Disk Queue Length counter is labeled (1). This peaks at 14 and averages 2.7. The counter that closely tracks it is %Disk write time, which is 100 percent at peak. Clearly, a lot of write activity is being performed. The data file and log file are on one disk, so what is responsible for the activity? The clue is our highlighted counter, Checkpoint pages/sec. This averages 140 pages/sec with a peak of 904 pages/sec. This results in a Page writes/sec, labeled (2), averaging 140 and peaking at 904. This is the checkpoint that is flushing to disk.


Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
100
80
60
40
20
0
1
2

Last	0.000	Average	139.595	Minimum	0.000
		Maximum	903.731	Duration	1:40

Color	Scale	Counter	Instance	Parent	Object	Com
	100....	Avg. Disk Queue Length	0 C:	---	Physic...	\\PE
	1.000	Checkpoint pages/sec	---	---	MSSQL...	\\PE
	0.01...	Page writes/sec	---	---	MSSQL...	\\PE
	1.000	% Disk Write Time	0 C:	---	Physic...	\\PE

Figure 9.25 *Investigating disk activity – examining a large update*

Finally, let us have a look at fn_virtualfilestats—a system table-valued function. This gives us very useful information about I/O statistics for individual data and log files. It is very easy to use.

```
SELECT * FROM :: fn_virtualfilestats(11, 1)
```

The first parameter is the database ID, and the second parameter is the file ID. Personally, I find the best way to obtain these values is with sp_helpdb and sp_helpfile. If you prefer, use the system functions DB_ID() and FILE_ID() to find these values. Example output is as follows:

```
DbId FileId TimeStamp  NumberReads NumberWrites  BytesRead  BytesWritten  IoStallMS

11   1      9293172    1579        572           398663680  73203712      11810
```

9.3.4 Solving problems with disk I/O

Having determined that there is indeed a disk I/O bottleneck and that there is a sustained queue of requests, the next step is to eliminate causes other than SQL Server, such as a memory bottleneck causing high levels of paging to disk.

If the disk bottleneck proves to be SQL Server, it could be a specific set of queries—in which case it is possible that these queries could be made more efficient by rewriting or by a change in index design. This often cures the problem. However, if the workload on the SQL Server as a whole is generating more disk I/O than the I/O subsystem can handle, it may be time to invest in a RAID approach.

There are a number of RAID topologies that can be used; the fastest implementation of RAID, however, is usually hardware based. We have already discussed RAID configurations in this chapter.

If RAID configurations are not available, using multiple data files and filegroups on multiple disk spindles may be another option.

Also, remember that Windows can defragment disk drives. It is possible that a database file is fragmented because of the way it was created. This may have happened if many automatic extensions took place and the disk was shared with other applications that create files.

Ensure that the hardware components can theoretically handle the load. Apart from the disk drives, the disk controllers and I/O bus have a finite bandwidth.

10

Transactions and Locking

I once visited a customer to sanity check the physical design for a new database. In the course of checking the design I happened to notice that there were some people in an adjoining room entering data into forms on their PCs. Every so often one of these people would raise their hands in the air for a few seconds. After a while my curiosity got the better of me, and I asked the person who had invited me to do the sanity check what was happening.

It transpired that the people in the next room were entering trades into a financial system, but the lock conflict caused by the action of entering two trades simultaneously was so bad that they found it easier to raise their hands just before they pressed Enter on the keyboard to signal to their colleagues not to do the same. Ironically, what they were doing was implementing a locking protocol, which single-threaded the insertion of a trade. This is an example of a multiuser system where two users are one user too many!

Unfortunately, there are many multiuser systems out there that suffer from locking problems. Whether you design a system with locking in mind tends, like most things in life, to depend on your previous experiences. While I was working for Digital Equipment Corporation I was involved in the design of many multiuser online transaction processing systems (OLTPs). I came to learn very quickly that if I did not constantly ask the question: "Is this transaction likely to be the cause of a locking bottleneck?," that I would run into trouble. If your background is single-user systems or read only databases, this question might not be the first one on your mind.

This chapter introduces the concepts of transactions and locking, perhaps two of the most important features provided by a modern database management system and, perhaps, two of the features whose correct implementation by a database designer is most critical to database performance. The default SQL Server locking protocol provided by SQL Server is sophisticated. However, for those developers who need it, the default locking protocol provided by SQL Server can easily be changed to behave in a number of different ways. These capabilities will be covered in this chapter.

10.1 Why a locking protocol?

Single-user access to a database does not require a locking protocol. Nor does single or multiuser access to a read only database. Database management systems in reality must support more than one user concurrently accessing information, and it is this multiuser access that requires the database management system to provide a protocol to ensure that the changes being made to the database data by one user are not corrupted by another. Locking is not a luxury in a multiuser environment—it is a necessity.

Locking protocols are not all or nothing. Some protocols are more stringent than others with different database management systems adopting their own unique approaches. Locking is the natural enemy of performance, and so a more stringent locking protocol is more likely to adversely affect performance than a less stringent one. However, a more stringent locking protocol is also likely to provide a more consistent view of the data.

To provide an idea as to why a locking protocol is necessary let us consider some multiuser scenarios.

10.1.1 Scenario 1

In this scenario Mike modifies a stock level by subtracting 1,000 from it, leaving 100 items. Katy reads the stock level and sees that there are only 100 items in stock. Immediately after Katy has read this value and acted upon it, Mike's transaction fails and is rolled back, returning the stock level to its original value of 1,100.

This scenario highlights a classic problem. Katy has been allowed to read changes made by Mike before Mike has committed the changes—in other words, before Mike has irrevocably changed the data by ending the transaction with a commit. Until the transaction ends, Mike can choose to roll back the transaction, change the value again, or commit the transaction. In our example, Mike's transaction actually fails before it completes, causing the database management system to roll back the change. Katy is said to have read uncommitted, or dirty data. This is shown in Figure 10.1.

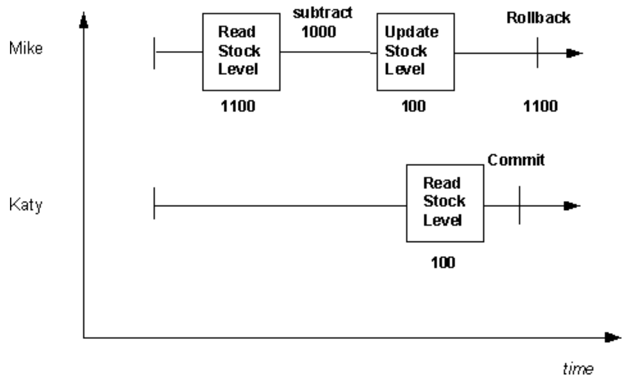


Figure 10.1 *Reading uncommitted changes*

10.1.2 Scenario 2

In this scenario Mike's transaction sums a list of debts in a table and checks the result against a total debt value held elsewhere in the database. While Mike's transaction is summing the values in the list, Katy's transaction inserts a new row into the debt table after Mike's transaction has passed by and updates the total debt value. When Mike finishes summing the list and compares the calculated sum with the total debt value, it reports a discrepancy, where, in fact, there is no discrepancy at all. This is called the phantom insert phenomenon. This is shown in Figure 10.2.

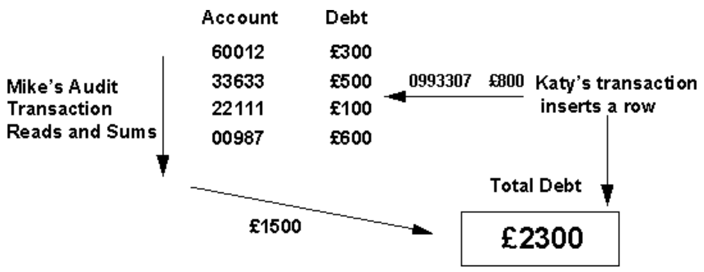


Figure 10.2 *The phantom insert phenomenon*

These are only two examples of a number of possibilities that can occur if locking protocols are not used or the locking protocol used is not stringent enough. We will revisit some of these scenarios later. We have said that SQL Server uses a locking protocol, so let us now investigate how this works.

10.2 The SQL Server locking protocol

The locking protocol adopted by SQL Server consists of placing different types of locks on different database objects. In SQL Server these objects include a table, a database page, a row, and an index entry. As we have seen, a database page is 8 KB in size, and any object resident within this 8 KB is locked implicitly when the database page is locked. Therefore, if a database page is locked, every row held on that page is effectively locked. Similarly, if a table is locked, every row in that table is locked.

We will now look in detail at the types of locks used, what objects can be locked, and the duration of these locks.

10.2.1 Shared and exclusive locks

To generalize, SQL Server applies a write lock when it writes information or a read lock when it reads information. Writing information usually refers to inserting, updating, or deleting rows, whereas reading information usually refers to retrieving rows with, for example, a SELECT statement. There are some simple rules that we can make at this point:

- If a user has placed a read lock on an object such as a row, another user can also place a read lock on that object. In other words, both users can read the same object simultaneously. In fact, any number of users can place a read lock on an object at the same time.
- If a user has placed a write lock on an object, another user cannot also place a write lock on that object. Also, another user cannot place a read lock on that object. In other words, once a user has placed a write lock on an object, other users cannot place read or write locks on the same object simultaneously.

Because many users can place read locks on the same table, page, or row concurrently these read locks are usually referred to as shared locks. Write locks, on the other hand, are normally referred to as exclusive locks. Table 10.1 shows the compatibility between shared and exclusive locks. As can be seen, only shared locks are compatible.

Once a lock has been placed on an object, it has a lifetime. Suppose a Transact-SQL statement that causes a row lock to be taken out is executed inside a user-defined transaction. In the default case, shared locks live for

Table 10.1 *Compatibility between shared and exclusive locks*

Mode of Currently Granted Lock	Mode of Requested Lock	
	Exclusive	Shared
Exclusive		
Shared		

the time it takes the SQL statement to read the row, whereas exclusive locks live for the length of the user-defined transaction. This is shown in Figure 10.3.

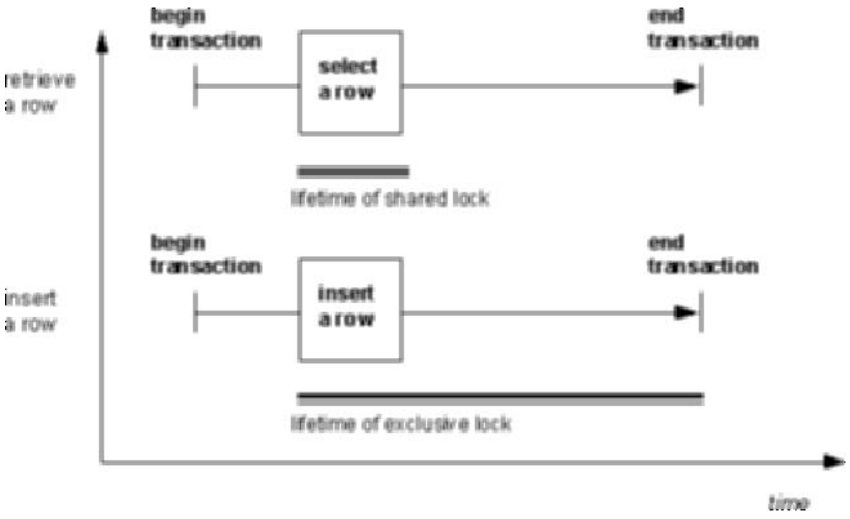


Figure 10.3 *The default lifetime of SQL server locks*

This behavior can be overridden with the use of the REPEATABLE READ keyword or transaction isolation levels, as we will see later in this chapter.

Note: Beware of the SET IMPLICIT_TRANSACTIONS ON statement. It will automatically start a transaction when Transact-SQL statements such as SELECT, INSERT, UPDATE, and DELETE are used. The transaction will not be committed and its locks will not be released until an explicit COMMIT TRANSACTION statement is executed. To see if it is set, use DBCC USEROPTIONS (described later).

SQL Server also uses locks other than shared or exclusive. For example, it uses update locks as an optimization to avoid deadlocks. We will look at update locks when we investigate deadlocks later in the chapter.

10.2.2 Row-, page-, and table-level locking

Is row-level locking better than page-level locking? It depends. Applications require different levels of locking granularity. One application may benefit from page-level locking while another application may benefit from row-level locking. Why is this? To investigate it is useful to consider the different granularity of lock that could be taken out by some theoretical database management system.

Figure 10.4 shows the database concurrency for different lock granularity. By lock granularity we mean the object that is locked from, on one side of the spectrum, an individual column in a row to the other side of the spectrum, a whole database. As can be observed from Figure 10.4, locking individual columns provides the highest level of concurrency. By this we mean that multiple users could be updating different columns in the same row simultaneously. They would not be involved in lock conflict.

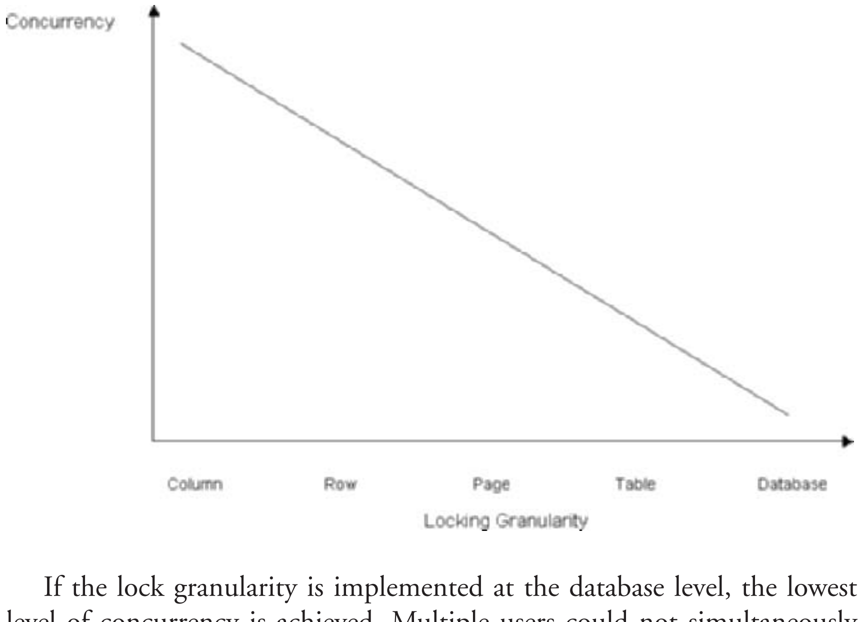


Figure 10.4
Concurrency versus locking granularity

If the lock granularity is implemented at the database level, the lowest level of concurrency is achieved. Multiple users could not simultaneously change anything at all in the database. If they tried, they would be involved in lock conflict.

So, if locking individual columns provides the highest level of concurrency, why do SQL Server and databases in general not lock at the column level? To explain this we need to add some more information to our graph. In Figure 10.5, we have added system resource use to our graph. It can be seen that an increase in system resource use parallels an increase in lock granularity. The finer the granularity, the more system resources used.

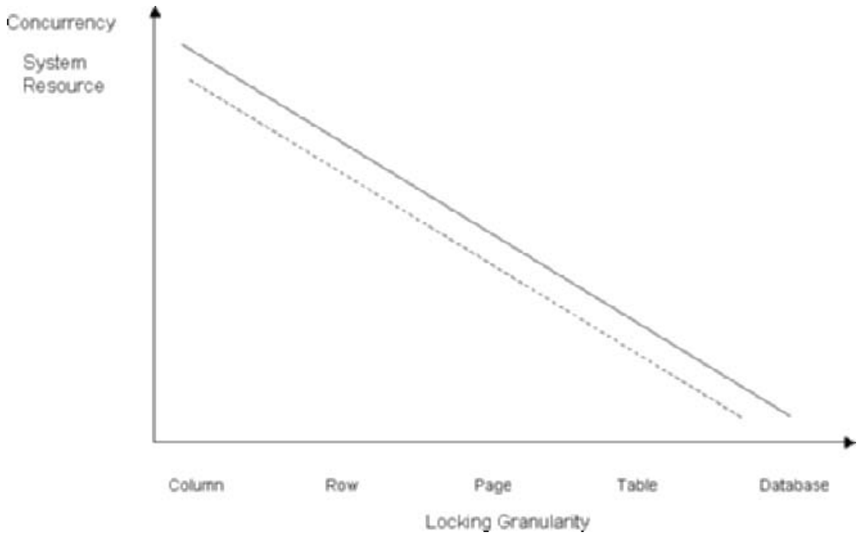


Figure 10.5 *System resource versus locking granularity*

This is why SQL Server and databases in general do not lock at the column level. The system resource use in terms of the number of locks required and their management would be too great. Locks are approximately 100 bytes each in SQL Server. Using 100 bytes of memory to lock a ten-byte column seems a little over the top. To lock at the column level would probably use tens of thousands of locks in a medium-sized database, which could equate to many megabytes of memory. The CPU resource needed to manage these locks would be massive.

Consequently, SQL Server locks rows, pages, and tables, which, depending on the application, is a reasonable approach. The database itself can, of course, be set to single-user mode, which effectively provides locking at the database level.

10.2.2.1 When are row-level locks used?

Locking at the row level can be considered to be the default situation. Usually, unless you have changed the default behavior, SQL Server will take shared and exclusive locks out on rows. When we refer to rows, we are referring to data rows in the data pages of a table. However, within an index, index pages contain index entries. These can also be locked with a lock equivalent to a row lock, known as a key lock.

Conventionally, the data pages in a table on which there is a clustered index present are considered to be the leaf level of the clustered index—that is, part of the clustered index. For this reason, the row locks on the data rows in a table with a clustered index are managed as key locks. Figure 10.6 shows individual rows being locked within the pages of a table.

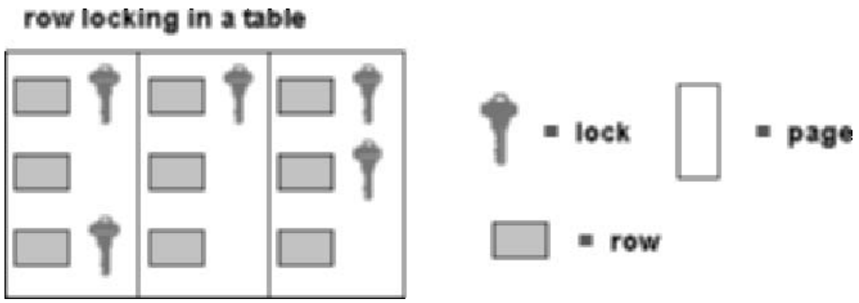


Figure 10.6 *Row-level locking*

Figure 10.7 shows page locks being used to lock the individual pages within a table. In this case one lock will effectively lock all the rows in the page.

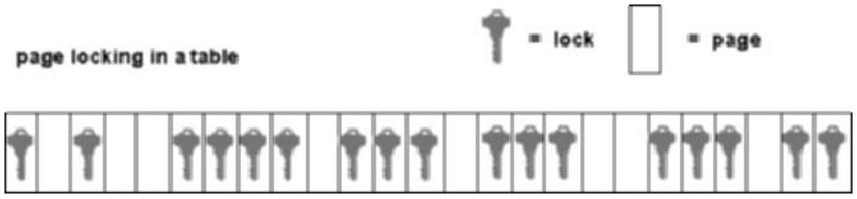


Figure 10.7 *Page-level locking*

10.2.2.2 When are table-level locks used?

One of the reasons that SQL Server tends to lock at the row level is that it has the capability to escalate locks but not to de-escalate locks. Therefore, if SQL Server decides that a SQL statement is likely to lock the majority of rows in a table, it may lock at the table level. The same logic is used if SQL Server determines that most of the rows in a page are likely to be locked—it may take out a page lock instead of multiple row locks.

The advantage to holding a single table lock is due to system resources. Managing a single table lock is less resource intensive than managing multiple row locks, and saving locks will save memory. However, locking at the table level may reduce concurrency. For example, an exclusive lock held at the table level will block all other users from accessing rows within that table, whether they wish to acquire shared or exclusive locks. Figure 10.8 shows table-level locking.

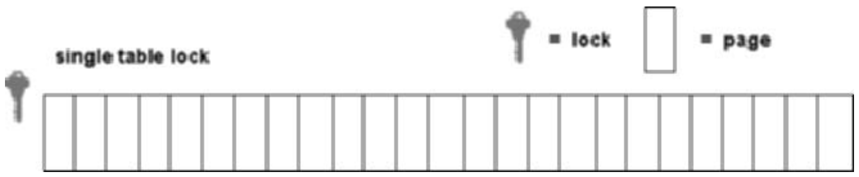


Figure 10.8
Table-level locking

SQL Server controls when escalation occurs. The database administrator has no control over this, since there is no relevant server configuration option.

Note: If a table scan is being used to read data, row locks will be taken out and released in a sequential fashion. If we choose to use certain lock hints—for example, REPEATABLEREAD, discussed later—we are requesting not to release the row lock when we have finished with the row. In this circumstance, when performing a table scan, SQL Server may well take out a table lock if the number of row locks exceeds an internal threshold.

So, we have introduced shared and exclusive locks, as well as page-, table-, and row-level locking. We need to introduce more types of locks before we can give examples of the SQL Server locking protocol in action; but first let us look at lock timeouts and then a phenomenon known as a deadlock or deadly embrace.

10.2.3 Lock timeouts

If a user's lock is blocked by another lock, the user must wait until the blocking lock is released before he or she can acquire the lock. If the blocking lock is not released for a long time, the user will have to wait for a long time. An application design flaw may mean that the blocking lock is not released at all, and then the database administrator must intervene.

It is possible in SQL Server to set a lock timeout value for a connection so that it will only wait to be granted its lock for a predefined period of time, after which it will receive an error message informing it that the timeout period has been exceeded. This approach assumes that if a lock is kept waiting for a period of time there must be a problem, and it is better that the connection gives up and releases its locks rather than wait indefinitely, perhaps blocking other users. The connection can always try again, or log the problem and gracefully inform the user that it cannot continue.

What constitutes a realistic timeout value? Too long and the user will become impatient, too short and the connection will give up when it would have acquired the lock had it waited a little longer. Personally, I think around ten seconds is not unreasonable.

A lock timeout value is set per connection as follows:

```
SET LOCK_TIMEOUT 10000
```

The timeout value is specified in milliseconds. A value of –1 means wait indefinitely (the default), whereas a value of 0 means do not wait at all. I do not recommend using this value. You could timeout as soon as you attempt to execute a statement, whereas if you had waited a fraction of a second you would have acquired the lock.

If a timeout occurs, an error, 1222, is returned and the connection is rolled back.

To test the value of lock timeout set for a connection the function @@LOCK_TIMEOUT can be used.

10.2.4 Deadlocks

A deadlock situation can occur in SQL Server when a user holds a lock on a resource needed by a fellow user who holds a lock on a resource needed by the first user. This is a deadly embrace, and the users would wait forever if SQL Server did not intervene (see Figure 10.9.)

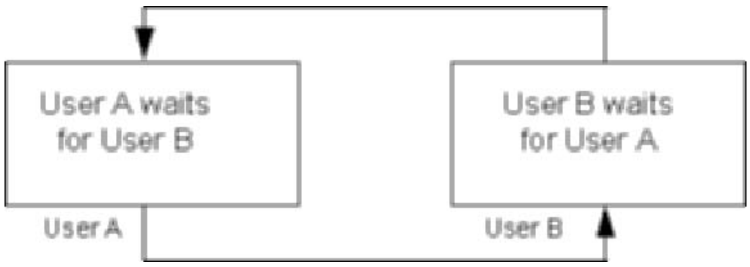


Figure 10.9 *A deadlock between two users*

SQL Server chooses one of the deadlocked users as a victim and issues a rollback for its transaction. It will receive an error message similar to the following:

```
Server: Msg 1205, Level 13, State 1, Line 1

Your transaction (Process ID 52) was deadlocked on {lock}
resources with another process and has been chosen as the
deadlock victim. Rerun your transaction.
```

In the application code, this error should be trapped and dealt with cleanly. The application might retry a number of times before giving up and informing the user that there is a problem.

A connection can set its deadlock priority such that, in the event of it being involved in a deadlock, it will be chosen as the victim, as follows:

```
SET DEADLOCK_PRIORITY LOW
```

To return to the default deadlock handling mechanism, use the following code:

```
SET DEADLOCK_PRIORITY NORMAL
```

Generally, the transaction involved in the deadlock that has accumulated the least amount of CPU time is usually chosen as the victim.

10.2.5 Update locks

As well as placing shared and exclusive locks on database rows, SQL Server also makes use of a type of lock known as an update lock. These locks are associated with SQL statements that perform update and delete operations, which need to initially read rows before changing or deleting them. These rows have update locks placed on them that are compatible with shared read locks but are not compatible with other update locks or exclusive locks. If the rows must subsequently be updated or deleted, SQL Server attempts to promote the update locks to exclusive locks. If any other shared locks are associated with the rows, SQL Server will not be able to promote the update locks until these are released. In reality the update lock is not promoted, but a second lock is taken out, which is, in fact, an exclusive lock.

Why bother with update locks? Update locks are really an optimization to minimize the possibility of deadlocks. Consider two users, Mike and Katy, who are about to update the same row. Without update locks, each user will take out a shared lock on the row. Shared locks are compatible, so both users will acquire the lock successfully. Mike's UPDATE statement, finding that the row meets the criteria in its WHERE clause, attempts to take out an exclusive lock on it. Mike's UPDATE statement will now have to wait, since it is blocked by Katy's shared lock.

Katy's UPDATE statement, finding that the row meets the criteria in its WHERE clause, attempts to take out an exclusive lock on the row. Katy's UPDATE statement cannot take out the exclusive lock, since it is blocked by Mike's shared lock. Her update statement would also be forced to wait, except that this is clearly a deadlock. SQL Server will choose a victim and its transaction will be rolled back. This is shown in Figure 10.10.

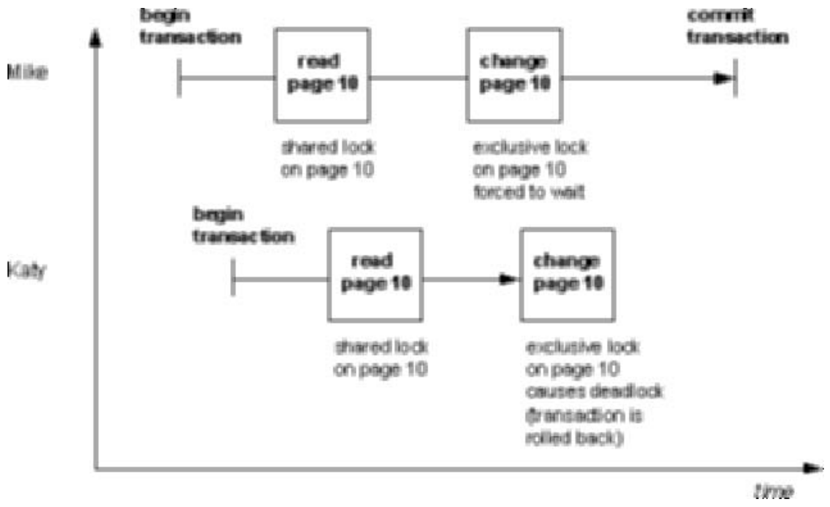


Figure 10.10 *A deadlock caused by two users updating the same page*

Now let us take the same example, but this time we will make use of update locks. This is exactly what SQL Server does.

When Mike issues his UPDATE statement, he now takes out an update lock on the row instead of a shared lock. Katy's UPDATE statement also attempts to take out an update lock on the row, but update locks are not compatible so she will be forced to wait. Mike's UPDATE statement, finding that the row meets the criteria in its WHERE clause, attempts to take out an exclusive lock on the row. Since Katy does not have any locks on the row, Mike's UPDATE statement successfully acquires the exclusive lock and completes. Mike now commits his transaction and releases his locks. Katy's

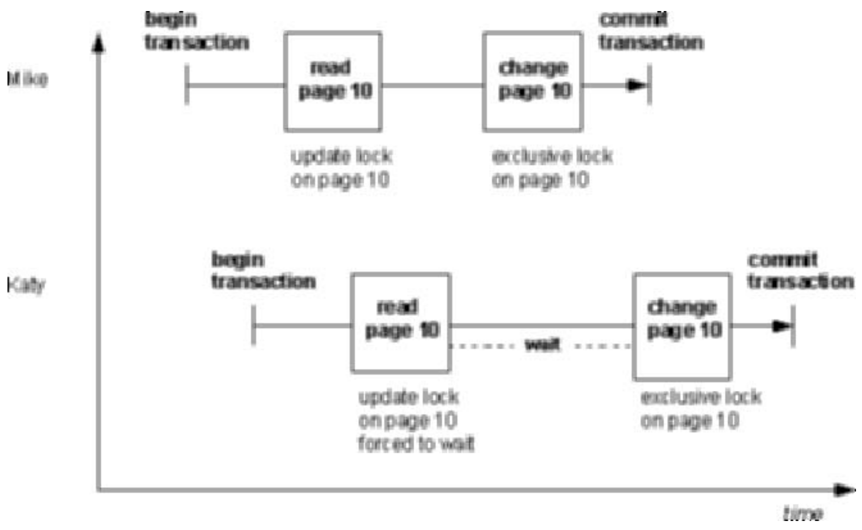


Figure 10.11 *A deadlock avoided by using update locks*

UPDATE statement, which has been waiting, can now proceed. This is shown in Figure 10.11.

Clearly, this is a cleaner mechanism. No transactions are deadlock victims, which means no transactions are cancelled and rolled back. Transactions that are rolled back have their work effectively thrown away. Using update locks, Katy's UPDATE statement merely suffers a short delay.

10.2.6 Intent locks

As well as placing shared and exclusive locks on database tables, SQL Server also makes use of a type of lock known as an intent lock. Intent locks are placed on the table and pages in the table when a user locks rows in the table, and they stay in place for the life of the row locks. These locks are used primarily to ensure that a user cannot take out locks on a table or pages in the table that would conflict with another user's row locks. For example, if a user was holding an exclusive row lock and another user wished to take out an exclusive table lock on the table containing the row, the intent lock held on the table by the first user would ensure that its row lock would not be overlooked by the lock manager.

10.2.7 Modifying the default locking behavior

There are two ways in which SQL Server's default locking behavior can be modified. Individual SQL statements can be qualified with a keyword known as a lock hint to modify the locking behavior for that particular statement, or a default locking behavior for the connection can be set with the SET TRANSACTION ISOLATION LEVEL statement.

10.2.7.1 Transaction isolation levels

SQL Server allows the transaction isolation level to be set for a connection. This sets a default locking behavior.

Levels of transaction isolation are specified by the ANSI standard, with each one defining the type of phenomenon not permitted while concurrent transactions are running. The higher the isolation level, the more stringent the locking protocol—with the higher levels being a superset of the lower levels. The transaction isolation levels are as follows:

- Read uncommitted
- Read committed
- Repeatable read
- Serializable

The locking behavior that corresponds with read uncommitted provides the least integrity but potentially the best performance. The read committed isolation level provides more integrity than read uncommitted, and the repeatable read isolation level provides even more integrity. The greatest integrity is provided by the serializable isolation level. We have already met dirty reads and the phantom phenomena. Table 10.2 shows whether the dirty read and the phantom phenomena are allowed by the various isolation levels.

Table 10.2 *Isolation levels and allowed locking phenomena*

Isolation Level	**Dirty Reads**	**Nonrepeatable Reads Allowed**	**Phantoms Allowed**
Serializable	No	No	No
Repeatable Read	No	No	Yes
Read Committed	No	Yes	Yes
Read Uncommitted	Yes	Yes	Yes

It can be seen that only the serializable isolation level prevents all these phenomena from occurring.

By default, SQL Server runs at transaction isolation level read committed.

The transaction isolation level is set for the connection with the following syntax:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

The DBCC utility with the USEROPTIONS parameter can be used to check the current isolation level of the connection, as follows:

```
DBCC USEROPTIONS

Set Option                    Value
textsize                      2147483647
language                      us_english
dateformat                    mdy
datefirst                     7
quoted_identifier             SET
arithabort                    SET
ansi_null_dflt_on             SET
ansi_defaults                 SET
ansi_warnings                 SET
ansi_padding                  SET
ansi_nulls                    SET
concat_null_yields_null       SET
isolation level               repeatable read
```

We will study how transaction isolation levels modify locking behavior between users later in this chapter.

10.2.7.2 Lock hints

The keywords available as lock hints for modifying locking behavior are as follows:

- DBLOCK

- HOLDLOCK
- NOLOCK
- PAGLOCK
- READCOMMITTED
- READPAST
- READUNCOMMITTED
- REPEATABLEREAD
- ROWLOCK
- SERIALIZABLE
- TABLOCK
- UPDLOCK
- XLOCK

Note: DBLOCK is new to SQL Server 2005. TABLOCKX is no longer available in SQL Server 2005.

Some hints are supported for backward compatibility such as:

- HOLDLOCK
- NOLOCK

The recommended hints to use instead are as follows:

- SERIALIZABLE
- READUNCOMMITTED

Some hints enable the developer to specify the lock granularity, such as:

- DBLOCK
- PAGLOCK
- ROWLOCK

- TABLOCK

Other hints enable the developer to specify the transaction isolation level behavior at the statement level, such as:

- READUNCOMMITTED
- READCOMMITTED
- REPEATABLEREAD
- SERIALIZABLE

Lock hints are used, for example, on a SELECT statement, as follows:

```
SELECT * FROM branches WITH (SERIALIZABLE)

SELECT balance FROM accounts WITH (READUNCOMMITTED)
   WHERE account_no = 1000
```

The effect of these lock hints can be described as follows:

- **DBLOCK:** The DBLOCK hint forces a shared database lock to be taken when enough information is read by a single SELECT statement.
- **HOLDLOCK:** The HOLDLOCK hint forces a shared lock on a table to remain until the transaction completes. Key range locking will also be used to prevent phantom inserts. Nonrepeatable reads are also prevented. This is equivalent to the SERIALIZABLE hint. Data consistency will be provided to the level experienced by transactions running at transaction isolation level SERIALIZABLE. Using the HOLDLOCK keyword may, and usually will, degrade performance, since lock contention may increase.
- **NOLOCK:** The NOLOCK hint allows a dirty read to take place—that is, a transaction can read the uncommitted changes made by another transaction. The exclusive locks of other transactions are not honored, and the statement using this hint will not take out shared locks. This is equivalent to the READUNCOMMITTED hint. Data consistency will be provided to the level experienced by trans-

actions running at transaction isolation level READ UNCOMMITTED. Using the NOLOCK keyword may increase performance, since lock contention may decrease, but this will be at the risk of lower consistency.

- **PAGLOCK:** The PAGLOCK hint forces shared page locks to be taken where otherwise SQL Server may have used a table or row lock. For example, consider the following statement:

  ```
  SELECT balance FROM accounts WITH (REPEATABLEREAD,
  PAGLOCK)
  ```

  If there is no appropriate index, the query optimizer will choose a table scan as the strategy used to execute the query. Depending on the number of rows that may be locked, the lock manager will take out row locks or perhaps a table lock because the REPEATABLE READ lock hint will force the shared row locks to be held until the end of the transaction, and therefore a single table lock is far more efficient. The PAGLOCK hint will ensure that the lock manager will use page locking instead of table locking or row locking. This hint does not only apply to shared locks. Exclusive page locks will also be forced if, say, an UPDATE statement rather than a SELECT statement was using the hint.

- **READCOMMITTED:** The READCOMMITTED hint ensures that the statement behaves in the same way as if the connection were set to transaction isolation level READ COMMITTED. This is the default behavior for SQL Server. Shared locks will be used when data is read, which prevents dirty reads, but the shared locks are released at the end of the read and are not kept until the end of the transaction. This means that non-repeatable reads or phantom inserts are not prevented.

- **READPAST:** This lock hint enables a statement to skip rows that are locked by other statements. The READPAST lock hint applies only to transactions operating at READ COMMITTED isolation level and will read only past row-level locks. This is only valid on a SELECT statement. This is useful when, for example, multiple transactions are reading items from a queue implemented as a table and a transaction wants to skip a locked queue item and read another item to process.

- **READUNCOMMITTED:** This lock hint is equivalent to the NOLOCK lock hint.

- **REPEATABLEREAD:** The REPEATABLEREAD hint ensures that the statement behaves in the same way as if the connection were set to transaction isolation level REPEATABLE READ. This is not the default behavior for SQL Server. Shared locks will be used when data is read, and these will not be released until the end of the transaction. This means that non-repeatable reads are prevented. However, phantom inserts are not prevented. This lock hint may reduce concurrency, since shared locks are held for longer periods of time than if the default read committed behavior is used.
- **ROWLOCK:** This hint forces the use of rowlocks and is similar in use to PAGLOCK.
- **SERIALIZABLE:** The SERIALIZABLE hint forces shared locks to stay until the transaction completes. This is equivalent to specifying the HOLDLOCK hint. Key range locking will be used to prevent phantom inserts if indexes are present. Non-repeatable reads are also prevented. Data consistency will be provided to the level experienced by transactions running at transaction isolation level SERIALIZABLE. Using the SERIALIZABLE keyword may, and usually will, degrade performance, since lock contention may increase.
- **TABLOCK:** The TABLOCK hint forces a shared table lock to be taken where otherwise SQL Server may have used row locks. It will not be held until the end of the transaction unless hints such as REPEATABLEREAD are also used.
- **UPDLOCK:** The UPDLOCK hint forces SQL Server to take update locks where otherwise SQL Server would have used shared locks. The update locks are held until the end of the transaction. Update locks are compatible with shared locks but not exclusive locks or other update locks.
- **XLOCK:** This hint forces exclusive locks to be taken out. It is typically used with TABLOCK and PAGLOCK.

10.2.8 Locking in system tables

Transact-SQL statements such as CREATE TABLE manipulate system tables. For example, when a table is created, rows are inserted into the sysobjects, sysindexes, and syscolumns system tables. Data definition language (DDL) statements can appear in explicit transactions, and, therefore, any locks taken out as a result of actions to the system tables can be held for a period of time—blocking other users if the developer is not careful. Here are some examples of DDL statements that can appear in an explicit transaction:

```
ALTER TABLE            DROP PROCEDURE
CREATE DEFAULT         DROP RULE
CREATE INDEX           DROP TABLE
CREATE PROCEDURE       DROP TRIGGER
CREATE RULE            DROP VIEW
CREATE TABLE           GRANT
CREATE TRIGGER         REVOKE
CREATE VIEW            SELECT INTO
DROP DEFAULT           TRUNCATE TABLE
DROP INDEX
```

As an example of this behavior, suppose a table is created in an explicit transaction. SQL Server takes out exclusive locks in the sysobjects, sysindexes, and syscolumns system tables. These locks are key locks, since each of these system tables has a clustered index present. If the transaction does not complete, a query issued in another connection against these system tables will be blocked. For example, a CREATE TABLE statement issued within an explicit transaction will block an sp_help issued on another connection. It is important, therefore, that these transactions are committed quickly.

Note that Sch-M (schema modification) locks are taken when a table data definition language (DDL) operation is being executed. This is incompatible with all other lock types.

10.2.9 Monitoring locks

Finally, we need to introduce the means by which we can observe SQL Server lock management in action, and then we can look at some examples of the SQL Server locking protocol. There are a number of ways to find information about the locking that is happening within SQL Server. These include the following:

- Use the sp_lock system stored procedure.
- Use the SQL Enterprise Manager.

- Use the Performance Monitor.
- Interrogate the system table syslockinfo directly.
- Use the SQL Profiler.

Additionally, the sp_who system stored procedure is useful in finding blocked and blocking processes, and the DBCC utility can be used to set trace flags to record lock and deadlock information.

10.2.9.1 Using the sp_lock system stored procedure

The sp_lock system stored procedure displays information about the locks held by processes using the server. It can be entered as a stand-alone statement, in which case it will display all locks managed by the server, or it can take up to two SQL Server process identifiers (SPIDs) as a parameter. Some example output from the sp_lock system stored procedure is as follows:

```
EXEC sp_lock

spid    dbid    ObjId        IndId    Type    Resource           Mode     Status
                                              --------           ----     -----
51      7       0            0        DB                         S        GRANT
51      7       965578478    2        PAG     1:113              IS       GRANT
51      7       965578478    2        KEY     (4501518d90d1)     S        GRANT
51      7       965578478    0        RID     1:348:14           S        GRANT
51      7       965578478    0        PAG     1:348              IS       GRANT
51      7       965578478    0        TAB                        IS       GRANT
52      7       965578478    0        TAB                        IX       GRANT
52      7       965578478    0        PAG     1:348              IX       GRANT
52      7       965578478    0        RID     1:348:14           X        CNVT
52      7       965578478    0        RID     1:348:14           U        GRANT
52      7       965578478    2        KEY     (4501518d90d1)     U        GRANT
52      7       965578478    2        PAG     1:113              IU       GRANT
52      7       0            0        DB                         S        GRANT
```

Here's a hint—to translate the ObjId to a table name, use the built-in system function OBJECT_NAME. For example:

```
SELECT OBJECT_NAME (965578478)
--------
accounts
```

The above output from sp_lock shows a number of locks held on various objects. Let us discuss the meaning of the columns in the output before we investigate the rows.

The first column contains the SPID value. A client connection to SQL Server is allocated an SPID value, and each row in the output represents a lock requested by the SPID that has not been released at the time sp_lock was executed. A typical server will be managing many locks at a given instance in time, so it is often more practical to limit the output to a particular SPID or pair of SPIDs by supplying these values as parameters.

The next five columns, dbid, ObjId, IndId, Type, and Resource, help to define the resource that is locked. We know already that SQL Server locks objects such as rows and tables, and these columns let us know what type of resource is locked as well as which instance of this resource type. The dbid column contains the database ID, the ObjId column contains the object ID, and the IndId contains the index ID. This column can contain the values 0, to represent the table itself; 1, the clustered index, if one is present; > 1 for a non-clustered index; and 255 for TEXT/IMAGE data. The Type column tells us the type of resource locked, such as a row or page, and, finally, the Resource column provides information to completely identify the resource instance. Whether these columns contain data depends on the type of resource being locked. For example, in the case of a database, the Resource column is empty.

The Mode column tells us whether we have a shared lock or exclusive lock or one of a myriad of other modes of lock on our resource. Finally, the Status column shows us whether the lock has been granted (GRANT), is waiting to be granted (WAIT), or is waiting to be converted to another mode (CNVT). When investigating lock problems, I often hunt first for locks that have not been granted. They normally relate to the blocked user and represent a small number of locks on the system. Let us now look at the connections in our example.

All the connections—that is, SPIDs—have been granted a shared lock on the database with ID value 7:

```
51 7          0    0 DB   S GRANT
52 7          0    0 DB   S GRANT
```

An easy way to translate the dbid to a database name is to execute the system stored procedure sp_helpdb, which returns this information in its display. Alternatively, use the function DB_NAME(). The reason the connections have been granted a shared lock is that any connection that has selected a database with a USE statement explicitly or implicitly via the drop-down list in the query analyzer is granted such a lock. This is used to manage such operations as a connection attempting to set the database to single-user mode.

Let us investigate the locks held by SPID 51. Apart from the database lock, it has requested and been granted shared (S) locks on two resources: a KEY and a RID.

```
51 7 965578478  2 KEY  (4501518d90d1) S  GRANT
51 7 965578478  0 RID  1:348:14 S GRANT
```

A RID is a row lock on a data row on a data page. A KEY lock is a row lock on an index entry (key plus pointer) on an index page.

Note: Conventionally, the data pages in a table with a clustered index are considered to be part of the clustered index. For that reason a row lock on a data row on a data page in such a table is considered to be a KEY lock, not a RID lock.

If we take the row lock first, we can see that the resource information shows us that we have a dbid value of 7, which represents the database BankingDB, and an ObjId value of 965578478, which, when translated with the OBJECT_NAME function, represents the table, Accounts, in this database. The IndId column contains a value of 0, which represents the table rather than an index on the table. The Resource column value is 1:348:14, which specifies that the resource in the table is identified as file ID 1, page 348, slot 14. This uniquely identifies a row on the page. The file ID must be present, since page numbers are only unique with a database file.

Hint: To convert a file ID to a filename, use the FILE_NAME() function.

If we look at the KEY lock, we can see the same values in the dbid and ObjId columns, but there is a value of 2 in the IndId column.

The following Transact-SQL will translate this index ID to an index name:

```
SELECT name FROM SYSINDEXES
 WHERE id = OBJECT_ID('Accounts') AND indid = 2
```

Of course, since we already know the object ID value, we could have just used this instead of translating the object name.

So we now know the index in which our KEY lock is held. The Resource column value is (4501518d90d1). This is of little use to us, since it is a hexadecimal number, which is the result of some hash function used internally, presumably used on the key value and other inputs. The other locks held by SPID 51 are intent locks:

```
51      7       965578478    2   PAG     1:113    IS    GRANT
51      7       965578478    0   PAG     1:348    IS    GRANT
51      7       965578478    0   TAB              IS    GRANT
```

We discussed intent locks earlier in the chapter. We stated that intent locks are placed on the table and pages in the table when a user locks rows in the table, and they stay in place for the life of the row locks. We can see that a shared intent (IS) lock has been taken out on page 1:348 and page 1:113. This is expected behavior, because we have a row lock held in data page 1:348. Page 1:113 will be the index page containing the locked index entry. Both of these pages are subordinate to the table, and so we see an intent lock on the table. These intent locks will prevent, for example, another connection from taking out an exclusive (X) lock on the table while our connection has shared (S) locks on rows in the table.

Those were the locks held by SPID 51. Let us now investigate the locks held by SPID 52. They are repeated here for clarity:

```
52 7 965578478 0 TAB IX GRANT
52 7 965578478 0 PAG 1:348 IX GRANT
52 7 965578478 0 RID 1:348:14 X CNVT
52 7 965578478 0 RID 1:348:14 U GRANT
52 7 965578478 2 KEY (4501518d90d1) U  GRANT
52 7 965578478 2 PAG 1:113 IU GRANT
52 7 0 0 DB S GRANT
```

We can see that SPID 52 has been granted two update (U) locks. These are compatible with shared (S) locks, as we described earlier in the chapter, and are used in UPDATE and DELETE statements during the search phase, when target rows are being identified. In fact, SPID 52 has issued an

UPDATE statement, which is attempting to change a row on which SPID 51 has shared (S) locks. Both update (U) locks have been granted, and the columns in the display contain values that are the same as the shared (S) locks on the KEY and RID for SPID 51. However, we can see that SPID 52 also has a lock that has not been granted:

```
52 7 965578478 0 RID 1:348:14 X CNVT
```

The lock manager has attempted to convert an update (U) lock to an exclusive (X) lock in order to change the row. It cannot do this, since SPID 51 has a shared (S) lock on this row, and we know that these locks are incompatible. For this reason the lock is now waiting to be converted, at which point it will have a status of GRANT. If the blocked lock were a new lock that the connection had tried to acquire, rather than the conversion of an existing lock, we would have seen a status of WAIT.

The intent locks behave in a fashion similar to those for SPID 51:

```
52    7     965578478    0     TAB               IX     GRANT
52    7     965578478    0     PAG     1:348     IX     GRANT
52    7     965578478    2     PAG     1:113     IU     GRANT
```

Exclusive intent (IX) locks have been granted on the data page and table, since these are compatible with the shared intent (IS) locks of SPID 51. An update intent (IU) lock has also been granted on the index page, since an update lock (U) has been granted on the index entry. The lock manager is not going to take out an exclusive (X) lock on the index entry, since the index column was not being updated.

10.2.9.2 Using the SQL Server 2005 Management Studio

The SQL Server 2005 Management Studio is as much about monitoring of performance as it is about solving performance problems. However, this chapter is all about locks. So, some basic locking information is included here. Chapter 12 will expand on the Management Studio in general.

What was called the Activity Folder (Enterprise Manager), in SQL Server 2000, is now called the Activity Monitor in SQL Server 2005. Figure 10.12 shows the Activity Monitor in the Management folder of the Object Explorer pane of the Management Studio.

As shown in Figure 10.12, the Activity Monitor allows you to examine database connections. The three available options cover processes, locks by

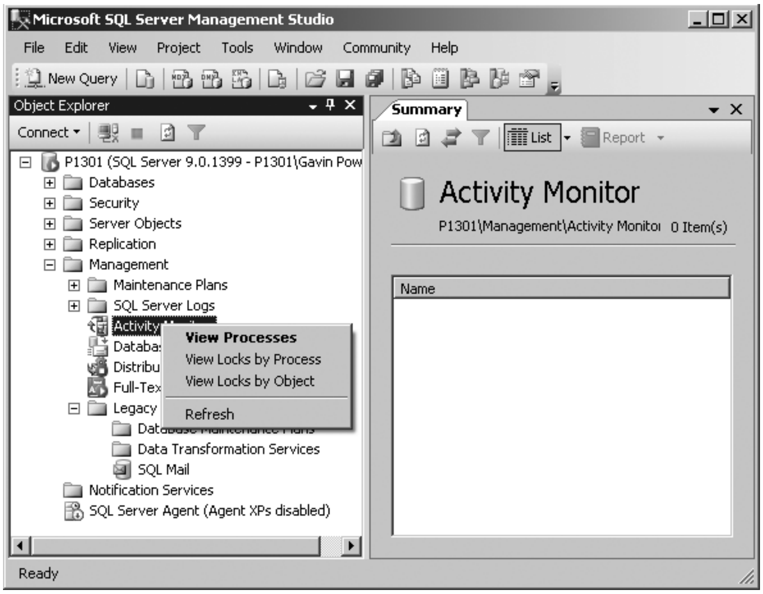
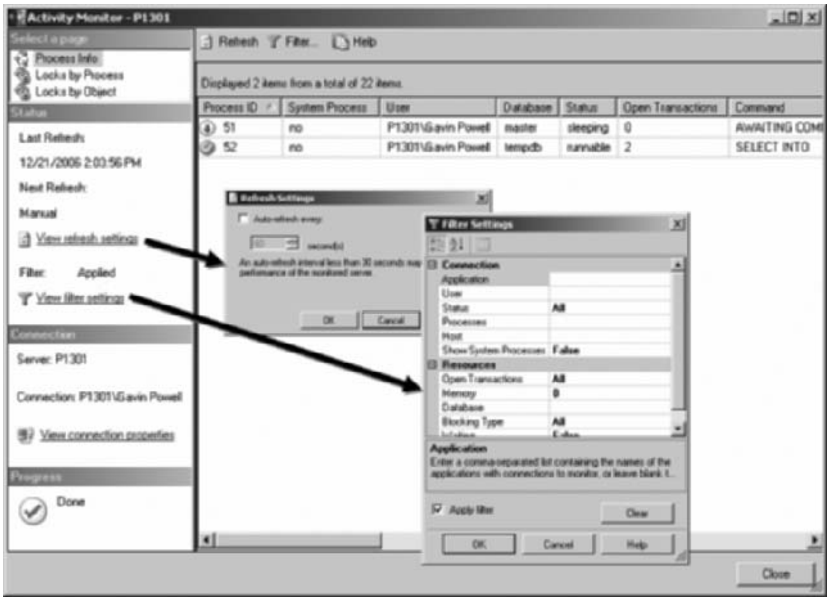


Figure 10.12
The Activity Monitor

process, and locks by object. If we expand Process Info and hide the console tree, we find the display shown in Figure 10.13.



Last Refresh: 12/21/2006 2:03:56 PM
Next Refresh: Manual
Filter: Applied
Server: P1301
Connection: P1301\Gavin Powell

Displayed 2 items from a total of 22 items.

Process ID	System Process	User	Database	Status	Open Transactions	Command
51	no	P1301\Gavin Powell	master	sleeping	0	AWAITING COM
52	no	P1301\Gavin Powell	tempdb	runnable	2	SELECT INTO

An auto-refresh interval less than 30 seconds may affect the performance of the monitored server.

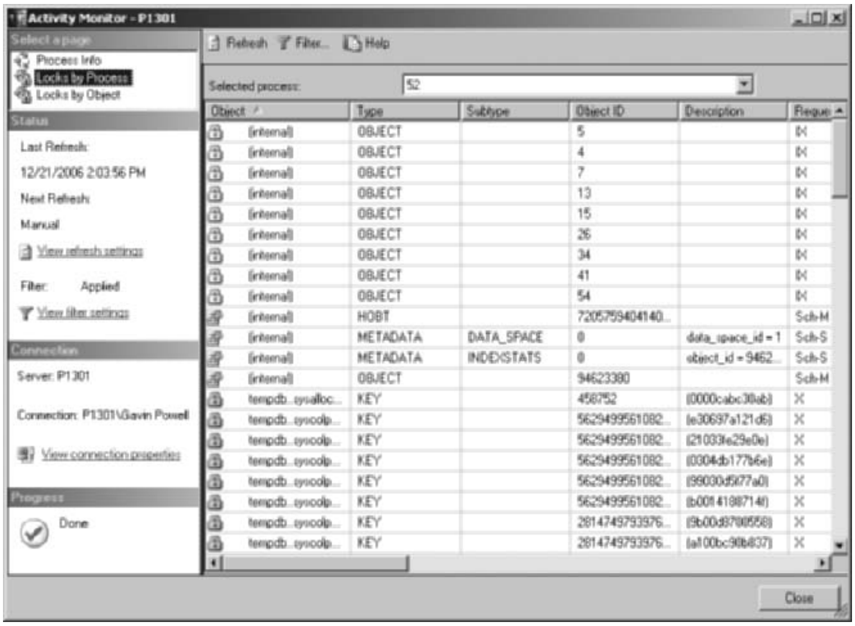
Connection	
Application	
User	
Status	All
Processes	
Host	
Show System Processes	False
Resources	
Open Transactions	All
Memory	0
Database	
Blocking Type	All

Application
Enter a comma-separated list containing the names of the applications with connections to monitor, or leave blank t...

Figure 10.13
Activity Monitor processes view

As shown in Figure 10.13, you can filter rows returned and set the refresh rate automatically. You can also move columns and sort by a particular column.

In Figure 10.14, you can isolate locks based on which process causes a specific lock to occur.



Selected process: 52

Object	Type	Subtype	Object ID	Description	Reque
(internal)	OBJECT		5		IX
(internal)	OBJECT		4		IX
(internal)	OBJECT		7		IX
(internal)	OBJECT		13		IX
(internal)	OBJECT		15		IX
(internal)	OBJECT		26		IX
(internal)	OBJECT		34		IX
(internal)	OBJECT		41		IX
(internal)	OBJECT		54		IX
(internal)	HOBT		7205759404140...		Sch-M
(internal)	METADATA	DATA_SPACE	0	data_space_id = 1	Sch-S
(internal)	METADATA	INDEXSTATS	0	object_id = 9462...	Sch-S
(internal)	OBJECT		94623380		Sch-M
tempdb..sysalloc...	KEY		458752	(0000cabc30ab)	X
tempdb..sysodp...	KEY		5629499561082...	(e30697a121d6)	X
tempdb..sysodp...	KEY		5629499561082...	(21033fe29e0e)	X
tempdb..sysodp...	KEY		5629499561082...	(0304db177b6e)	X
tempdb..sysodp...	KEY		5629499561082...	(99030d5f77a0)	X
tempdb..sysodp...	KEY		5629499561082...	(b0014188714f)	X
tempdb..sysodp...	KEY		2814749793976...	(9b00d8700558)	X
tempdb..sysodp...	KEY		2814749793976...	(a100bc98b837)	X

Figure 10.14
Locks by Process

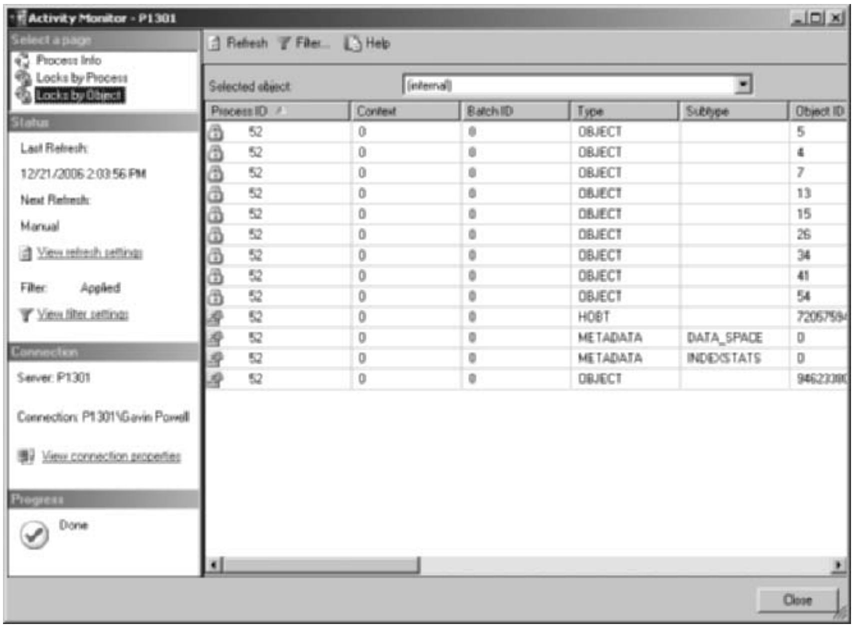
In Figure 10.15, locks can be isolated based on the object causing a locking problem.

The objective of the Activity Monitor and its locking monitoring tools is to allow for locks to be removed—if a lock is causing some kind of performance problem. In extreme cases, a lock such as a deadlock can even cause a database halt.

10.2.9.3 Using the System Monitor

The System Monitor is a Windows utility that enables system managers and database administrators to monitor the many objects within a Windows system. There are many counters that can be monitored for many objects, but here we are interested in those counters specific to the SQL Server:Locks object. These counters are shown in Table 10.3.

The counters shown in Table 10.3 are for a particular instance of locked object. The instances that can be monitored are as follows:



Last Refresh:
12/21/2006 2:03:56 PM
Next Refresh:
Manual
View refresh settings
Filter: Applied
View filter settings
Server: P1301
Connection: P1301\Gavin Powell
View connection properties
Done

Selected object: (internal)

Process ID	Context	Batch ID	Type	Subtype	Object ID
52	0	0	OBJECT		5
52	0	0	OBJECT		4
52	0	0	OBJECT		7
52	0	0	OBJECT		13
52	0	0	OBJECT		15
52	0	0	OBJECT		26
52	0	0	OBJECT		34
52	0	0	OBJECT		41
52	0	0	OBJECT		54
52	0	0	HOBT		7205759
52	0	0	METADATA	DATA_SPACE	0
52	0	0	METADATA	INDEXSTATS	0
52	0	0	OBJECT		9462338

Figure 10.15
Locks by Object

Table 10.3 *Counters monitored for the SQL Server lock object*

SQL Server: Locks	
Object Counters	**Explanation**
Average Wait Time (ms)	Average amount of wait time (in milliseconds) for each lock request that resulted in a wait.
Lock Requests/sec	Number of new locks and lock conversions per second requested from the lock manager.
Lock Timeouts/sec	Number of lock requests per second that timed out, including internal requests.
	for NOWAIT locks.
Lock Wait Time (ms)	Total wait time (in milliseconds) for locks in the last second.
Lock Waits/sec	Number of lock requests per second that could not be satisfied immediately and required the caller to wait before being granted the lock.
Number of Deadlocks/sec	Number of lock requests per second that resulted in a deadlock.

- RID
- Key
- Page
- Extent
- Table
- Database
- Total

This allows us to monitor counters for a particular type of lock or for all locks (Total).

Note: The System Monitor differentiates between SQL Server 2005 instances. An instance named PEGASUS\SQL_A running on server PEGASUS will have a locks object named MSSQL$SQL_A:Locks.

10.2.9.4 Interrogating the syslockinfo table

The syslockinfo system table can be interrogated in the same way that any other system table can be interrogated. It is only found in the master database, where it holds information concerning the locks held in SQL Server. Unlike most other system tables, it is materialized when a query is executed that accesses it; otherwise, it does not exist physically. A query issued against the syslockinfo table produces the following output:

```
SELECT rsc_text, rsc_dbid, rsc_indid, rsc_objid, rsc_type,
req_mode, req_status, req_spid FROM sys.syslockinfo

rsc_text       rsc_dbid rsc_indid  rsc_objid rsc_type req_mode req_status req_spid
                      7         0          0        2        3          1       52
                      7         0          0        2        3          1       51
         1:113        7         2  965578478        6        7          1       52
         1:113        7         2  965578478        6        6          1       51
(4501518d90d1)        7         2  965578478        7        4          1       52
(4501518d90d1)        7         2  965578478        7        3          1       51
      1:348:14        7         0  965578478        9        4          1       52
             :
             :
```

Not all the columns from syslockinfo have been displayed, since some are a binary representation of the ones shown and some are for Microsoft internal use. The displayed columns have the definitions shown in Table 10.4.

Table 10.4 *Column definitions for the syslockinfo system table*

Column	Definition
`rsc_text`	Textual description of a lock resource
`rsc_dbid`	The database ID of the resource
`rsc_indid`	The index ID of the resource if an index
`rsc_objid`	The object ID of the resource if an object
`rsc_type`	The type of resource—e.g., page
`req_mode`	The mode of the lock—e.g., shared (S)
`req_status`	The status of the lock—e.g., granted
`req_spid`	The SPID owning the lock

Examples of common values for rsc_type are shown in Table 10.5.

Table 10.5 *Values for the rsc_type column in syslockinfo*

Value	Type
1	NULL Resource
2	Database
3	File
4	Index
5	Table
6	Page
7	Key
8	Extent
9	RID
10	Application

Apart from the locks we have already discussed, there are several other types of locks. File locks tend to be acquired when a file is being added to a database, or a file is being shrunk, or similar file-related activities. Extent locks are used by SQL Server to internally manage the allocation and deallocation of extents. Extents, as discussed in Chapter 4, are of types mixed and uniform and are 64 KB (eight pages) in size. These locks can often be seen while you are inserting rows into a table. Index locks can be seen when an index is being created on a table.

The column req_mode represents the mode of the lock requested. We have discussed most of the common ones. There are, however, a number of more obscure modes, and we will list these here for completeness. Numbers greater than 12 are used for key range locks, discussed later. The req_mode values are listed in Table 10.6.

Table 10.6 *Values for the req_mode column in syslockinfo*

Value	Lock Mode Code	Lock Mode Name
0	NULL	Used as a placeholder only
1	Sch-S	Schema stability
2	Sch-M	Schema modification
3	S	Shared
4	U	Update
5	X	Exclusive
6	IS	Intent Shared
7	IU	Intent Update
8	IX	Intent Exclusive
9	SIU	Shared Intent Update
10	SIX	Shared Intent Exclusive
11	UIX	Update Intent Exclusive
12	BU	Bulk
13	RangeS_S	Shared Key Range + Shared Resource
14	RangeS_U	Shared Key Range + Update Resource
15	RangeI_N	Insert Key Range + NULL Resource
16	RangeI_S	RangeI_N +S
17	RangeI_U	RangeI_N + U

Table 10.6 *Values for the req_mode column in syslockinfo (continued)*

18	RangeI_X	RangeI_N +X
19	RangeX_S	RangeI_N + RangeS_S
20	RangeX_U	RangeI_N + RangeS_U
21	RangeX_X	Exclusive Key Range + Exclusive Resource

Note: A range lock will lock a range of key values (in an index not the table). Range locks help to maintain data consistency for transactions.

The schema stability locks are used to control access to schema objects, such as tables, to avoid problems where multiple connections are referencing an object that a connection wishes to modify or drop. The SIX, SIU, and UIX locks are special kinds of intent locks. The bulk lock is used when bulk loads are being performed into a table—for example, when the data transformation services BULK INSERT task is used to load data into a table and the option Table Lock has been set on the task. There are also some other lock modes used for serializable transactions, which we will discuss later.

Finally, the req_status column allows just three values, as shown in Table 10.7.

Table 10.7 *Values for the req_status Column in syslockinfo*

Value	Status
1	Granted
2	Converting
3	Waiting

10.2.9.5 Using the system procedure sp_who

The system procedure sp_who can be used to obtain information on the processes active within SQL Server. It can be entered as a stand-alone statement, in which case it will display information about all users and processes. It can take a SQL Server process identifier (SPID) or alternatively a SQL Server login name as a parameter. Also, the parameter value ACTIVE can be used, which eliminates user connections that are waiting for input from the user—that is, with AWAITING COMMAND in the cmd column. Some example output from the sp_who system stored procedure is as follows:

```
SPID ecid status     loginame hostname blk dbname    cmd

1    0    background sa                0   NULL      LAZY WRITER
2    0    sleeping   sa                0   NULL      LOG WRITER
3    0    background sa                0   master    SIGNAL HANDLER
4    0    background sa                0   NULL      LOCK MONITOR
5    0    background sa                0   master    TASK MANAGER
6    0    sleeping   sa                0   NULL      CHECKPOINT SLEEP
7    0    background sa                0   master    TASK MANAGER
8    0    background sa                0   master    TASK MANAGER
9    0    background sa                0   master    TASK MANAGER
10   0    background sa                0   master    TASK MANAGER
51   0    runnable   sa       PEGASUS  0   master    SELECT
52   0    sleeping   dave     PEGASUS  0   BankingDB AWAITING COMMAND
53   0    sleeping   sue      PEGASUS  52  BankingDB UPDATE
54   0    sleeping   tony     PEGASUS  0   BankingDB AWAITING COMMAND
```

Note that the process with SPID 53 has a value of 52 in the blk column, whereas other processes have 0. This is because the process with SPID 53 is being blocked by another user—in fact, the user with SPID 52.

Note: Microsoft also ships a stored procedure, called sp_who2. This outputs more information and in a slightly more readable form than sp_who.

10.2.9.6 The SQL Server Profiler

The SQL Server Profiler will be discussed in detail in Chapter 12. However, we need to mention it here, since it has capabilities that help us investigate lock problems. The SQL Server Profiler allows us to trace events graphically into a table and/or into a file. If the events are captured into a file or table, they can be analyzed later.

The Locks Event Category contains a number of Locks Event Classes, and these are shown in Table 10.8.

When an event is traced, the SQL Server Profiler captures various pieces of information about the event. These pieces of information are specified as Data Columns in the trace definition. Many data columns always contain the same information, regardless of the event class being traced. For exam-

Table 10.8 *Lock Event Classes in the SQL Server Profiler*

Event Class	Description
Lock:Acquired	A lock has been taken out on a row, page, etc.
Lock:Cancel	A held lock has been cancelled—e.g., by a deadlock.
Lock:Deadlock	A deadlock has occurred.
Lock:Deadlock Chain	The events preceding a deadlock.
Lock:Escalation	Lock escalation has occurred—e.g., a row escalated to a table.
Lock:Released	A lock has been taken off a row, page, etc.
Lock:Timeout	A lock has timed out.

ple, the CPU column is the amount of CPU in milliseconds used by the event. However, some data columns contain values that are specific to a particular event class. For the Lock Event Class there are some very useful data columns.

Generally, the Binary Data column contains the resource ID for a lock event class and the Object ID contains the ID of the object participating in the lock. Duration tends to represent wait time and the Mode represents the lock mode.

With a little practice some elements of the resource ID can be recognized and decoded as the lock type. If the SQL Server Profiler is being used interactively, this is done for you. Selecting the lock event with the mouse pointer will display the lock type.

10.2.9.7 Using trace flags with DBCC

The SQL Server documentation states that trace flag behavior may or may not be supported in future releases. It is worth mentioning this here, though, since trace flags can be used to provide some lock trace information. The database consistency checker, more usually referred to as DBCC, can be used to set trace flags, or they can be set if SQL Server is started at the command line or via the Startup Parameters in the General tab of Server Properties in the SQL Server Enterprise Manager. Trace information can be sent to destinations such as the errorlog (using trace flag 3605) or the client (using trace flag 3604). Locking information can be generated by setting the trace flags to 1200 or, for deadlock information, 1204 and 1205. An example trace output is as follows:

```
DBCC TRACEON (3604,1200)

SELECT SUM(balance) FROM accounts

Process 51 acquiring S lock on KEY: 7:1:2 (9c0206b5c98d) (class bit0 ref1)
result: OK
Process 51 acquiring S lock on KEY: 7:1:1 (ee006c4e98d2) (class bit0 ref1)
result: OK
Process 51 acquiring Schema lock on TAB: 7:965578478 [] (class bit0 ref1)
result: OK
Process 51 acquiring S lock on KEY: 7:3:2 (9302d58cf78b) (class bit0 ref1)
result: OK

:
Process 51 acquiring S lock on PAG: 7:1:41 (class bit0 ref1) result: OK
Process 51 releasing lock on PAG: 7:1:41
Process 51 acquiring S lock on PAG: 7:1:42 (class bit0 ref1) result: OK
Process 51 releasing lock on PAG: 7:1:42
Process 51 acquiring S lock on PAG: 7:1:50 (class bit0 ref1) result: OK
Process 51 releasing lock on PAG: 7:1:50
Process 51 acquiring S lock on PAG: 7:1:91 (class bit0 ref1) result: OK
Process 51 releasing lock on PAG: 7:1:91
Process 51 acquiring S lock on PAG: 7:1:160 (class bit0 ref1) result: OK
Process 51 releasing lock on PAG: 7:1:160
:
Process 51 releasing lock on TAB: 7:965578478 []
```

The output can be somewhat cryptic, but with a little effort a database administrator can follow what is happening. In this example, SPID 51 is performing a table scan and, after some initial reading of the system tables, is sequentially reading pages. When it wants to read a page, it requests and acquires a page lock; when it has read a page, it releases the page lock. Note that page locks refer to page numbers, whereas table locks (we will have taken out an intent table lock) refer to the object ID of the table. As we have seen, the OBJECT_NAME() function can be used to find the table name, as follows:

```
SELECT OBJECT_NAME (965578478)

accounts
```

Whether tables or pages are being referenced, the number preceding the object ID or page number is the database ID. The DB_NAME() function can be used to find the database name, as follows:

```
SELECT DB_NAME(7)

BankingDB
```

To find which object a page belongs, use the DBCC PAGE statement, as follows:

```
DBCC TRACEON (3604)
DBCC PAGE (7,1,50,0)

PAGE: (1:50)

BUFFER:

BUF @0x10EB7FC0

bpage = 0x1BA2E000      bhash = 0x00000000  bpageno = (1:50)
bdbid = 7               breferences = 1  bstat = 0x9
bspin = 0               bnext = 0x00000000

PAGE HEADER:

Page @0x1BA2E000

m_pageId = (1:50)       m_headerVersion = 1  m_type = 1
m_typeFlagBits = 0x0  m_level = 0              m_flagBits = 0x8000
m_objId = 965578478      m_indexId = 0  m_prevPage = (0:0)
m_nextPage = (0:0)      pminlen = 424  m_slotCnt = 16
m_freeCnt = 1232        m_freeData = 6928  m_reservedCnt = 0
m_lsn = (1274:16:151)  m_xactReserved = 0  m_xdesId = (0:0)
m_ghostRecCnt = 0       m_tornBits = 805306369

Allocation Status

GAM (1:2) = ALLOCATED  SGAM (1:3) = NOT ALLOCATED
PFS (1:1) = 0x63 MIXED_EXT ALLOCATED  95_PCT_FULL  DIFF (1:6) =
CHANGED
ML (1:7) = NOT MIN_LOGGED
```

The field containing the object ID is in bold type. Note also that to the right of that field is the index ID of the index to which the page belongs, if it is an index page.

The DBCC PAGE statement is specifying, in order, the database ID of 7, the file ID of 1, the page number 50, and 0 to indicate that we only need to see the header, not the data.

This trace flag returns the type of locks participating in a deadlock and the current commands involved. I usually set this trace flag with trace flag 3605 (log to errorlog). Here is some example output when a deadlock occurrs:

```
10:39:49.10 spid4  Deadlock encountered .... Printing deadlock
information

10:39:49.10 spid4   Owner:0x1b69f380 Mode: X Flg:0x0 Ref:0
Life:02000000 SPID:55
ECID:0

10:39:49.10 spid4  SPID: 55 ECID: 0 Statement Type: UPDATE Line #: 1
10:39:49.10 spid4
10:39:49.10 spid4      Wait-for graph
10:39:49.10 spid4
10:39:49.10 spid4      Node:1
10:39:49.10 spid4      RID: 7:1:537:14 CleanCnt:1 Mode: X Flags: 0x2
10:39:49.10 spid4      Grant List::
10:39:49.11 spid4      Input Buf: UPDATE CUSTOMERS SET customer_lname
= 'Phillips'
 WHERE customer_no = 1000

10:39:49.11 spid4  Requested By:

10:39:49.11 spid4  ResType:LockOwner Stype:'OR' Mode: U SPID:53
ECID:0

Ec:(0x1b9e13e0)
Value:0x1b6a3300 Cost:(0/A0)

10:39:49.11 spid4

10:39:49.11 spid4  Node:2

10:39:49.11 spid4  RID: 7:1:338:9 CleanCnt:1 Mode: X Flags: 0x2
```

```
10:39:49.11 spid4  Grant List::

10:39:49.11 spid4 Owner:0x1b69f2e0 Mode: X Flg:0x0 Ref:0
Life:02000000
SPID:53 ECID:0

10:39:49.11 spid4  SPID: 53 ECID: 0 Statement Type: UPDATE Line #: 1

10:39:49.11 spid4  Input Buf: UPDATE ACCOUNTS SET balance = 99
 WHERE account_no = 2000

10:39:49.11 spid4  Requested By:

10:39:49.11 spid4  ResType:LockOwner Stype:'OR' Mode: U SPID:55
ECID:0

Ec:(0x1c1cd3e0)
Value:0x1b6a33c0 Cost:(0/98)

10:39:49.11 spid4  Victim Resource Owner:

10:39:49.11 spid4  ResType:LockOwner Stype:'OR' Mode: U SPID:55 ECID:0
Ec:(0x1c1cd3e0) Value:0x1b6a33c0 Cost:(0/98)
```

I have removed the data from the date/time to fit more information onto the page. We can see that a deadlock was encountered, and by examining the output we can see the following:

- SPID 53 and SPID 55 are involved in a deadlock.
- Resources involved are RID: 7:1:537:14 and RID: 7:1:338:9.
- The last statements sent by the participating connections were:

```
'UPDATE CUSTOMERS SET customer_lname = 'Phillips'

   WHERE customer_no = 1000 '
'UPDATE ACCOUNTS SET balance = 99 WHERE account_no =
2000'
```

- SPID 55 was chosen as the deadlock victim.
- The locks involved were update (U) locks.

This trace flag returns more detailed information about the deadlock. You will need to set trace flag 1204 to get information out of trace flag 1205, but, to be honest, the extra information is probably only likely to be useful (and understandable) by Microsoft Support.

10.3 SQL Server locking in action

Now that we understand how SQL Server uses its locking protocol, we can look at some examples. Our examples will all follow the same format, that of the T graph. Some people believe it is called a T graph because it looks like a T; others believe it is because the vertical axis represents time! Whatever the reason, it is a useful method for representing the interaction of locks in a multiuser scenario. In order to keep the output as clear as possible, the actual results of the SELECT statements are not shown.

Our examples will use the Accounts table in the BankingDB database. In these examples, all indexes have been removed from this table unless otherwise specified. Also, until we change it, the default locking protocol will be used—that is, transaction isolation level read committed:

```
Mike                              Katy
SELECT * FROM accounts                  SELECT * FROM accounts
WHERE account_no = 1000                 WHERE account_no = 2000

*** OK ***                                *** OK ***
```

In the above example, Mike retrieves all the rows in the Accounts table. Katy attempts to concurrently retrieve all the rows in the Accounts table and is successful. This is because Mike places and releases shared locks on the rows in the Accounts table as he scans through it. Katy also attempts to place shared locks on the rows in the Accounts table, and, since shared locks are compatible, her attempt is successful.

In the following example, Mike updates all the rows in the Accounts table. He performs this operation within a transaction, which he does not end. Katy attempts to retrieve rows from the Accounts table:

```
Mike                                   Katy
BEGIN TRANSACTION
```

```
UPDATE accounts SET balance = 0
 WHERE account_no = 1000

                                            SELECT * FROM accounts
                                             WHERE account_no = 2000

*** OK ***

                                            *** wait ***
```

In this example, Mike is updating a row in the Accounts table, and so SQL Server takes out an exclusive (X) row lock. Katy's SELECT statement needs to search the table looking for rows that match her criteria (account_no = 2000). SQL Server decides that it is efficient to search using page locks. This is not unreasonable, since it knows it will be retrieving every row on every page. This is because, with no indexes present, a table scan is performed, and every page must be retrieved from the Accounts table.

As Katy scans through the table acquiring and releasing shared (S) page locks, she reaches the page on which Mike has taken an exclusive (X) lock on his row. As SQL Server will have also placed an Exclusive Intent (IX) lock on the page in which his row resides, Katy's shared (S) page lock will be blocked. A shared (S) lock is not compatible with an exclusive intent (IX) lock.

This example serves to illustrate a very important point: Transactions should be kept as short as possible. If they are not, then they could block another transaction for an unacceptable length of time.

If we were to issue an sp_lock at this point, we would see the following fragment of output relating to Mike and Katy's connections:

```
SPID dbid ObjId     IndId Type Resource Mode Status

53   7    965578478 0     TAB           IS   GRANT
53   7    965578478 0     PAG  1:348    S    WAIT
54   7    965578478 0     RID  1:348:14 X    GRANT
54   7    965578478 0     PAG  1:348    IX   GRANT
54   7    965578478 0     TAB           IX   GRANT
```

Her shared lock is blocked on the page. An sp_who issued at this point would show the following columns (with some deleted):

```
SPID status   loginame hostname blk dbname

54   sleeping mike     PEGASUS  0   BankingDB
53   sleeping katy     PEGASUS  54  BankingDB
```

In the following example, Mike again updates all the rows in the Accounts table. Again, he performs this operation within a transaction, which he does not end. This time Katy attempts to delete the rows in the Accounts table:

```
Mike                           Katy
BEGIN TRANSACTION

UPDATE accounts SET balance = 0       BEGIN TRANSACTION
WHERE account_no = 1000

                                      DELETE FROM accounts
                                      WHERE account_no = 2000

*** OK ***

                                      *** wait ***
```

In this example, Katy attempts to place an update (U) lock on the rows in the Accounts table while searching for a row that meets her criteria for deletion. Since there are no indexes on the table, every row must be checked. Eventually Katy attempts to place an update (U) lock on the row Mike has just updated, which holds an exclusive (X) lock. An exclusive (X) lock is incompatible with all other locks, so Katy is blocked. If we were to issue an sp_lock at this point, we would see the following fragment of output relating to Mike and Katy's connections:

```
Spid dbid ObjId     IndId Type Resource Mode Status

53   7    965578478 0     RID  1:348:14 U    WAIT
53   7    965578478 0     PAG  1:348    IU   GRANT
53   7    965578478 0     TAB           IX   GRANT
54   7    965578478 0     TAB           IX   GRANT
54   7    965578478 0     RID 1:348:14  X    GRANT
54   7    965578478 0     PAG 1:348     IX   GRANT
```

We can see Katy's blocked update (U) lock on row 1:348:14. This example is similar to the previous example with the exception that Katy is searching with update (U) locks on rows rather than shared (S) locks on pages.

In the following example Mike will again update rows in the Accounts table and Katy will retrieve them. This is the same as the second example except that now Katy will issue her SELECT statement first. We will use BEGIN TRANSACTION for both users:

```
Mike                                          Katy
                            BEGIN TRANSACTION
                            SELECT * FROM accounts
                                                            WHERE
account_no = 2000
BEGIN TRANSACTION
UPDATE accounts SET balance = 0
 WHERE account_no = 1000
                            *** OK ***
*** OK ***
```

In this example, Katy attempts to place shared locks in the Accounts table. She is successful, since Mike has not issued his update yet. Mike then issues his update, which is also successful. Mike's exclusive lock is not blocked by Katy's shared locks, because SQL Server will have released the shared locks when the SELECT statement completed. Katy's locks were gone before Mike issued his update. The fact that Katy issues her SELECT statement within a transaction is irrelevant.

Because SQL Server runs at the default transaction isolation level of READ COMMITTED, shared locks are not held until the end of the transaction but are released as soon as the row or page is read. This increases concurrency (and therefore performance), but this does mean that the read is not guaranteed to be repeatable, as we shall see shortly.

Let us now create some indexes on the Accounts table:

```
CREATE UNIQUE NONCLUSTERED INDEX NCI_AccountNo
   ON accounts (account_no)
```

Mike will now update rows in the Accounts table while Katy attempts to delete them. We will use a WHERE clause in order to choose different rows:

```
Mike                                Katy
BEGIN TRANSACTION

UPDATE accounts SET balance = 0
WHERE account_no = 1000                      BEGIN TRANSACTION

                                             DELETE FROM accounts
                                             WHERE account_no = 2000


*** OK ***


                                             *** OK ***
```

Both users succeeded. This is because indexed access can now be used, and, consequently, row-level locks can be taken out just on the resources required. If we were to issue an sp_lock at this point, we would see the following fragment of output:

```
Spid dbid ObjId     IndId Type Resource      Mode Status

53   7    965578478 0     RID  1:537:14      X    GRANT
53   7    965578478 2     KEY (ea003d68f923) X    GRANT
53   7    965578478 0     PAG  1:537         IX   GRANT
53   7    965578478 2     PAG  1:2612        IX   GRANT
53   7    965578478 0     TAB                IX   GRANT
54   7    965578478 0     TAB                IX   GRANT
54   7    965578478 0     RID 1:348:14       X    GRANT
54   7    965578478 0     PAG 1:348          IX   GRANT
```

We can see that all locks have been granted. Katy (SPID 53) holds exclusive locks on a row and an index entry. This is because her delete will not only remove the row but will also remove the index entry. Mike holds an exclusive lock on the row only, since he will not change the index entry in any way—he is updating the balance column, not the account_no column.

Suppose Mike and Katy insert rows into the Accounts table. Let us assume that there are no indexes on the Accounts table:

```
Mike                          Katy
BEGIN TRANSACTION

INSERT INTO accounts VALUES
(112501, 2000, 1000, 1510.77,
'some notes')

                                     BEGIN TRANSACTION

                                     INSERT INTO accounts VALUES
                                     (112502, 2012, 987, 123.78,
                                     'some notes')

*** OK ***

                                     *** OK ***
```

There is no problem. Because SQL Server supports row-level locking, there is generally no blocking on insert. The same is true if indexes are present on the table, since the individual index entries will be locked with KEY locks.

10.4 Uncommitted data, non-repeatable reads, phantoms, and more

With our knowledge of locking protocols we can now investigate how SQL Server deals with the reading of uncommitted data, non-repeatable reads, and phantoms.

10.4.1 Reading uncommitted data

Figure 10.1 illustrated the problems with reading uncommitted data. As should already be clear, SQL Server forbids this by virtue of the fact that any row that has been changed cannot be read by another user, since an exclusive lock will prevent the row from being retrieved until the write transaction ends.

SQL Server, however, allows the default behavior to be overridden. A query is allowed to read uncommitted data with the use of the READUNCOMMITTED keyword, introduced earlier in this chapter. For example, the following SELECT statement would read the row from the Accounts table regardless of whether another transaction had a row locked with an exclusive lock:

```
SELECT balance FROM accounts WITH (READUNCOMMITTED)
   WHERE account_no = 15000
```

The lock hint is recommended rather than NOLOCK, which is retained for backward compatibility.

Suppose Mike updates a row in the Accounts table. He performs this operation within a transaction, which he does not end. Katy attempts to retrieve rows from the titles table:

```
Mike                         Katy
BEGIN TRANSACTION

UPDATE accounts SET balance = 500
 WHERE account_no = 5000

                                  SELECT balance FROM accounts

                                  WITH (READUNCOMMITTED)
                                   WHERE account_no = 5000

*** OK ***

                                  *** OK ***
```

In this example, Katy does not attempt to place a shared lock, and she can read the row that Mike has updated. She will read a balance of 500. Mike may well ultimately choose to roll back his change, leaving Katy with incorrect balance information.

This behavior is the same as if the connection had set the transaction isolation level to READ UNCOMMITTED. However, the behavior would apply to all the transactions executed on that connection until another SET TRANSACTION changed the isolation level, or the statement overrode the isolation level for itself with a lock hint.

10.4.2 Non-repeatable reads

In the case of a non-repeatable read, a transaction is allowed to read a data item on more than one occasion and retrieve different values each time. This is shown in Figure 10.16. By default, SQL Server allows non-repeatable reads. It is sometimes desirable, however, to guarantee repeatable reads—that is, each read of the same data item while in the same transaction returns the same value. The means of guaranteeing repeatable reads in SQL Server is by the use of the REPEATABLEREAD keyword.

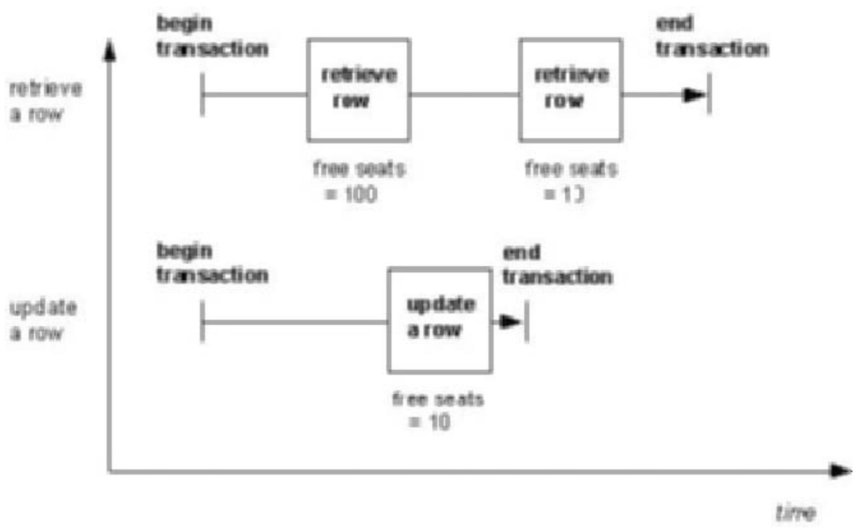


Figure 10.16 *Non-repeatable reads*

If the REPEATABLEREAD keyword is used, the page is read the first time a shared lock is taken out as usual. This then remains until the end of the transaction. This blocks any other transaction from changing the data item:

```
Mike                                Katy
                                    BEGIN TRANSACTION
                                    SELECT balance FROM accounts WITH
(REPEATABLEREAD)
                                     WHERE account_no = 5000
                                    *** OK ***
BEGIN TRANSACTION
UPDATE accounts SET balance =
50.00
 WHERE account_no = 5000
*** wait ***
                                    SELECT balance FROM accounts
                                       WITH (REPEATABLEREAD)
                                     WHERE account_no = 5000
                                    *** OK ***
```

Now Mike is forced to wait. Katy's shared locks block Mike's exclusive lock, and when Katy repeats her read she will receive the same value—hence, the use of the REPEATABLEREAD keyword has provided repeatable reads. Again, this is at the expense of concurrency.

Setting the isolation level to REPEATABLE READ (or SERIALIZABLE) will also provide repeatable reads:

```
Mike                           Katy
                               SET TRANSACTION
                                  ISOLATION LEVEL REPEATABLE
                                              READ
                                              BEGIN TRANSACTION
                               SELECT balance FROM accounts
                              WHERE account_no      = 5000
                              *** OK ***
BEGIN TRANSACTION
UPDATE accounts SET balance =
50.00
 WHERE account_no    = 5000
*** wait ***
                               SELECT balance FROM accounts
                                      WHERE account_no        = 5000
                               *** OK ***
```

Again, Mike is forced to wait. Katy's shared locks block Mike's exclusive lock, and when Katy repeats her read she will receive the same value. The use of the REPEATABLEREAD lock hint is not required, since the set transaction isolation level repeatable read statement has provided repeatable reads.

10.4.3 Phantoms

The phantom problem was illustrated in Figure 10.2. By default, SQL Server does not forbid phantoms, but the use of the SERIALIZABLE hint will prevent them, as the following examples show:

```
Mike                            Katy
BEGIN TRANSACTION
```

```
SELECT SUM(balance) FROM accounts
 124961532.6600

*** OK ***

                    INSERT INTO accounts VALUES
                       (112502, 2012, 987, 123.78,
                    'some notes')

                    *** OK ***

SELECT SUM(balance) FROM accounts
 1249616510.4400

*** OK ***
```

In the previous example, phantoms are allowed to occur. The two sums of the same list of values, give different results. In the following example, Katy's transaction is blocked, and the phantom phenomenon is not allowed to occur.

```
Mike                    Katy
BEGIN TRANSACTION
SELECT SUM(balance)
FROM accounts WITH
(SERIALIZABLE)
 124961532.6600
*** OK ***
                        INSERT INTO accounts VALUES
                           (112502, 2012, 987, 123.78,
                        'some notes')
                        *** wait ***

Mike                    Katy
SELECT SUM(balance)
FROM accounts WITH
(SERIALIZABLE)
 124961532.6600

*** OK ***
```

The use of the SERIALIZABLE keyword is not required if the set transaction isolation level serializable is used:

```
Mike              Katy
SET TRANSACTION
 ISOLATION LEVEL SERIALIZABLE

BEGIN TRANSACTION
SELECT SUM(balance) FROM accounts
 124961532.6600
*** OK ***

                   INSERT INTO accounts VALUES
                      (112502, 2012, 987, 123.78,
                   'some notes')
                   *** wait ***

SELECT SUM(balance) FROM accounts
 124961532.6600
*** OK ***
```

Note that the SERIALIZABLE lock hint is recommended rather than HOLDLOCK, which is retained for backward compatibility.

To enforce serializability the lock manager must use some special techniques. In a sense, if we consider our previous example, the lock manager must lock something that does not exist! It cannot lock the row that Katy inserts, because it does not exist at the time of the first SELECT operation. Now SQL Server could lock the whole table if it wanted to, and, if there were no relevant indexes on the table, this is possibly what it might do. This would certainly stop phantoms.

However, if there are indexes on the table, then the SQL Server lock manager uses a technique known as key-range locking. A key-range lock works by covering the index rows and the ranges between those index rows. Any row insertion, update, or deletion within the range by another connection that requires a modification to the index causes the second connection to wait.

For example, suppose we execute the following query against the Branches table:

```
SELECT branch_no, branch_name FROM branches
WHERE branch_name BETWEEN 'Ealing' AND 'Exton'
```

We find the following branch names:

```
branch_no      branch_name
1081           Ealing
1021           Eden
1031           Edmonton
1051           Elton
1061           Epsom
1071           Eton
1041           Exton
```

We may want to ensure that we cannot insert a new branch between executions of this query. To do this we run the statement with the SERIALIZABLE lock hint:

```
BEGIN TRANSACTION
SELECT branch_no, branch_name FROM branches WITH
(SERIALIZABLE)
   WHERE branch_name BETWEEN 'Ealing' AND 'Exton' : :
```

If we investigate the locks acquired during this transaction, we find the following (simplified) output from sp_lock:

```
SPID dbid ObjId     IndId Type Resource ModeStatus

57   7    0         0     DB                 S          GRANT
57   7    981578535 2     KEY  (680236ce107b) RangeS-S   GRANT
57   7    981578535 0     PAG  1:102          IS         GRANT
57   7    981578535 0     PAG  1:103          IS         GRANT
57   7    981578535 0     PAG  1:100          IS         GRANT
57   7    981578535 2     KEY  (b8020849fa4b) RangeS-S   GRANT
57   7    981578535 2     KEY  (b802f9924eb9) RangeS-S   GRANT
57   7    981578535 2     KEY  (b702b7e93c9b) RangeS-S   GRANT
57   7    981578535 2     KEY  (b002a45d0732) RangeS-S   GRANT
57   7    981578535 2     KEY  (b802194c7ac6) RangeS-S   GRANT
57   7    981578535 2     KEY  (6c028abdf769) RangeS-S   GRANT
```

There are eight key locks acquired, but if we look at the mode we can see RangeS-S. This tells us that these are key-range locks. Basically, a key-range lock covers a range of values starting with the key before the key that is locked.

In our example, the first branch name in our range is Ealing. The branch name preceding the start of our range is Ducklington. The key-range lock on the index entry Ealing would cover Ducklington to Ealing and this would then prevent a branch being inserted with the name Eaglesfield or Duddington, because those key values lie in between Ducklington and Ealing. In theory this is too restrictive, since these are not in our range. This said, key-range locking is pretty good and a lot better than locking the whole page or table; after all, we can successfully insert the local branch in Duchally!

Similarly, the branch name following the end of our range is Fairford. We would not be able to insert branches named Eyam or Failsworth, but we would be able to insert Fairlight.

Note: In fact, we would be able to insert branches named Ducklington or Fairford but, of course, only if the index on branch_name was not unique.

The number of RangeS-S locks held is $N + 1$, where N is the number of rows that satisfy the query. In our case, seven rows satisfy the query, so eight RangeS-S locks are held.

The name of the key-range mode is in two parts. The RangeS part represents the lock mode protecting the range between two consecutive index entries, and the part after the "-" represents the lock mode protecting the index entry itself. So, RangeS-S means the range is locked in shared mode and the index entry itself is locked in shared mode. Another key range mode is RangeS-U. The difference between RangeS-S and RangeS-U is similar to the difference between shared (S) and update (U) locks, which has been discussed previously. RangeX-X is used when a key in a range is updated. Finally, RangeI-N is used as a probe to test ranges before inserting a key into an index.

10.4.4 More modified locking behavior

While showing examples of how the lock hints and transaction isolation levels can modify the default locking behavior, it is also worth looking at examples of some of the other lock hints introduced earlier in this chapter. An interesting lock hint is READPAST. Consider the case when we have no index on the Accounts table:

```
Mike                Katy
```

```
BEGIN TRANSACTION

UPDATE accounts SET balance = 0
 WHERE account_no = 1000

                   SELECT * FROM accounts
                      WHERE account_no = 2000

*** OK ***

                   *** wait ***
```

This was our second example. Katy is forced to wait because her sequential table scan hits Mike's locked row and cannot get past it. With the READPAST lock hint Katy will skip the locked row and continue searching:

```
Mike                  Katy
BEGIN TRANSACTION

UPDATE accounts SET balance = 0
 WHERE account_no = 1000

                  SELECT * FROM accounts WITH
                  (READPAST)
                     WHERE account_no = 2000

*** OK ***

                  *** OK ***
```

10.5 Application resource locks

SQL Server exposes an interface to its lock manager with the system stored procedure sp_getapplock and sp_releaseapplock. Suppose we execute sp_getapplock, as follows:

```
DECLARE @resultcode int
EXEC @resultcode = sp_getapplock @Resource = 'Store 5'
, @LockMode = 'Exclusive'
, @LockOwner = 'Session'
```

We are taking out an exclusive lock on a resource named Store 5. Although this resource may have no relationship to objects in the SQL Server database, we are able to use the SQL Server 2000 lock manager to manage our application lock protocol. Any other connection attempting to take out a lock on a resource named Store 5 will be forced to wait.

An application resource lock may be acquired with an owner of Transaction (the default) or Session. If the owner is Transaction the application resource lock behaves like any other lock acquired in an explicit transaction—it will disappear when the transaction completes with a commit or rollback. However, if the owner is Session, the application resource lock will be held until it is explicitly released with the system stored procedure sp_releaseapplock. For example:

```
DECLARE @resultcode int
EXEC @resultcode = sp_releaseapplock @Resource = 'Store 5'
, @LockOwner = 'Session'
```

This is very useful, since it means that an application resource lock may be acquired for a period of time that is independent of the individual SQL Server transactions that are being performed on the underlying data. In our example, we can take out an application resource lock on a resource known as Store 5. This stops any other user from working on Store 5. However, our inserts, updates, and deletes against the database data that represent Store 5 can be performed in very short transactions, so normal SQL Server resource locks do not become bottlenecks.

10.6 A summary of lock compatibility

We have seen a number of scenarios involving locks and it is worth now summarizing the compatibility between different locks. Locks can be Shared (S), Exclusive (X), or Update (U). They can also be intent shared (IS), Intent Exclusive (IX), or Intent Update (IU). These interact as shown in Table 10.9.

We mentioned schema stability locks earlier in this chapter. They too have a compatibility. The schema stability lock (Sch-S) is compatible with all lock modes except the schema modification lock (Sch-M). The schema modification lock (Sch-M) is incompatible with all lock modes. The bulk update (BU) lock is compatible only with schema stability and other bulk update locks. This is how parallel BCP loads are possible.

Table 10.9 *Lock Compatibility*

Mode of Requested Lock		Mode of Currently Granted Lock				
	IS	S	U	IX	SIX	X
Intent Shared (IS)						
Shared (S)						
Update (U)						
Intent Exclusive (IX)						
Shared with Intent Exclusive (SIX)						
Exclusive (X)						

In a multiuser system that has not been designed with concurrency in mind, lock conflict is often the cause of performance degradation, and the effects of this are second only to the effects of bad query and index design.

II

Architectural Performance Options and Choices

Some SQL Server database choices are architectural ones. A database architectural choice is essentially an additional option, as well as the basic SQL Server 2005 database engine. For example, a decision to use replication to either distribute or ensure failover capability is a software architectural choice.

It's not always the case, but most often, the overall objective of using add on options, such as replication, is all about availability and scalability. Making a database more available means that your database is available 365 days a year, 24 hours a day—that's the most extreme scenario. Not all databases require 100% availability. Any time a database is not available is known as downtime. Downtime is where nothing and nobody can talk to your database—apart from the database administrator fixing the problem that caused the database to go down in the first place. Making a database more scalable means you want to allow a lot more users to talk to your database at the same time. This is known as concurrency. Making a database more scalable basically lets you service a lot more potential customers. In the age of the Internet, and its global population, both availability and scalability are very important.

The process of performance tuning all of these different types of architectures is really a little too detailed for a book such as this one. However, simple provision of availability and scalability, by use of exotic add-on architectural components, is a performance tuning method. This is because service capacity is improved. So, it would make little sense to exclude topics such as replication, partitioning, and standby from this book.

11.1 The Management Studio and the .NET Framework

One of the most significant changes helping improve performance in SQL Server 2005 is a development issue rather than one of availability and scalability. This development improvement is the inclusion of all tools into the SQL Server Management Studio, plus the capabilities of writing and executing code in any programming language, using the .NET Framework. Essentially, developers can build databases and software much faster. Also, using the .NET Framework allows building of any kind of code, be it Visual Basic to allow for rapid coding and testing, or C programming to allow for ultra-fast execution times.

Note: Short descriptions of the Management Studio and the .NET Framework can be found in Chapter 1. Going into details of each gets more into database administration and programming, both of which are related to performance tuning, but moreover are each topics worthy of an entire book each.

11.2 Striping and mirroring

Before discussing topics such as standby mirrors and partitioning to stripe files into separate chunks, it is necessary to note the existence of striping and mirroring capabilities outside of the scope of SQL Server 2005.

11.2.1 RAID arrays

As seen in Chapter 9, there is something called RAID array. A RAID array is a bunch of disks, all grouped together, used as a single virtual hardware disk storage subsystem. All of the disk access is controlled by RAID hardware or software (or both hardware and software). Some really expensive RAID arrays are completely independent of the operating system, with their own built-in hardware control system and on-board memory (RAM). RAID arrays allow striping and mirroring.

Some operating systems will allow striping and mirroring within the disk subsystem, with everything controlled by the operating system. This type of software can also be a middleware type of application, residing between the operating system and the underlying hardware.

It is common for various relational database engines to provide inherent striping and mirroring from within the database. This includes SQL Server 2005. How this functionality is implemented is not dependent on a database vendor but more likely dependent on skills within an organization and preference.

11.2.2 Partitioning and Parallel Processing

Partitioning allows you to split data at the table level. So, instead of breaking up information by striping at the disk level, using something like a RAID array, you can split data logically. What is the result of splitting data logically? What it really means is that you can break up tables based on the data content of those tables. In a really simplistic form, you could partition a table between current rows, and archived rows. The result would be two partitions—one your OLTP data, the other your data warehouse analytical type data. Your online applications could run much faster because they only need to read the OLTP partition to answer customer queries over the Internet. And obviously, when running analytical reports, your customers will not be disrupted by huge read access requirements for monstrous analytical reporting techniques such as cubes.

The other beneficial side effect of partitioning is when combining it with parallel processing. A parallel processing capable platform will allow execution of more than one thing at the same time. Technically, this can increase processing times significantly. When combining parallel processing with partitioning, and executing the same query against multiple partitions in parallel, the performance benefits can be enormous.

Note: Partitioning is covered in more detail in Chapter 4. In Chapter 5 there is information of creation of parallel processing capable indexes. Chapter 9 contains a description of parallel queries.

11.3 Workflow management

Workflow management is the way in which tasks of work flow across an organization. With respect to an SQL Server database environment, this involves the passing of messages across a network, both within and outside of a company. The result is a computer system coordinating the flow of information between people and computers. Also included is tracking information, ways to prioritize messages and tasks, scheduling of automated tasks, reporting functions, and so on.

Computerized workflow management is software which manages and coordinates workflow between people's computers, using predetermined procedures, presenting each person in a team with all tasks and responsibilities.

The benefit to a business of using workflow management software, is increased flexibility with the ability to assess differing scenarios. So, in addition to effective management of people's time, is the ability to plan for varying scenarios, or pick the best scenario from a set of alternatives. The result is a smoother running business, more efficient use of time, and more effective and better computer resources—this includes an SQL Server 2005 database. Workflow management software is a part of SQL Server 2005 in the form of Notification Services and the Service Broker.

The SQL Server 2005 Service Broker will permit both internal and external processes, to send information from within Transact-SQL. That information is placed onto streams, much like a kind of full accessible pipeline, which computers can access by simply writing to and reading from the stream of information. SQL Server 2005 can queue messages onto a queue. That queue can be passed on to other SQL Server instances on a local or a remote server.

Notification Services allows for building of applications which can create and send notifications out to the user population. Those notifications can be personalized for any specific person, sent at specified times and automatically scheduled, to different types of targets. Those targets can include computers, cell phones, digital assistants, online messenger tools, or even an email account. Users who receive the messages subscribe to a specific notification service in order to receive the messages being sent on the message queue stream. Messages can even be sent to indicate some kind of event has occurred, such as sending an alert to a database administrator if a database is having problems.

11.4 Analysis Services and data warehousing

Analysis Services provides OLAP (Online Analytical Processing) and data mining. Data mining allows for drill-down and search into sources of data, allowing for searches on both specific items and patterns. OLAP, on the other hand, allows you to build metadata structures, in addition to tables, which can be read by queries against the database. This works by usually creating what is called a materialized view. A view contains a query and always reads data in underlying tables. A materialized view creates a copy of data in underlying tables, such that something called query rewrite can read data from the materialized view, and not the underlying tables. That's how it all works. Also consider that most OLAP type queries contain some kind of aggregation of one form or another. So, it is highly likely that a materialized view has far fewer rows than the underlying tables. Additionally, if there is more than one underlying table, a query would normally read a join—whereas a materialized view is effectively a single table (object), and no costly join operations are required.

This book does not need to go into the details of SQL Server 2005 Analysis Services, or the theory behind OLAP and how all the various queries are built. It does, however, make perfect sense to state that when building analytical or data warehouse reports consider Analysis Services. Do not attempt to write horribly complex and slow running queries yourself, running against normalized OLTP relational tables. Use Analysis Services to build that functionality.

11.4.1 Data modeling techniques in SQL Server 2005

The UDM or Unified Dimensional Model is new to SQL Server 2005, and more specifically to Analysis Services. The UDM implements a form of the dimensional data model.

Note: The dimensional model for data warehouses is described in Chapter 2.

Analysis Services graphically presents an entire suite of processes, which can be used to extract data from multiple heterogeneous data sources, store it in materialized form, and build queries and reports from that materialized data.

Once again, there is no need to go into the nitty-gritty details of Analysis Service because that would be a book in itself. However, use of Analysis Services, the built-in graphical tools, and the UDM presents end-users with more business-oriented tools and functionality—as compared to the underlying tables in what could be multiple relational databases, even from multiple vendors.

The result is easy integration and analysis of data. It provides something that end-users understand (those requiring analytical reporting). It doesn't need an army of database administrators and programmers to produce the data that management is really looking for.

11.5 Distribution and replication

Replication is a process of splitting data out to multiple databases, usually distributed out to multiple computers. Those computers can also be geographically dispersed, such as across an entire country. The separate sites can help performance of the database as a whole, because each replicated database is servicing a fraction of the user population. The result is less competition for user access at each site. Additionally, some data can be specific to each geographical site (you don't have to replicate everything).

In the extreme all data is replicated from all databases to all other databases—and it's done in real-time. The result is that all databases in a distributed network have all the same data copied to all databases, regardless of which database changes originate from.

There are two different types of replication. A simple model is master to slave replication as shown in Figure 11.1.

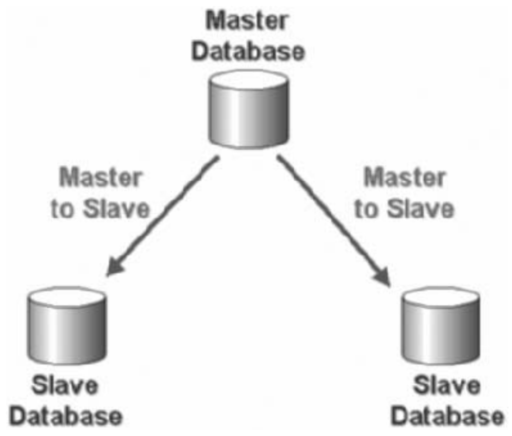


Figure 11.1
Master to slave replication

In master to slave replication there is a single master database. All other databases in the distribution network are slave databases. The master database is the only database which can be changed by users and applications. All changes are replicated (copied) out to the slave databases. The result is that users connected to a slave database get faster access to the database because of less competition (fewer users connected to each individual database). Also, users connected to a slave database don't have to talk to a master database on the other side of the country, across a long network cable.

Note: A slave database may not receive and update changes in real-time. Replication can be performed in real-time. Replication can be performed periodically depending on the needs of the users, and the limitations of the hardware.

A far more complex implementation of replication is to set up master to master replication as shown in Figure 11.2.

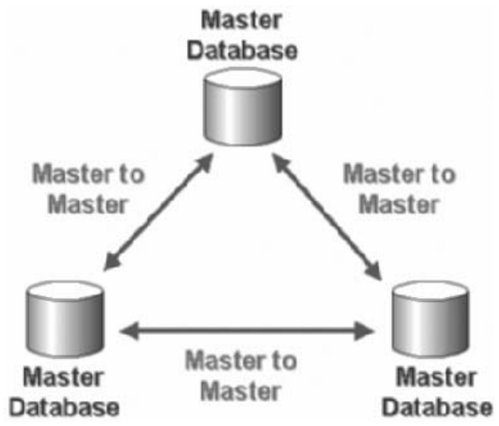


Figure 11.2
Master to master replication

In master to master replication all databases in the distributed network can accept changes from users and applications. Also, all databases can replicate changes out to all other databases in real-time (or periodically). The resource requirements for master to master replication are intense. In reality, it is likely that the best design is a mixture of master and slave databases across a distributed network, depending on requirements in different geographical locations.

That's the basics of replication. The downside of replication is that implementation is usually difficult because it is highly complex and detailed. One of the most interesting problems caused by replication is that it is often inappropriately used, such as for a failover database or even for backups. Replication is simply too complicated to use for tasks which can be resolved by more appropriate implementations.

There are other methods of implementing replication, without using replication itself. These methods are not recommended as they may require some serious programming skills. However, in some cases more innovative methods can function better than database vendor-provided replication tool sets. SQL Server 2005 Notification Services and the Service Broker could potentially be used to pass database changes onto a queue—and down to all other replicated databases as a stream. A picture of this is shown in Figure 11.3.

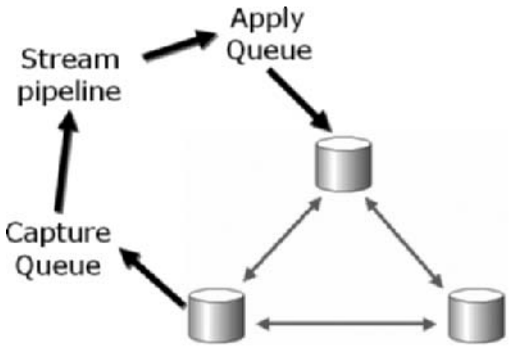


Figure 11.3 *Using streams to manually implement replication*

Materialized views are also a possibility but for master to slave replication only. This is because materialized views are created using underlying tables and cannot be used to replicate changes back into tables. This is not the case for all database vendors but the downside would be much more complexity. Of course, replication can be manually coded using something like Transact-SQL and triggers, but that might be just too complex to even consider.

The objective of SQL Server 2005 replication is to raise data availability by distributing multiple copies of data across multiple database servers. Users in different locations can then access data from their local database. In other words, the load for reading the database in queries is spread out. SQL Server 2005 replication functions by allowing a database to be declared as being a publisher or a subscriber. A publishing database produces new data (master database), and a subscriber subscribes to published data (slave database). The Replication Monitor tool is built into the Management Studio making implementation, maintenance, and monitoring of replication much easier than it has been in the past.

From a purely performance perspective, replication and distributing of data allows for lower numbers of concurrent users reading data from each database. However, replication can have its own critical performance issues, which are briefly as follows:

- **Latency and Synchronization:** The time to propagate data across the nodes (replicated databases) of a distributed network can be critical for users getting up-to-date results from queries. Some applications will require as real-time a response as is possible. For example, an airline booking system cannot be allowed to double book seats. Then again, airlines overbook flights constantly to ensure their flights are profitable. However, how many times have you been bumped off a flight? Myself? Never!
- **Throughput:** How much data replication can a distributed network manage? Again, if real-time replication is required then throughput has to be fast enough to get changes to all distributed databases as quickly as possible.
- **Concurrency of Replication:** Distribution of data to multiple replicated databases can help performance on each individual database. However, too much replication activity might result in the act of replication using too many resources on local databases, and perhaps negating the beneficial effect of distribution. In other words, if database changes are too frequent, they could swamp the concurrency capacity of individual databases in the network. The use of hardware resources cannot outweigh the benefit of data distribution.

In conclusion, replication is complex and can require heavy processing both on publishing and subscriber database servers. Used appropriately (if data latency is acceptable), replication can help performance in general. However, inappropriate use can hinder performance drastically. This is usually only discovered after the fact, so careful planning and thorough understanding is best gained before implementation.

11.6 Standby failover (hot spare)

SQL Server 2005 uses what it calls database mirroring, in order to implement a standby failover. A standby failover is a hot spare database, preferably updated in real-time. The standby can automatically take over servicing of database requests, in the event that the primary (or principal) database should fail.

An SQL Server 2005 database mirror uses transaction log entries to pass changes to a database mirror. Log entries are then applied to the database mirror as reverse log entries.

Note: Transaction log entries in a relational database store all changes to a database. Applying log entries in reverse implies that the database mirror duplicates changes recorded as log entries, onto the database mirror. This is a mirror image database.

The database mirror will not likely be as up to date as an equivalent replicated database would be, but it could be very close. The only entries that could be lost in the event of principal database failure would be any changes on the principal database that are yet to be logged—or any log entries made on the principal database not yet transferred to the database mirror. This potential loss could be quite substantial depending on requirements, but that level of loss could be acceptable depending on the company.

Note: Officially, all committed transactions should be stored on the database mirror. Any pending transactions are not yet written to log files, and thus pending transactions may be lost.

In general, because a database mirror uses a back door to pass log entries onto a not directly connected hot spare server, then the database mirror architecture is technically much easier to implement than replication. And a hot spare is also much faster in terms of transfer and update of the mirrored database. By passing log entries from principal to hot spare database using log files, the transfer occurs within the file system and not the databases. Copying a file within the operating system is much faster than transferring and applying each change one at a time, which is the way that replication does it. The result of application of log entries on the database mirror has no connection to the principal database and is thus not disruptive to the principle database. In some scenarios, a hot spare database can be used as a read only database. This can be useful for getting reporting functionality off the principal database, further reducing demand on a principal and database mirror servers.

11.6.1 Clustered failover databases

The Enterprise and Datacenter editions of Windows 2003 contain clustering technology. Clustering is a very high availability architecture in the form of a clustered or load balanced. Essentially, multiple SQL Servers can reside on each node of the cluster creating a very highly available and scalable failover solution.

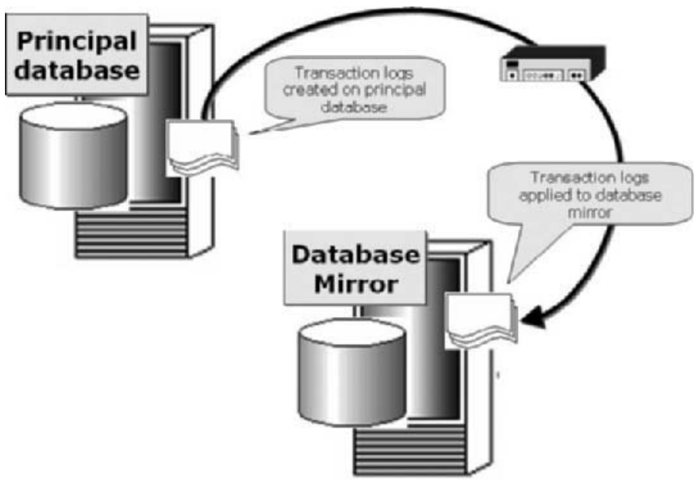

Principal database
Transaction logs created on principal database
Transaction logs applied to database mirror
Database Mirror

Figure 11.4 *Principal to database mirror (hot spare) architecture*

11.7 Flashback snapshot databases

In relational database terminology a snapshot is like a still picture or still-life of a database at a specific point in time. For example, a snapshot of a database taken at midday will look the same an hour later, or the middle of the next week—unless you delete the snapshot. One of the great benefits of snapshots is the possibility of really rapid recovery in the case of failure. In SQL Server 2005, a snapshot is a point in view-like copy of a database. Also, that snapshot is automatically updated making copies of changed pages to the snapshot from the principle database. So, a snapshot also provides for multiple sources of data, much like replication and database mirrors can.

12

Monitoring Performance

As we have mentioned on a number of occasions, physical database design is not a static, one-off process. Once the database has gone into production, the user requirements are likely to change. Even if they do not, the database data is likely to be volatile, and tables are likely to grow. Figure 12.1 shows a typical monitoring and tuning cycle.

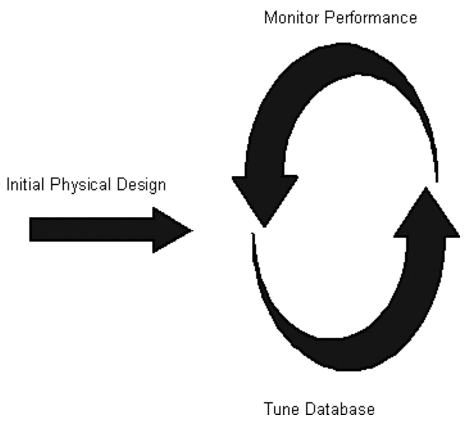


Figure 12.1
The monitoring and tuning cycle

In the previous chapters, we have seen a number of tools that can be used to monitor performance. There are also other tools that have hardly been mentioned. This chapter will look at the array of tools the database administrator can use to monitor SQL Server performance. These tools include the following:

- System stored procedures
- Windows operating system tools: System Monitor, Performance Logs, and Alerts

- SQL Server Management Studio
- SQL Server Profiler
- Database Engine Tuning Advisor
- SQL OS

12.1 System stored procedures

There are a number of system stored procedures that can assist in performance monitoring, including:

```
sp_lock
sp_who
sp_monitor
```

The system stored procedures sp_lock and sp_who provide information on locks, blocked connections, and much more. Both these system stored procedures were described in Chapter 10, so we will concentrate on sp_monitor here.

SQL Server keeps resource use information available through system statistical functions and sp_monitor then formats and displays this information. In fact, it displays the current values of resource use and the difference between these current values and the values last time sp_monitor was run:

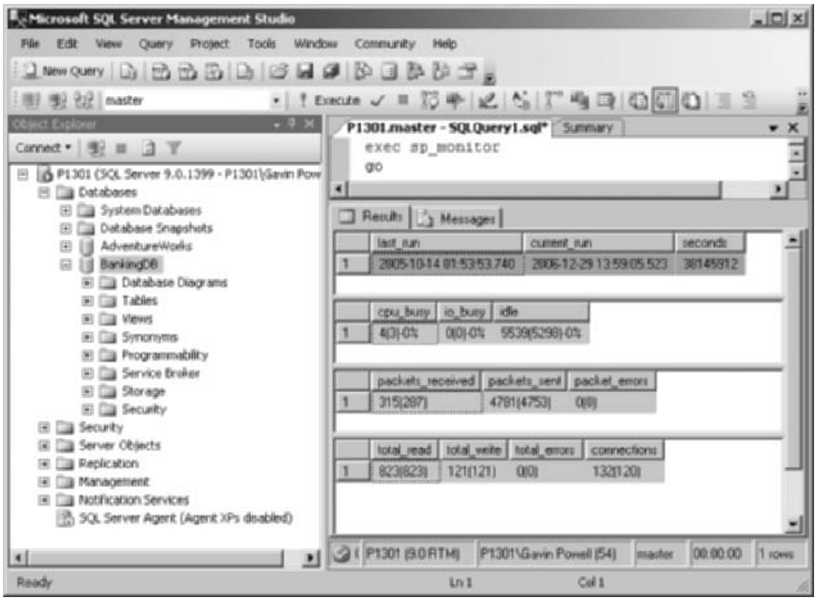
```
EXEC sp_monitor

last_run                    current_run                               seconds
2000-08-17 18:33:25.263                       2000-08-17 18:36:43.500 198

cpu_busy                    io_busy  idle
62(61)-30%                  1(0)-0%  651(130)-65%

packets_received            packets_sent      packet_errors
110(66)                     109(66)           0(0)

total_read                  total_write       total_errorsconnections
432(217)                    69(6)             0(0)  18(2)
```

The cpu_busy, io_busy, and idle values are measured in seconds. The value 62(61)-30% is decoded as 62 seconds of CPU use since SQL Server was started, and (61) is decoded as 61 seconds of CPU use since sp_monitor was last executed. The CPU has been busy 30 percent of the time since sp_monitor was last executed. Similarly, for total_write the value 69(6) can be decoded as 69 writes since SQL Server was started, and (6) is decoded as six writes since sp_monitor was last executed.

These functions are available to be executed by Transact-SQL statements if the database administrator prefers his or her own format. The sp_monitor Transact-SQL definition can easily be examined using the SQL Enterprise Manager.

Many database administrators use their own home-grown stored procedures to interrogate the system tables. Taking this approach means that the output is customized to individual preference and is fine-tuned for the application.

In the Management Studio, the sp_monitor procedure can be executed as a query and would look as shown in Figure 12.2.



```
exec sp_monitor
go
```

	last_run	current_run	seconds
1	2005-10-14 01:53:53.740	2006-12-29 13:59:05.523	38145912

	cpu_busy	io_busy	idle
1	4(3)-0%	0(0)-0%	5539(5298)-0%

	packets_received	packets_sent	packet_errors
1	315(287)	4781(4753)	0(0)

	total_read	total_write	total_errors	connections
1	823(823)	121(121)	0(0)	132(120)

Figure 12.2 *Query results in the Management Studio*

12.2 System monitor, performance logs, and alerts

The System Monitor and Performance Logs and Alerts are provided with the Windows server operating system in order to facilitate performance monitoring through a graphical interface.

There are many objects that can be monitored for the Windows operating system. These objects include the processor object and the memory object, and for each object various counters can be monitored. The processor object has counters such as %Processor Time.

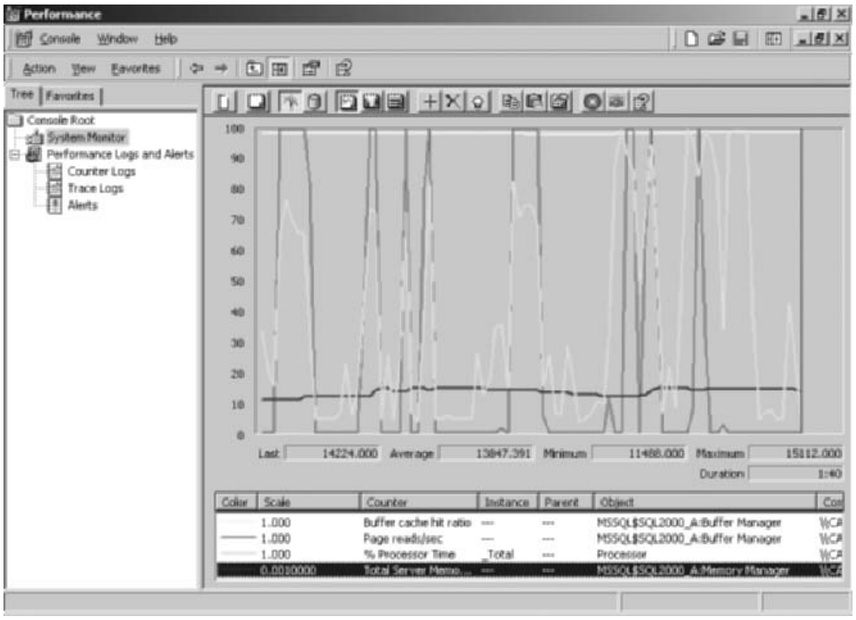
There are special objects for SQL Server, including the following:

```
SQLServer: Access Methods
SQLServer: Backup Device
SQLServer: Buffer Manager
SQLServer: Buffer Partition
SQLServer: Cache Manager
SQLServer: Databases
SQLServer: General Statistics
SQLServer: Latches
SQLServer: Locks
SQLServer: Memory Manager
SQLServer: Replication Agents
SQLServer: Replication Dist.
SQLServer: Replication Logreader
SQLServer: Replication Merge
SQLServer: Replication Snapshot
SQLServer: SQL Statistics
SQLServer: Use Settable Object
```

If multiple instances of SQL Server are being used, the object name is formed from the instance name. For example, the SQL Server instance named SQL_A will use object names such as MSSQL$SQL_A: Locks.

Ensuring that System Monitor is selected in the console pane, click the Add (+) button. This will display drop-down lists of objects and counters and the computers that can be monitored. Monitoring performance will affect performance, so running the System Monitor on a computer other than the server being monitored will reduce its impact on that server.

The SQLServer: Access Methods object has associated counters such as Page Splits/sec, the SQLServer: Buffer Manager object has associated counters such as Buffer cache hit ratio, the SQLServer: Databases object has associated counters such as Percentage Log Used, and the SQLServer: Locks object has associated counters such as Lock Requests/sec. A typical display, showing Buffer cache hit ratio and three other counters, is shown in Figure 12.3.



Last 14224.000 Average 13847.391 Minimum 11488.000 Maximum 15112.000
Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Co
	1.000	Buffer cache hit ratio	---	---	MSSQL$SQL2000_A:Buffer Manager	\\CA
	1.000	Page reads/sec	---	---	MSSQL$SQL2000_A:Buffer Manager	\\CA
	1.000	% Processor Time	_Total	---	Processor	\\CA
	0.0010000	Total Server Memo...	---	---	MSSQL$SQL2000_A:Memory Manager	\\CA

Figure 12.3
The System Monitor chart display

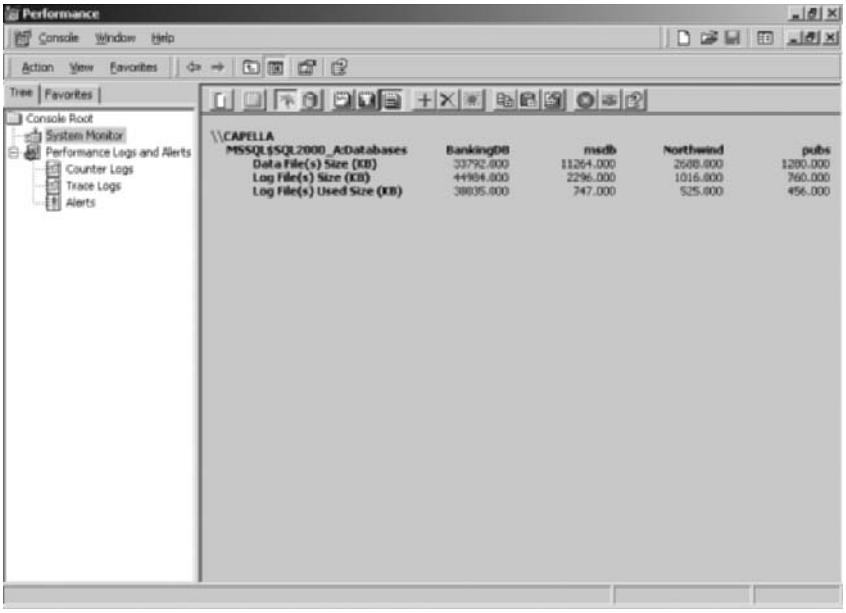
Many counters can be displayed simultaneously, and the display can be changed to a histogram or a report. A report display using SQLServer: Databases counters is shown in Figure 12.4.

Alerts can also be defined via Performance Logs and Alerts. This must be selected and expanded in the console pane. The Alerts folder is right mouse–clicked and New Alert Settings chosen. A counter is selected and a threshold value chosen over (or under) which the alert is signaled. When an alert is signaled, various actions can be taken, such as an entry being logged in the application event log, a program executed, or a network message sent.

Figure 12.5 shows the performance console with two alerts running.

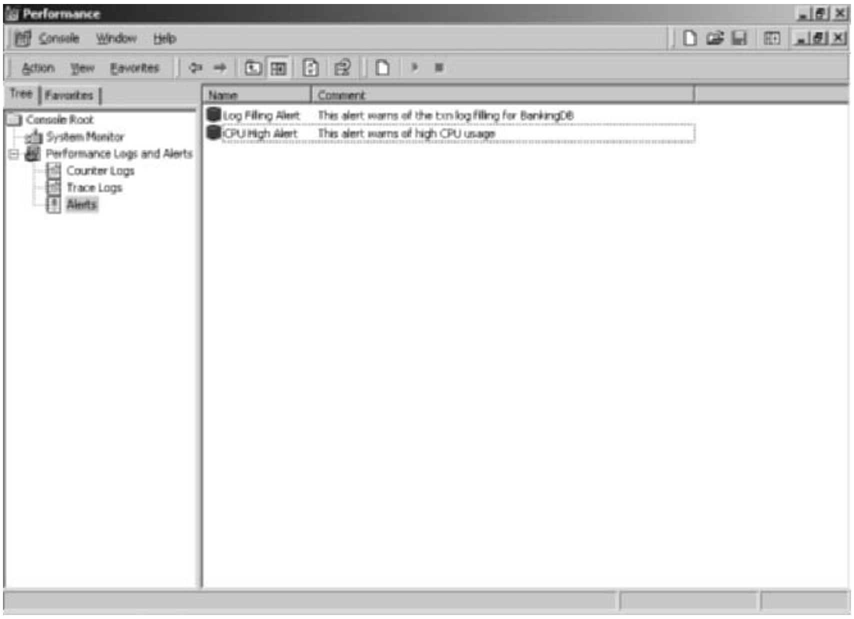
Figure 12.6 shows a network message sent when one of the alerts has been exceeded.

A useful feature is the capability to log counters to a file and then monitor the logged values later. This facility is very useful, since it means that samples can be taken, say, every few minutes, over a period of days. Perfor-



```
\\CAPELLA
  MSSQL$SQL2000_A:Databases        BankingDB        msdb   Northwind       pubs
    Data File(s) Size (KB)         33792.000   11264.000    2688.000   1280.000
    Log File(s) Size (KB)          44984.000    2296.000    1016.000    760.000
    Log File(s) Used Size (KB)     38835.000     747.000     525.000    456.000
```

Figure 12.4 *The System Monitor report display*



Name	Comment
Log Filling Alert	This alert warns of the txn log filling for BankingDB
CPU High Alert	This alert warns of high CPU usage

Figure 12.5 *The alert display*

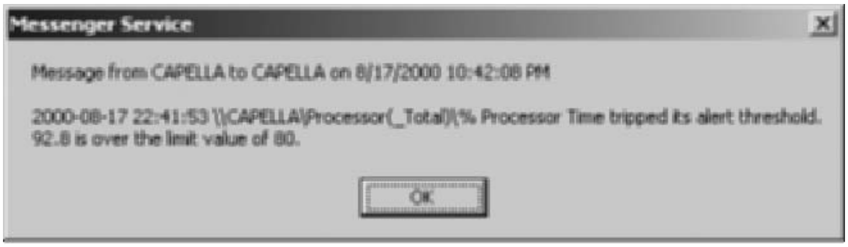


Figure 12.6
A network message resulting from an alert

Messenger Service

Message from CAPELLA to CAPELLA on 8/17/2000 10:42:08 PM

2000-08-17 22:41:53 \\CAPELLA\Processor(_Total)\% Processor Time tripped its alert threshold. 92.8 is over the limit value of 80.

mance monitoring over a long period of time makes it easier to spot trends and sustained bottlenecks. A log is set up via Performance Logs and Alerts.

This must be selected and expanded in the console pane. The Counter Logs folder is right mouse–clicked and New Log Settings chosen.

The System Monitor and Performance Logs and Alerts are key tools for monitoring SQL Server performance, and any SQL Server database administrator should familiarize himself or herself with these tools.

12.3 SQL Server 2005 Management Studio

The SQL Server 2005 Management Studio tool is a GUI, which has been introduced as a method of repackaging all SQL Server tools into a single interface. Some of the GUI tools from SQL Server 2000 are still accessible by themselves. Others are only accessible from within other tools, such as the Management Studio. The general trend is toward centralizing and ultimately easing the tasks of administration, maintenance, and performance tuning of SQL Server environments.

12.3.1 Client statistics

Query optimization has already been examined extensively in previous chapters but in the context of viewing estimated query execution plans. This section takes a brief look into client statistics. Client statistics can be gathered and examined for a query executed, in the Management Studio, by selecting the Include Client Statistics tab at the top of the window.

Note: This functionality is known as Show Server Trace and Show Client Statistics in SQL Server 2000.

The Include Client Statistics tab shows client-side information about the execution of a query. An example is shown in Figure 12.7.

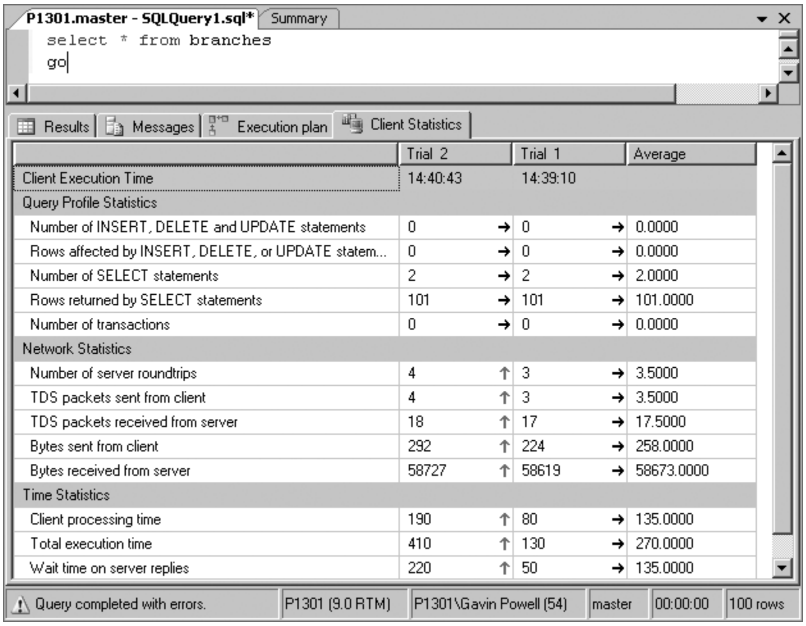


Figure 12.7
The Include Client Statistics tab

```
select * from branches
go
```

	Trial 2		Trial 1		Average
Client Execution Time	14:40:43		14:39:10		
Query Profile Statistics					
Number of INSERT, DELETE and UPDATE statements	0	→	0	→	0.0000
Rows affected by INSERT, DELETE, or UPDATE statem...	0	→	0	→	0.0000
Number of SELECT statements	2	→	2	→	2.0000
Rows returned by SELECT statements	101	→	101	→	101.0000
Number of transactions	0	→	0	→	0.0000
Network Statistics					
Number of server roundtrips	4	↑	3	→	3.5000
TDS packets sent from client	4	↑	3	→	3.5000
TDS packets received from server	18	↑	17	→	17.5000
Bytes sent from client	292	↑	224	→	258.0000
Bytes received from server	58727	↑	58619	→	58673.0000
Time Statistics					
Client processing time	190	↑	80	→	135.0000
Total execution time	410	↑	130	→	270.0000
Wait time on server replies	220	↑	50	→	135.0000

```
Query completed with errors.   P1301 (9.0 RTM)   P1301\Gavin Powell (54)   master   00:00:00   100 rows
```

The client statistics are grouped into three areas, as follows:

- Query Profile Statistics—containing information such as the number of SELECT statements.
- Network Statistics—containing information such as the number of server roundtrips.
- Time Statistics—containing information such as the cumulative client processing time.

12.3.2 The SQL Server Profiler

The SQL Server Profiler is probably one of the most useful tools for performance investigation. It allows the database administrator to trace the events that are happening on an SQL Server.

Note: In SQL Server 2005 you can even replay a sequence of events without changing anything in the database. The objective of replay is to isolate a previously occurring problem. The replay facility is very useful when trying to track down intermittent problems. Intermittent problems are often difficult to duplicate because they do not repeat consistently in a given scenario.

One or more traces are defined that are designed to capture a set of events. The trace definition will also specify what information is to be captured concerning the events, and what filtering criteria are to be used. It may be that you only wish to capture events for a particular database, or for events that exceed a minimum duration.

The information captured by the trace can be displayed graphically and can also be written to a file and/or a database table. This allows the traced data to be analyzed later.

12.3.2.1 What events can be traced?

There are many events that can be traced. These are known as event classes. Related event classes are grouped into event categories. For example, the Lock:Acquired and Lock:Timeout event classes are grouped together into the Locks event category. Event categories are shown in Table 12.1.

Table 12.1 *SQL Server Profiler event categories*

Event Category	Definition
Broker	Service Broker events
CLR	.NET Framework objects
Cursors	Event classes concerned with cursors
Database	Event classes concerned with data and log file growth and shrinkage
Deprecation	Indicates future removal of features from SQL Server software
Errors and Warnings	Event classes concerned with errors, warnings, and writes to error logs
Full-Text	Full-text search events
Locks	Event classes concerned with locks
Objects	Event classes concerned with an object being opened, closed, created, and deleted, as well as the execution of autostats
OLEDB	OLEDB call events
Performance	Event classes concerned with query plans, parallelism, and DML commands
Progress Report	Progress of online index rebuilds
Scans	Event classes concerned with table and index scans

Table 12.1 *SQL Server Profiler event categories (continued)*

Security Audit	Event classes concerned with security operations; logins/logouts; and server starts, stops, and pauses
Server	Event classes concerned with server events, such as memory changes
Sessions	Event classes concerned with connects and disconnects
Stored procedures	Event classes concerned with stored procedures
Transactions	Event classes concerned with transactions starting and stopping—includes MS DTC transactions and writes to the transaction log
TSQL	Event classes concerned with SQL statements and batches
User Configurable	Event classes concerned with user-defined events created with the stored procedure sp_trace_generateevent

Some event classes are very useful and are often traced, while some event classes are more obscure. You will often find that the traces you wish to create will involve the same event classes. For this reason, as we shall see, templates can be created containing your common event classes that can then form the basis of your traces.

12.3.2.2 What information is collected?

Before looking at specific event classes, let us look at the information that can be collected about them and how they are filtered. The elements of information that can be collected are known as Data Columns, and there are over 40 of them. Some data columns are not relevant for an event class. For example, the Reads data column is not relevant for the Lock:Aquired event class. Generally speaking, though, many data columns are relevant for most event classes.

Some data columns contain information whose definition remains the same regardless of the event class being traced. A data column such as CPU, which holds the amount of CPU time (in milliseconds) used by the event, always holds this value for any event that CPU is relevant for. On the other hand, data columns such as Binary Data, Integer Data, and TextData hold values that are dependent on the event class captured in the trace. For example, the Errorlog event class, which occurs when error events have been logged in the SQL Server error log, causes the Text data column to hold the text of the error message. On the other hand, the Missing Column Statistics event class, which occurs when column statistics that could be used by the query optimizer are not available, causes the Text data column to hold the list of the columns with missing statistics.

When defining a trace, the data columns can be grouped. Grouping overrides the default behavior in the graphical interface of the SQL Server Profiler by displaying events in the order that they occur. For example, grouping the events by Application Name groups together all the events for an application.

12.3.2.3 Filtering information

In order to reduce the volume of information traced, it can be filtered. Filtering can also reduce the impact of the trace on the server. You will need to take care, however, that what you choose to filter out of the trace is not a participant in the situation you are trying to observe. It may be that filtering out events whose duration is less than one second will help you see the wood for the trees, but if an SQL:StmtCompleted event takes just less than a second but is being executed thousands of times, it may be the culprit behind a performance problem.

Most, but not all, data columns can have filters defined for them. We can create a filter that includes applications with a filter that specifies LIKE MyProc% or NOT LIKE MS EM%. The % symbol represents a wildcard character, which can substitute for zero or more characters (just the same as LIKE in Transact-SQL). We might specify that we only wish to trace events with Duration greater than or equal to 1,000 or DatabaseID = 12.

12.3.2.4 Creating an SQL Server profiler trace

Now that we have introduced the basic concepts behind an SQL Server Profiler trace, we can create one. Let us start by creating a trace to capture events whose duration is greater than or equal to one-hundredth of a second. This will filter out very short-lived events. Let us assume we are interested in looking for rogue Transact-SQL statements.

Launch the SQL Server Profiler from the Start menu, the Management Studio, or even from within the Database Engine Tuning Advisors. The tools are all closely linked together in SQL Server 2005. You will be faced with a fairly blank window, as shown in Figure 12.8.

We can then select File and New Trace, click on the New Trace button—or just type CTRL+N. Having responded to the connection prompt with appropriate security credentials, the SQL Server Profiler displays the Trace Properties window, as shown in Figure 12.9.

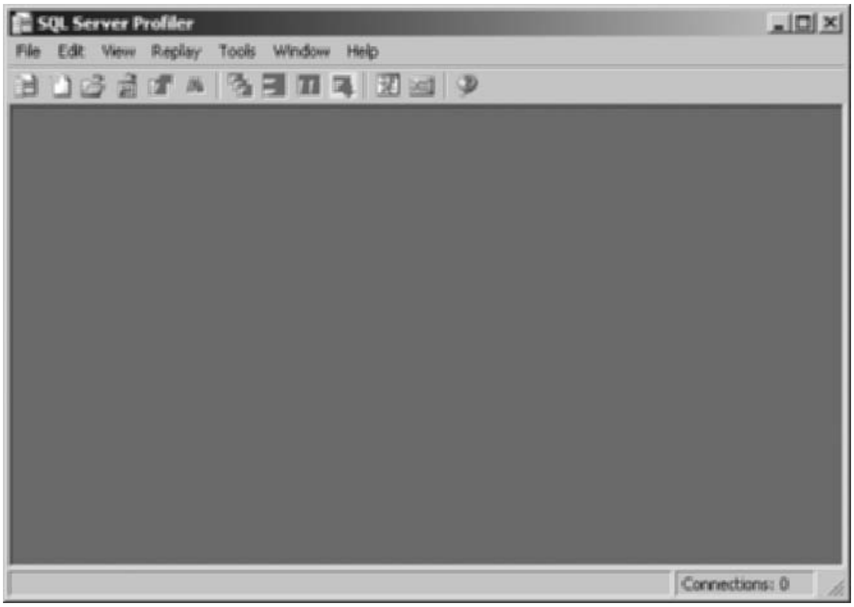
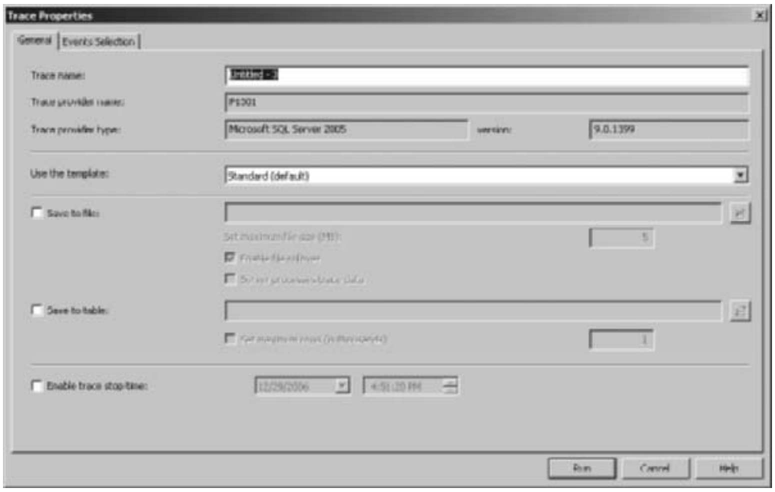


Figure 12.8
The SQL Server Profiler initial window



Trace name: Untitled - 1
Trace provider name: PS301
Trace provider type: Microsoft SQL Server 2005 version: 9.0.1399
Use the template: Standard (default)
☐ Save to file:
Set maximum file size (MB): 5
☑ Enable file rollover
☐ Server processes trace data
☐ Save to table:
☐ Set maximum rows (in thousands): 1
☐ Enable trace stop time: 12/29/2006 4:51:20 PM

Figure 12.9
The SQL Server Profiler trace properties window

First of all, the trace is named and the SQL Server or SQL Server instance that is to be traced selected. A trace template is then selected. A trace template contains a predefined set of event classes and data columns. These are used as a convenience when creating new traces. Their event classes and data columns can be added to or removed, and the resulting template can be saved under a new name if desired. Apart from Blank, there are a number of template names to choose from. We will choose the SQLServerProfilerStandard template, since this fits our needs quite well.

Next, we must specify where we are going to save trace information, if at all. The information will always be displayed in the SQL Server Profiler graphical interface, but we also have the choice of saving the information in a file or database table, or both. Microsoft suggests that saving data to a file is faster than saving data to a database table. Analyzing data in a table, though, is much easier. To have the best of both worlds, save the trace information to a file and then afterwards open the trace file and save it as a trace table.

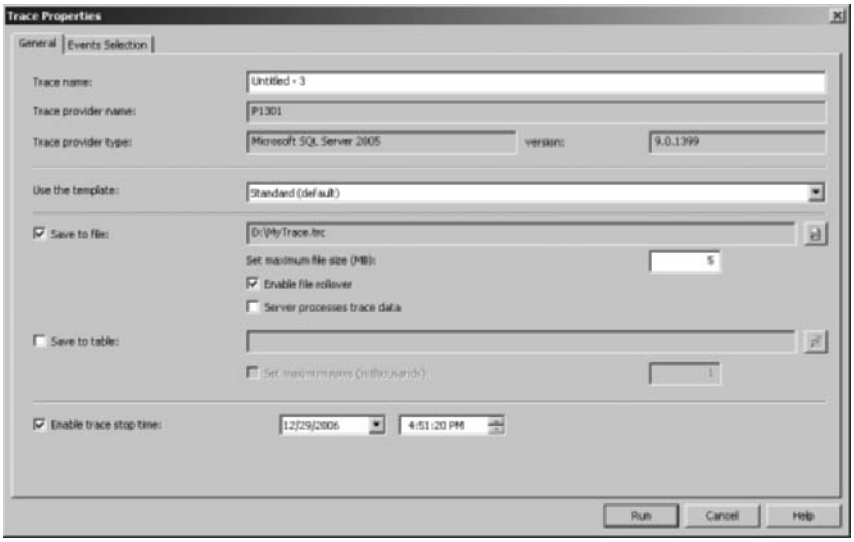
If Save to file is checked, the SQL Server Profiler will prompt for a location and filename. This SQL Server Profiler trace file will have an extension of .trc. A maximum file size (MB) may be optionally specified. A trace whose maximum file size has been specified finishes saving trace information to the file after the maximum file size has been reached. Another option, Enable file rollover, may be checked if the Set maximum file size (MB) is checked. With this option set, when the original file reaches the maximum size, a second file is opened and trace data is written to it. When the second file reaches the maximum size, a third file is opened and so on. The SQL Server Profiler adopts a simple strategy for the filenames. It merely appends an integer to the original filename. The filename MyTrace.trc becomes MyTrace_1.trc, then MyTrace_2.trc, and so on.

The Server processes SQL Server trace data option may be checked if the server running the trace is to process the trace data rather than the client. Selecting this option may adversely affect the performance of the server being traced, since it ensures that no events are skipped—even when the server is overloaded.

As well as, or instead of, capturing trace information to a file, it can also be captured in a table. The table can be present on any SQL Server, and, by default, it takes the name of the trace. The maximum number of rows to capture can be set, after which no more trace information is stored in the table.

Finally, a stop time can be set. Once this time is reached, the trace will stop and close itself. Figure 12.10 shows an example of the General tab of the SQL Server Profiler Trace Properties window.

Next, the event classes that are to be traced must be specified. The event classes are chosen in the Events tab of the SQL Server Profiler Trace Properties window. The default configuration is shown in Figure 12.11.


Trace Properties

General | Events Selection

Trace name: Untitled - 3
Trace provider name: P1301
Trace provider type: Microsoft SQL Server 2005 version: 9.0.1399

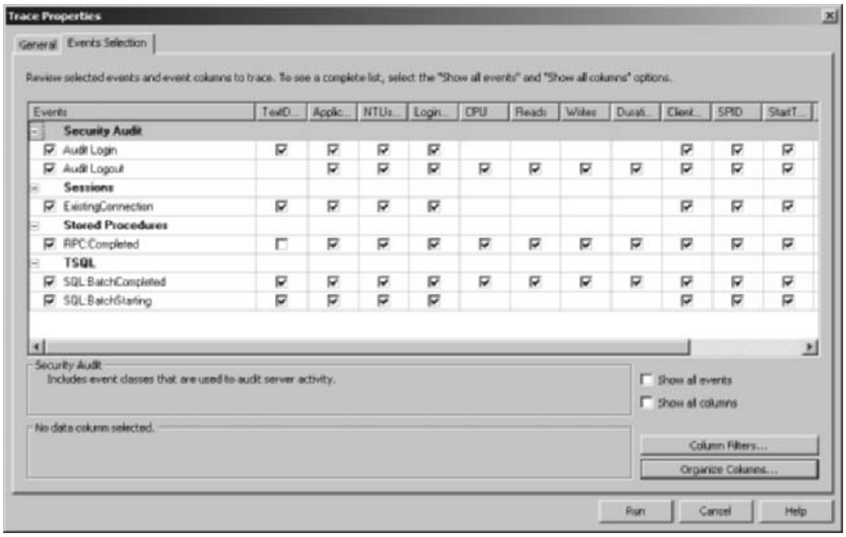
Use the template: Standard (default)

[✓] Save to file: D:\MyTrace.trc
Set maximum file size (MB): 5
[✓] Enable file rollover
[ ] Server processes trace data

[ ] Save to table:
[ ] Set maximum rows (in thousands): 1

[✓] Enable trace stop time: 12/29/2006 4:51:20 PM

Figure 12.10
The SQL Server Profiler trace properties General tab


Trace Properties

General | Events Selection

Review selected events and event columns to trace. To see a complete list, select the "Show all events" and "Show all columns" options.

Events	TextD...	Applic...	NTUs...	Login...	CPU	Reads	Writes	Durati...	Client...	SPID	StartT...
Security Audit											
✓ Audit Login	✓	✓	✓	✓					✓	✓	✓
✓ Audit Logout		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Sessions											
✓ ExistingConnection	✓	✓	✓	✓					✓	✓	✓
Stored Procedures											
✓ RPC:Completed	☐	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
TSQL											
✓ SQL:BatchCompleted	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
✓ SQL:BatchStarting	✓	✓	✓	✓					✓	✓	✓

Security Audit
Includes event classes that are used to audit server activity.

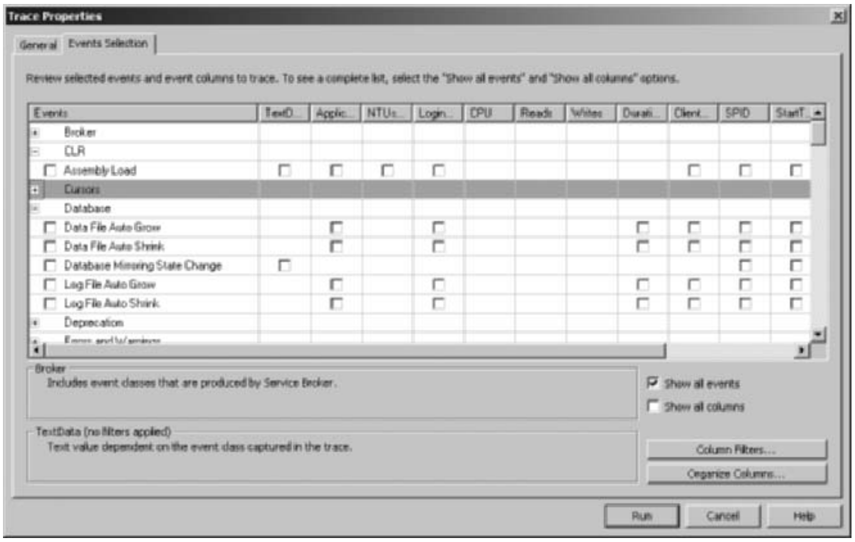
No data column selected.

[ ] Show all events
[ ] Show all columns

Figure 12.11
The SQL Server Profiler trace properties Events tab

Now check the Show All Events checkbox and the screen shows a more familiar event class category set of choices.

As shown in Figure 12.12, to begin with, nothing is selected by default. Events categories and classes are initially set based on the template selected under the General tab, as shown in Figure 12.11. So, click the General tab, and select Tuning from the list of templates. And then click the Events Selection tab again. Check Show All Events again. Open up the Perfor-



Trace Properties

General | Events Selection

Review selected events and event columns to trace. To see a complete list, select the "Show all events" and "Show all columns" options.

Events	TextD...	Applic...	NTUs...	Login...	CPU	Reads	Writes	Durati...	Client...	SPID	StartT...
+ Broker											
− CLR											
☐ Assembly Load	☐	☐	☐	☐					☐	☐	☐
+ Cursors											
− Database											
☐ Data File Auto Grow		☐		☐				☐	☐	☐	☐
☐ Data File Auto Shrink		☐		☐				☐	☐	☐	☐
☐ Database Mirroring State Change	☐									☐	☐
☐ Log File Auto Grow		☐		☐				☐	☐	☐	☐
☐ Log File Auto Shrink		☐		☐				☐	☐	☐	☐
+ Deprecation											

Broker
Includes event classes that are produced by Service Broker.

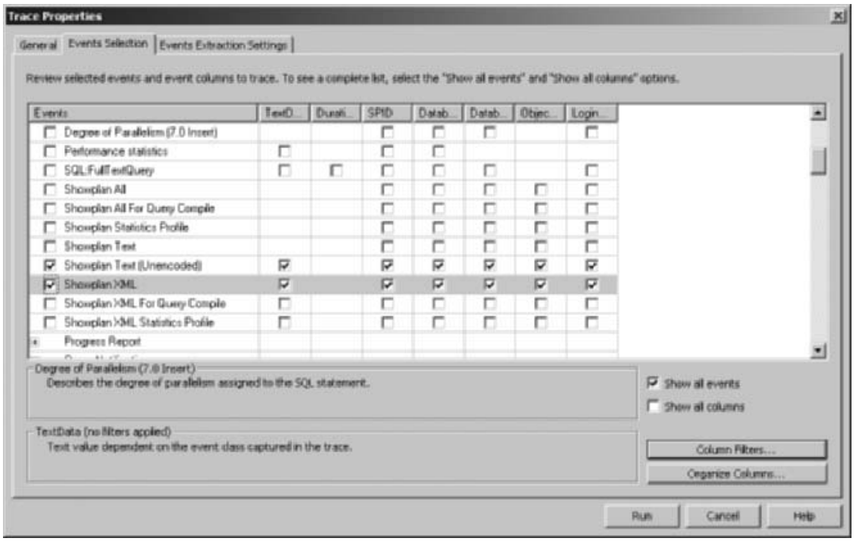
☑ Show all events
☐ Show all columns

TextData (no filters applied)
Text value dependent on the event class captured in the trace.

Figure 12.12
SQL Server Profiler events categories

mance category by clicking the little plus sign next to it. Scroll down and check the Showplan Text (Unencoded) and Showplan XML event classes.

Finally, we can specify a filter by clicking the Column Filters button as shown in Figure 12.13.



Trace Properties

General | Events Selection | Events Extraction Settings

Review selected events and event columns to trace. To see a complete list, select the "Show all events" and "Show all columns" options.

Events	TextD...	Durati...	SPID	Datab...	Datab...	Objec...	Login...
☐ Degree of Parallelism (7.0 Insert)			☐	☐	☐		☐
☐ Performance statistics	☐		☐	☐			
☐ SQL:FullTextQuery	☐	☐	☐	☐	☐		☐
☐ Showplan All			☐	☐	☐	☐	☐
☐ Showplan All For Query Compile			☐	☐	☐	☐	☐
☐ Showplan Statistics Profile			☐	☐	☐	☐	☐
☐ Showplan Text			☐	☐	☐	☐	☐
☑ Showplan Text (Unencoded)	☑		☑	☑	☑	☑	☑
☑ Showplan XML	☑		☑	☑	☑	☑	☑
☐ Showplan XML For Query Compile	☐		☐	☐	☐	☐	☐
☐ Showplan XML Statistics Profile	☐		☐	☐	☐	☐	☐
+ Progress Report							

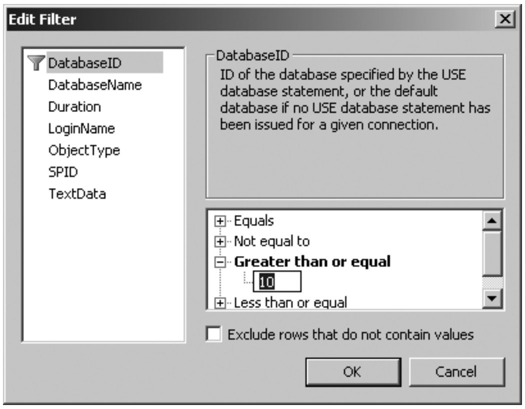
Degree of Parallelism (7.0 Insert)
Describes the degree of parallelism assigned to the SQL statement.

☑ Show all events
☐ Show all columns

TextData (no filters applied)
Text value dependent on the event class captured in the trace.

Figure 12.13
SQL Server Profiler column filtering selection

Now select Greater than or equal, enter a value of 10, as shown in Figure 12.14, and click the OK button.



Edit Filter

DatabaseID
DatabaseName
Duration
LoginName
ObjectType
SPID
TextData

DatabaseID
ID of the database specified by the USE database statement, or the default database if no USE database statement has been issued for a given connection.

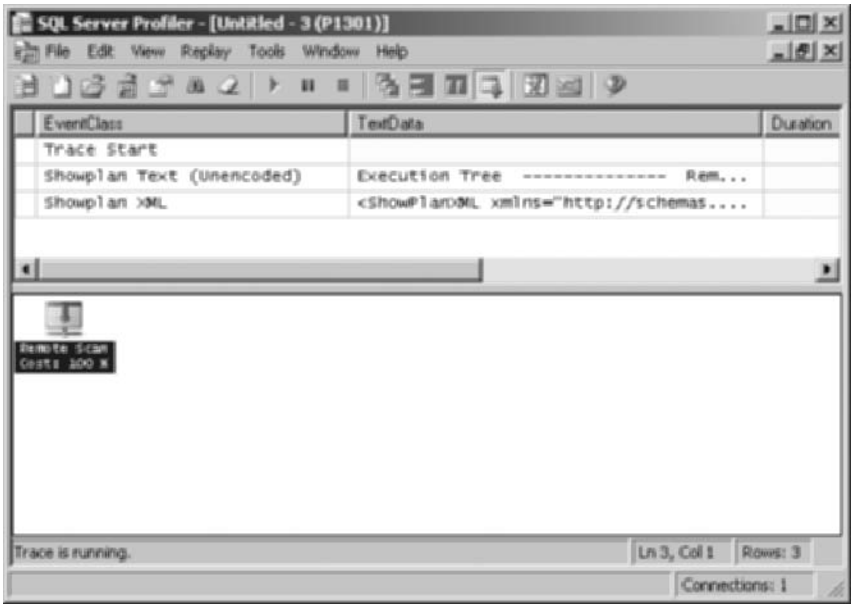
Equals
Not equal to
Greater than or equal
10
Less than or equal

Exclude rows that do not contain values

OK Cancel

Figure 12.14
SQL Server Profiler column filtering settings

We have decided only to include events of duration greater or equal to one-hundredth of a second. Now all we have to do is click the run button and our trace will start. The trace window looks as shown in Figure 12.15.



SQL Server Profiler - [Untitled - 3 (P1301)]

File Edit View Replay Tools Window Help

EventClass	TextData	Duration
Trace Start		
Showplan Text (Unencoded)	Execution Tree -------------- Rem...	
Showplan XML	<ShowPlanXML xmlns="http://schemas....	

Remote Scan
Cost: 100 %

Trace is running. Ln 3, Col 1 Rows: 3

Connections: 1

Figure 12.15
SQL Server Profiler column filtering settings

In reality, as a database administrator, you will probably be so busy that sitting and watching an SQL Server Profiler trace graphically will not be the best use of your time. It is often more convenient and productive to analyze the trace output that you have captured into a table. For example, another trace was run and also captured in the database table MyTrace, in a database called Perf-StatsDB.

Suppose we execute the following query:

```
SELECT TextData, Duration, CPU FROM MyTrace where Duration > 1000
```

Sample output would be as follows:

```
TextData                                                Duration    CPU

SELECT * FROM accounts WHERE balance = 100              1693        40
SELECT * FROM customers WHERE customer_no = 1000        1540        40
```

By using familiar Transact-SQL statements, the trace data can be analyzed to look for problem statements. As well as responding to problems, traces can be run on a regular basis and the trace data analyzed to monitor trends. The Transact-SQL functions AVG, MIN, and MAX are useful, and the data can be grouped by the first few characters of the TextData column so that the statements are distinguished:

```
Statement           AverageDuration     MaxDuration     AverageCPU     MaxCPU

exec USP_CustBal    33.333333           40              33.333333      40
```

This line of output was generated by the following Transact-SQL statement.

```
SELECT
CAST(TextData AS CHAR(16)) AS Statement,
AVG(Duration) AS AverageDuration,
MAX(Duration) AS MaxDuration,
 AVG(CPU) AS AverageCPU,
MAX(CPU) AS MaxCPU
 FROM MyTrace4
WHERE Duration > 10
 GROUP BY CAST(TextData AS CHAR(16))
```

The GROUP BY uses as CAST of CHAR(16) to group only by the stored procedure name and does not include any parameters. Of course, the other aggregate functions, such as COUNT, can be used. It is also useful to filter out the stored procedures and statements you are interested in with a LIKE operator in the WHERE clause.

12.3.2.5 Creating traces with stored procedures

As well as using the SQL Server Profiler graphical interface to create, modify, start, and stop traces, various system stored procedures can also be used. These are all documented, but the easiest way to create a script that utilizes them is to create a trace using the SQL Server Profiler graphical interface and then from the File menu in the graphical interface choose Script Trace. The trace can be scripted using the SQL Server 2005 system stored procedures. The script produced can then be edited and executed using the query tools in the Management Studio.

There are only a few system stored procedures that need to be used when creating and managing a trace. The ones we will use are as follows:

```
sp_trace_create
sp_trace_setevent
sp_trace_setfilter
sp_trace_setstatus
```

The system stored procedure sp_trace_create is typically run first to create the trace. Information such as the stop time, trace file name, maximum file size, and whether file rollover is performed can be specified. This system stored procedure returns an integer trace ID, which is subsequently used to identify the trace.

The system stored procedure sp_trace_setevent is used to add or remove an event or event column to a trace. The event ID and column ID pair is specified and is either turned on or off. The trace that is to be modified is identified through the trace ID.

The system stored procedure sp_trace_setfilter is used to specify filters. The trace that is to be modified is identified through the trace ID. A column is specified together with a value specifying whether it will be ANDed or ORed with other filter conditions. A value to represent a comparison operator, such as Greater Than, is specified for the column together with the value to be compared. Finally, the system stored procedure sp_trace_setstatus is used to stop and start the event. Again, the trace that is to be started or stopped is identified through the trace ID.

Here is a trace script generated by the SQL Server Profiler:

```
-- Create a Queue
DECLARE @rc int
DECLARE @TraceID INT
DECLARE @maxfilesize BIGINT
SET @maxfilesize = 5344176266805258
EXEC @rc = sp_trace_create @TraceID OUTPUT, 2, N'C:\
MyTrace.trc', @maxfilesize, NULL
IF (@rc != 0) GOTO error

-- Client side File and Table cannot be scripted

-- Set the events
DECLARE @on BIT
SET @on = 1
EXEC sp_trace_setevent @TraceID, 10, 1, @on
EXEC sp_trace_setevent @TraceID, 10, 6, @on
EXEC sp_trace_setevent @TraceID, 10, 9, @on
EXEC sp_trace_setevent @TraceID, 10, 10, @on
EXEC sp_trace_setevent @TraceID, 10, 11, @on
EXEC sp_trace_setevent @TraceID, 10, 12, @on
EXEC sp_trace_setevent @TraceID, 10, 13, @on
EXEC sp_trace_setevent @TraceID, 10, 14, @on
EXEC sp_trace_setevent @TraceID, 10, 16, @on
EXEC sp_trace_setevent @TraceID, 10, 17, @on
EXEC sp_trace_setevent @TraceID, 10, 18, @on
EXEC sp_trace_setevent @TraceID, 12, 1, @on
EXEC sp_trace_setevent @TraceID, 12, 6, @on
EXEC sp_trace_setevent @TraceID, 12, 9, @on
EXEC sp_trace_setevent @TraceID, 12, 10, @on
EXEC sp_trace_setevent @TraceID, 12, 11, @on
EXEC sp_trace_setevent @TraceID, 12, 12, @on
EXEC sp_trace_setevent @TraceID, 12, 13, @on
EXEC sp_trace_setevent @TraceID, 12, 14, @on
EXEC sp_trace_setevent @TraceID, 12, 16, @on

EXEC sp_trace_setevent @TraceID, 12, 17, @on
EXEC sp_trace_setevent @TraceID, 12, 18, @on
EXEC sp_trace_setevent @TraceID, 14, 1, @on
EXEC sp_trace_setevent @TraceID, 14, 6, @on
EXEC sp_trace_setevent @TraceID, 14, 9, @on
EXEC sp_trace_setevent @TraceID, 14, 10, @on
EXEC sp_trace_setevent @TraceID, 14, 11, @on
EXEC sp_trace_setevent @TraceID, 14, 12, @on
EXEC sp_trace_setevent @TraceID, 14, 13, @on
EXEC sp_trace_setevent @TraceID, 14, 14, @on
EXEC sp_trace_setevent @TraceID, 14, 16, @on
EXEC sp_trace_setevent @TraceID, 14, 17, @on
EXEC sp_trace_setevent @TraceID, 14, 18, @on
EXEC sp_trace_setevent @TraceID, 15, 1, @on
EXEC sp_trace_setevent @TraceID, 15, 6, @on
EXEC sp_trace_setevent @TraceID, 15, 9, @on
EXEC sp_trace_setevent @TraceID, 15, 10, @on
EXEC sp_trace_setevent @TraceID, 15, 11, @on
EXEC sp_trace_setevent @TraceID, 15, 12, @on
EXEC sp_trace_setevent @TraceID, 15, 13, @on
EXEC sp_trace_setevent @TraceID, 15, 14, @on
EXEC sp_trace_setevent @TraceID, 15, 16, @on
EXEC sp_trace_setevent @TraceID, 15, 17, @on
EXEC sp_trace_setevent @TraceID, 15, 18, @on
EXEC sp_trace_setevent @TraceID, 17, 1, @on
EXEC sp_trace_setevent @TraceID, 17, 6, @on
EXEC sp_trace_setevent @TraceID, 17, 9, @on
EXEC sp_trace_setevent @TraceID, 17, 10, @on
EXEC sp_trace_setevent @TraceID, 17, 11, @on
EXEC sp_trace_setevent @TraceID, 17, 12, @on
EXEC sp_trace_setevent @TraceID, 17, 13, @on
EXEC sp_trace_setevent @TraceID, 17, 14, @on
EXEC sp_trace_setevent @TraceID, 17, 16, @on
EXEC sp_trace_setevent @TraceID, 17, 17, @on
EXEC sp_trace_setevent @TraceID, 17, 18, @on

-- Set the Filters
DECLARE @intfilter INT
DECLARE @bigintfilter BIGINT

EXEC sp_trace_setfilter @TraceID, 10, 0, 7, N'SQL Server
Profiler%'
SET @intfilter = 100
EXEC sp_trace_setfilter @TraceID, 13, 0, 4, @intfilter

EXEC sp_trace_setfilter @TraceID, 35, 1, 6, N'BankingDB'

-- Set the trace status to start
EXEC sp_trace_setstatus @TraceID, 1

error:
GO
```

This trace creates a trace file, C:\MyTrace.trc, with file rollover (option value 2). There is no stop time (NULL), and the maximum file size possible is set.

Event IDs 10, 12, 14, 15, and 17 are set. These are RPC:Completed, SQL:BatchCompleted, Login, Logout, and ExistingConnection, respectively. The sp_trace_setevent stored procedure sets each required event ID and column ID pair. Therefore, we see examples such as the following:

```
EXEC sp_trace_setevent @TraceID, 12, 13, @on
```

This sets event ID 12 (SQL:BatchCompleted) with column ID 13 (Duration) on.

Filters are set to specify that the database must be BankingDB, the duration is greater than 100 milliseconds, and the application is not the SQL Server Profiler itself.

Finally, the trace is set to status value 1, which means start. A status value of 0 means stop. To subsequently view the trace file with the profiler, it is necessary to first stop the trace with status value 0 and then close it with status value 2.

To view information about current traces a useful function is ::fn_trace_getinfo. This takes a trace ID as an argument. Specifying NULL returns information for all existing traces. For example:

```
SELECT * FROM ::fn_trace_getinfo(NULL)

traceid        property       value

1              1              2
1              2              C:\DocumentsandSettings
                              \Administrator\MyDocuments
                              \MyTrace11.trc
1              3              5344176266805258
1              4              NULL
1              5              1
```

The property value 1 is the trace option value to sp_trace_create. In our example, 2 means file rollover is enabled. The property value 2 is the trace file name, and 3 is the maximum file size. The property value 4 is the stop time, and 5 is the current trace status, as set in sp_trace_setstatus. In our example, no stop time is specified. The trace status 1 means the trace is started.

The SQL Server Profiler is a very powerful tool, and I would urge database administrators to familiarize themselves with it. It has many other capabilities, which we will not cover here, but it can, for example, replay a trace file, which is useful for regression and stress testing. It is also able to single step through a trace file, similar to a debugger. Also, a workload saved by the SQL Server Profiler can be used in the Database Engine Tuning wizard, described next.

12.3.3 Database Engine Tuning Advisor

Much like the Management Studio, the Database Engine Tuning Advisor is partly new features and also partly SQL Server 2000 features bundled into a much better interface. The Database Engine Tuning Advisor tool was presented in Chapter 1 as a new feature of SQL Server 2005. So, reiterating at this point would simply be repetitious. Essentially, the Database Engine Tuning Advisor can make recommendations about indexes, partitioning, and underlying physical design structures.

The Database Engine Tuning Advisor can make suggestions about changes, such as the most effective indexes that could be created on a table (or view), based on a workload previously captured by the SQL Server Profiler. Of course, recommendations assume that the workload is representative, and so the onus is on the database administrator to ensure that this is the case. I personally use automated index tuning to get a second opinion on my index design, rather than as a tool that produces a definitive index design.

In an example workload, the Tuning Advisor recommended that three indexes, if created, should improve performance based on the workload. It predicted an 83 percent improvement.

The indexes recommended were as follows:

- A clustered index on the Accounts table on columns balance.
- A non-clustered index on the Accounts table on columns balance, account_no
- A non-clustered index on the Accounts table on column customer_no
- A clustered index on the Customers table on column customer_no

This report showed the percentage of queries in the workload that would make use of the new index and the estimated size of the new index.

Once again, recommendations are based on the workload. That workload has to be representative of daily operations. Also, recommendations of using clustered indexes, as opposed to non-clustered index—on primary keys, should be verified with manual testing. It really depends on how your application accesses data.

12.4 SQL OS and resource consumption

SQL Server 2005 contains many newly introduced metrics. Many are for measuring the database internal performance, and resource usage (memory, locks, schedules, transactions, network, I/O). The SQL OS accesses underlying resource details using a set of what are called Dynamic Management Views (or DMVs).

SQL OS attempts to fill the statistical gap, perhaps more apparent in previous versions of SQL Server. SQL OS creates a virtual performance monitoring layer placed between SQL Server and the underlying Windows operating system. Table 12.2 shows a list of available DMVs and their respective functions.

Table 12.2 *SQL OS Dynamic Management Views*

DMV	Statistics
`sys.dm_os_buffer_descriptors`	Buffer pools
`sys.dm_os_cluster_nodes`	Windows cluster nodes
`sys.dm_os_hosts`	Host details
`sys.dm_os_latch_stats`	Latch statistics (buffer pool locks)
`sys.dm_os_loaded_modules`	Loaded Dynamic Link Libraries (DLLs)
`sys.dm_os_memory_cache_clock_hands`	Cached objects
`sys.dm_os_memory_cache_counters`	Size of cached objects
`sys.dm_os_memory_cache_entries`	Cached object entry specifics
`sys.dm_os_memory_cache_hash_tables`	Cached object soft and hard parsing counts
`sys.dm_os_memory_clerks`	Memory clerks for SQL Server services and processes
`sys.dm_os_memory_objects`	Memory objects

Table 12.2 *SQL OS Dynamic Management Views (continued)*

`sys.dm_os_memory_pools`	Memory pools
`sys.dm_os_performance_counters`	Performance monitor counters
`sys.dm_os_scheduler`	Memory to CPU schedule mapping
`sys.dm_os_stacks`	SQL Server process call stack
`sys.dm_os_sys_info`	Windows operating system information
`sys.dm_os_tasks`	Windows session, scheduling and request tasks
`sys.dm_os_threads`	SQL Server thread use
`sys.dm_os_virtual_address_dump`	Virtual memory pages
`sys.dm_os_wait_stats`	Wait statistics
`sys.dm_os_waiting_tasks`	Tasks waiting for something else to complete
`sys.dm_os_workers`	Thread worker information

We have looked at a number of monitoring tools in this chapter. I find the SQL Server Profiler particularly useful when hunting for poorly performing queries. The Query Analyzer is then really useful for analyzing the problem query to check on the query plan. As an initial step, the System Monitor is very useful for getting an overall feel for the system.

A

Syntax Conventions

Syntax diagrams in this book will utilize what is known as Backus-Naur Form syntax notation convention. Backus-Naur Form has become the de facto standard for most computer texts. SQL is used to describe the notation.

- **Angle brackets: < … >**. Angle brackets are used to represent names of categories (substitution variable representation). In this example <table> will be replaced with a table name in a schema as shown.

  ```
  SELECT * FROM <table>;
  ```

  Becomes:

  ```
  SELECT * FROM ARTIST;
  ```

Note: Angle brackets are generally not used in this book unless stated as such at the beginning of a chapter.

- OR: |. A pipe or | character represents an OR conjunction meaning either can be selected. The asterisk (*) and curly braces are explained further on. In this case all or some columns can be retrieved, some meaning one or more.

  ```
  SELECT { * | { <column>, … } } FROM <table>;
  ```

- Optional: […]. In a SELECT statement a WHERE clause is syntactically optional.

  ```
  SELECT * FROM <table> [ WHERE <column> = … ];
  ```

- **At least one of: { … | … | … }**. In this example the SELECT statement retrieval list must include an asterisk (*), retrieving all columns in a table, or a list of one or more columns.

```
SELECT { * | { <column>, … } } FROM <table>;
```

Note: This is not a precise interpretation of Backus-Naur Form where curly braces usually represent zero or more. In this book curly braces represent one or more iterations, never zero.

B

Database Scripts

All relevant schema scripts can be found from a simple menu on my website at the following URL:

```
http://www.oracledbaexpert.com/sqlserver/tuning/index.html

IF @@TRANCOUNT > 0
    ROLLBACK TRAN

--
-- This loads 5000 customers, 10000 accounts and 100 branches
--
GO

--DROP DATABASE BankingDB
--GO

CREATE DATABASE BankingDB
GO
USE BankingDB
GO
SET NOCOUNT ON
GO

BACKUP LOG BankingDB WITH TRUNCATE_ONLY

DROP TABLE branches
GO
DROP TABLE accounts
GO
```

```
DROP TABLE customers
GO
DROP TABLE country
GO
DROP TABLE region
GO
DROP TABLE employee
GO

CREATE TABLE employee
(
     employee_id         CHAR(8)
    ,lname               CHAR(10)
    ,fname               CHAR(10)
    ,supervisor_id       CHAR(8)
    ,CONSTRAINT pk_employee PRIMARY KEY
        CLUSTERED(employee_id)
    ,CONSTRAINT fk_employee_supervisor FOREIGN
KEY(supervisor_id) REFERENCES employee
)
GO
CREATE NONCLUSTERED INDEX fkx_employee_supervisor ON
    employee(employee_id)
GO

CREATE TABLE region
(
     region_id       INT          NOT NULL
    ,region          CHAR(40)     NOT NULL
    ,population      INT          NULL
    ,area            INT          NULL
    ,CONSTRAINT pk_region PRIMARY KEY CLUSTERED(region_id)
)
GO

CREATE TABLE country
(
     country_id        INT          NOT NULL
    ,region_id         INT          NOT NULL
    ,country           CHAR(40)     NOT NULL
    ,code              CHAR(2)      NOT NULL
    ,population        INT          NULL
    ,area              INT          NULL
    ,fxcode            CHAR(3)      NULL
    ,currency          CHAR(40)     NULL
    ,rate              FLOAT        NULL
    ,CONSTRAINT pk_country PRIMARY KEY CLUSTERED(country_id)
    ,CONSTRAINT fk_country_region FOREIGN KEY(region_id)
    REFERENCES region
)
GO
CREATE NONCLUSTERED INDEX fkx_country_region ON
    country(region_id)
GO

CREATE TABLE customers
(
     customer_no      INT          NOT NULL
    ,region_id        INT          NOT NULL
    ,name             CHAR(40)     NOT NULL
    ,creditnotes      CHAR(1800)   NOT NULL
    ,CONSTRAINT pk_customers PRIMARY KEY
       CLUSTERED(customer_no)
    ,CONSTRAINT fk_customers_region FOREIGN KEY(region_id)
    REFERENCES region
)
GO
CREATE NONCLUSTERED INDEX fkx_customers_region ON
    customers(region_id)
GO

CREATE TABLE branches
(
     branch_no           INT        NOT NULL
    ,country_id          INT        NOT NULL
    ,branch_name         CHAR(60)   NOT NULL
    ,branch_address      CHAR(400)  NOT NULL
    ,managers_name       CHAR(60)   NOT NULL
    ,CONSTRAINT pk_branches PRIMARY KEY CLUSTERED(branch_no)
    ,CONSTRAINT fk_branches_country FOREIGN KEY(country_id)
       REFERENCES country
)
GO
```

```
CREATE NONCLUSTERED INDEX fkx_branches_country ON
    branches(country_id)
GO

CREATE TABLE accounts
(
     account_no        INT          NOT NULL
    ,customer_no       INT          NOT NULL
    ,branch_no         INT          NOT NULL
    ,balance           MONEY        NOT NULL
    ,account_notes     CHAR(1800)   NOT NULL
    ,CONSTRAINT pk_accounts PRIMARY KEY CLUSTERED(account_no)
    ,CONSTRAINT fk_accounts_customers FOREIGN
        KEY(customer_no) REFERENCES customers
    ,CONSTRAINT fk_accounts_branches FOREIGN KEY(branch_no)
    REFERENCES branches
)
GO
CREATE NONCLUSTERED INDEX fkx_accounts_customers ON
    accounts(customer_no)
GO
CREATE NONCLUSTERED INDEX fkx_accounts_branches ON
    accounts(branch_no)
GO

--
-- INSERT THE CUSTOMERS AND ACCOUNTS TABLE
--

BEGIN TRAN
INSERT into region (region_id,region,population,area)
VALUES(1,'Africa',789548670,26780325)
INSERT into region (region_id,region,population,area)
VALUES(2,'Asia',47382633,657741)
INSERT into region (region_id,region,population,area)
VALUES(3,'Australasia',24340222,7886602)
INSERT into region (region_id,region,population,area)
VALUES(4,'Caribbean',40417697,268857)
INSERT into region (region_id,region,population,area)
VALUES(5,'Central America',142653392,2360325)
INSERT into region (region_id,region,population,area)
VALUES(6,'Europe',488674441,4583335)
```

```
INSERT into region (region_id,region,population,area)
VALUES(7,'Far East',2100636517,15357441)
INSERT into region (region_id,region,population,area)
VALUES(8,'Middle East',294625718,6798768)
INSERT into region (region_id,region,population,area)
VALUES(9,'Near East',1499157105,4721322)
INSERT into region (region_id,region,population,area)
VALUES(10,'North America',331599508,18729272)
INSERT into region (region_id,region,population,area)
VALUES(11,'Oceania',9133256,536238)
INSERT into region (region_id,region,population,area)
VALUES(12,'Russian Federation',258037209,21237500)
INSERT into region (region_id,region,population,area)
VALUES(13,'South America',375489788,17545171)
COMMIT TRAN

BEGIN TRAN
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(1,1,'Algeria','AG',32930091,2381741,'DZD','Algeria
Dinars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(2,1,'Angola','AO',12127071,1246699,'AOA','Kwanza',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(3,1,'Benin','BN',7862944,110619,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(4,1,'Botswana','BC',1639833,585371,'BWP','Pulas',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(5,1,'Burkina
Faso','UV',13902972,273799,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(6,1,'Burundi','BY',8090068,25649,'BIF','Francs',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(7,1,'Central African
Republic','CT',4303356,622980,'','',0)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(8,1,'Congo','CG',62660551,2267599,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(9,1,'Djibouti','DJ',486530,21979,'DJF','Francs',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(10,1,'Equatorial
Guinea','EK',540109,28050,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(11,1,'Ethiopia','ET',74777981,1119683,'ETB','Birr',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(12,1,'Gabon','GB',1424906,257669,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(13,1,'Gambia','GA',0,10000,'GMD','Dalasi',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(14,1,'Ghana','GH',22409572,230020,'GHC','Cedis',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(15,1,'Guinea','GV',9690222,245861,'GNF','Francs',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(16,1,'Guinea-
Bissau','PU',1442029,28000,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(17,1,'Ivory Coast','IY',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(18,1,'Kenya','KE',34707817,569251,'KES','Shillings',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(19,1,'Liberia','LI',3042004,96320,'LRD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(20,1,'Libya','LY',5900754,1759540,'LYD','Dinars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(21,1,'Madagascar','MA',18595469,581540,'MGA','Ariary',
0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(22,1,'Malawi','MI',13013926,94079,'MWK','Kwachas',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(23,1,'Mali','ML',11716829,1219999,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(24,1,'Mauritania','MR',3177388,1030400,'MRO','Ouguiyas
',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(25,1,'Mauritius','MP',1240827,1849,'MUR','Rupees',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(26,1,'Morocco','MO',33241259,446301,'MAD','Dirhams',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(27,1,'Mozambique','MZ',19686505,784089,'MZM','Meticais
',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(28,1,'Namibia','WA',2044147,823291,'NAD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(29,1,'Niger','NG',12525094,1266699,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(30,1,'Nigeria','NI',131859731,910771,'NGN','Nairas',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
```

```
VALUES(31,1,'Rwanda','RW',8648248,24949,'RWF','Rwanda
Francs',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(32,1,'Senegal','SG',11987121,191999,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(33,1,'Sierra
Leone','SL',6005250,71621,'SLL','Leones',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(34,1,'Somalia','SO',8863338,627339,'SOS','Shillings',0
)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(35,1,'South
Africa','SF',44187637,1221040,'ZAR','Rand',6.2225)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(36,1,'Sudan','SU',41236378,2376001,'SDD','Dinars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(37,1,'Swaziland','WZ',1136334,17200,'SZL','Emalangeni'
,0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(38,1,'Tanzania','TZ',37445392,886039,'TZS','Shillings'
,0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(39,1,'Togo','TO',5548702,54390,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(40,1,'Tunisia','TS',10175014,155361,'TND','Dinars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(41,1,'Uganda','UG',28195754,199710,'UGX','Shillings',0
)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(42,1,'Zaire','ZE',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(43,1,'Zambia','ZA',11502010,740719,'ZMK','Kwacha',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(44,1,'Zimbabwe','ZI',12236805,386669,'ZWD','Zimbabwe
Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(45,2,'Burma','BM',47382633,657741,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(46,3,'Australia','AS',20264082,7617931,'AUD','Dollars'
,1.30141)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(47,3,'New
Zealand','NZ',4076140,268671,'NZD','Dollars',1.42369)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(48,4,'Bahamas','BF',0,10070,'BSD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(49,4,'Barbados','BB',279912,430,'BBD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(50,4,'Bermuda','BD',65773,49,'BMD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(51,4,'Costa
Rica','CS',4075261,50660,'CRC','Colones',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(52,4,'Cuba','CU',11382820,110860,'CUP','Pesos',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(53,4,'Dominican
Republic','DR',9183984,48381,'DOP','Pesos',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(54,4,'Haiti','HA',8308504,27560,'HTG','Gourdes',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(55,4,'Jamaica','JM',2758124,10829,'JMD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(56,4,'Martinique','MB',436131,1059,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(57,4,'Puerto
Rico','RQ',3927188,8959,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(58,5,'El
Salvador','ES',6822378,20720,'SVC','Colones',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(59,5,'Guatemala','GT',12293545,108430,'GTQ','Quetzales
',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(60,5,'Honduras','HO',7326496,111891,'HNL','Lempiras',0
)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(61,5,'Mexico','MX',107449525,1923039,'MXN','Pesos',11.
19)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(62,5,'Nicaragua','NU',5570129,120254,'NIO','Cordobas',
0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(63,5,'Panama','PM',3191319,75991,'PAB','Balboa',0)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(64,6,'Albania','AL',3581655,27400,'ALL','Leke',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(65,6,'Austria','AU',8192880,82730,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(66,6,'Belgium','BE',10379067,30230,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(67,6,'Bulgaria','BU',7385367,110549,'BGN','Leva',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(68,6,'Cyprus','CY',784301,9241,'CYP','Pounds',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(69,6,'Czech
Republic','EZ',10235455,78645,'CZK','Koruny',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(70,6,'Denmark','DA',5450661,42370,'DKK','Kroner',5.815
7)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(71,6,'Finland','FI',5231372,305470,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(72,6,'France','FR',60876136,545630,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(73,6,'Germany','GM',82422299,350261,'DM
','Deutsche Marks',1.5)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(74,6,'Greece','GR',10688058,130800,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(75,6,'Hungary','HU',9981334,92341,'HUF','Forint',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(76,6,'Iceland','IC',299388,100251,'ISK','Kronur',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(77,6,'Ireland','EI',4062235,68889,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(78,6,'Italy','IT',58133509,294019,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(79,6,'Luxembourg','LU',474413,2585,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(80,6,'Malta','MT',400214,321,'MTL','Liri',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(81,6,'Netherlands','NL',16491461,33939,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(82,6,'Northern Ireland','NR',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(83,6,'Norway','NO',4610820,307860,'NOK','Krone',6.5412
)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(84,6,'Poland','PL',38536869,304509,'PLN','Zlotych',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(85,6,'Portugal','PO',10605870,91639,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(86,6,'Romania','RO',22303552,230339,'ROL','Lei',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(87,6,'Scotland','SC',0,0,'','',0)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(88,6,'Slovak
Republic','LO',0,48800,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(89,6,'Spain','SP',40397842,499401,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(90,6,'Sweden','SW',9016596,411621,'SEK','Kronor',7.087
)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(91,6,'Switzerland','SZ',7523934,39769,'CHF','Francs',1
.217)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(92,6,'United
Kingdom','UK',60609153,241590,'GBP','Pounds',.538155)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(93,6,'Yugoslavia','YI',0,102136,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(94,7,'Cambodia','CB',13881427,176519,'KHR','Riels',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(95,7,'China','CH',1313973713,9326411,'CNY','Yuan
Renminbi',8.2765)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(96,7,'Indonesia','ID',245452739,1811831,'IDR','Rupiahs
',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(97,7,'Japan','JA',127463611,394744,'JPY','Yen',105.84)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(98,7,'Laos','LA',6368481,230800,'LAK','Kips',0)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(99,7,'Malaysia','MY',24385858,328549,'MYR','Ringgits',
3.8)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(100,7,'Mongolia','MG',2832224,1565000,'MNT','Tugriks',
0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(101,7,'Nepal','NP',28287147,136801,'NPR','Nepal
Rupees',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(102,7,'North
Korea','KN',23113019,120409,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(103,7,'Philippines','RP',89468677,298171,'PHP','Pesos'
,0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(104,7,'Singapore','SN',4492150,624,'SGD','Dollars',1.6
491)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(105,7,'Taiwan','TW',23036087,32261,'TWD','New
Dollars',31.79)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(106,7,'Thailand','TH',64631595,511771,'THB','Baht',38.
58)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(107,7,'Vietnam','VM',84402966,325361,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
```

```
VALUES(108,8,'Afghanistan','AF',31056997,647500,'AFA','Afghan
is',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(109,8,'Bahrain','BA',698585,619,'BHD','Dinars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(110,8,'Egypt','EG',78887007,995451,'EGP','Pounds',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(111,8,'Iran','IR',68688433,1635999,'IRR','Rials',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(112,8,'Iraq','IZ',26783383,433970,'IQD','Dinars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(113,8,'Israel','IS',6352117,20329,'ILS','New
Shekels',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(114,8,'Jordan','JO',5906760,91541,'JOD','Dinars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(115,8,'Kuwait','KU',2418393,17819,'KWD','Dinars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(116,8,'Lebanon','LE',3874050,10230,'LBP','Pounds',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(117,8,'Saudi
Arabia','SA',27019731,2149690,'SAR','Riyals',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(118,8,'Syria','SY',18881361,184051,'SYP','Pounds',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(119,8,'Yemen','YM',21456188,527969,'YER','Rials',0)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(120,9,'Bangladesh','BG',147365352,133911,'BDT','Taka',
0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(121,9,'India','IN',1095351995,2973190,'INR','Rupees',4
3.62)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(122,9,'Pakistan','PK',165803560,778720,'PKR','Rupees',
0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(123,9,'Sri
Lanka','CE',20222240,64740,'LKR','Rupees',99.4)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(124,9,'Turkey','TU',70413958,770761,'TRY','New
Lira',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(125,10,'Canada','CA',33098932,9220970,'CAD','Dollars',
1.2511)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(126,10,'Greenland','GL',56361,341701,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(127,10,'United
States','US',298444215,9166601,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(128,11,'American
Samoa','AQ',57794,199,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(129,11,'Bhutan','BT',2279723,47001,'BTN','Ngultrum',0)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(130,11,'Comoros Islands','CD',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(131,11,'Falkland
Islands','FS',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(132,11,'Fiji','FJ',905949,18270,'FJD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(133,11,'Maldive
Islands','MV',0,300,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(134,11,'New
Caledonia','NC',219246,18759,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(135,11,'Oceania','OC',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(136,11,'Papua New
Guinea','PP',5670544,451709,'PGK','Kina',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(137,12,'Russia','RS',142893540,17075400,'RUB','Rubles'
,0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(138,13,'Argentina','AR',39921833,2736690,'ARS','Pesos'
,0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(139,13,'Belize','BH',287730,22800,'BZD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(140,13,'Bolivia','BL',8989046,1084389,'BOB','Boliviano
s',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(141,13,'Brazil','BR',188078227,8456511,'BRL','Brazil
Real',2.6075)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(142,13,'Chile','CI',16134219,748800,'CLP','Pesos',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(143,13,'Colombia','CO',43593035,1038699,'COP','Pesos',
0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(144,13,'Ecuador','EC',13547510,276840,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(145,13,'French
Guiana','FG',199509,89150,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(146,13,'Guyana','GY',767245,196850,'GYD','Dollars',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(147,13,'Paraguay','PA',6506464,397301,'PYG','Guarani',
0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(148,13,'Peru','PE',28302603,1279999,'PEN','Nuevos
Soles',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(149,13,'Surinam','NS',0,161471,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(150,13,'Uruguay','UY',3431932,173621,'UYU','Pesos',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(151,13,'Venezuela','VE',25730435,882050,'VEB','Bolivar
es',1915.2)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(152,12,'Armenia','AM',2976372,29800,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(153,12,'Azerbaijan','AJ',7961619,86600,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(154,12,'Belarus','BO',10293011,207600,'AB
','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(155,1,'Cameroon','CM',17340702,469440,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(156,1,'Cote
Divoire','IV',0,318000,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(157,12,'Georgia','GG',4661473,69700,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(158,12,'Kazakhstan','KZ',15233244,2717300,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(159,7,'Myanmar','MM',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(160,8,'Palestinian
Territories','PT',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(161,6,'Serbia And
Montenegro','SM',0,0,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(162,7,'South
Korea','KS',48846823,98189,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(163,12,'Ukraine','UP',46710816,603700,'','',0)
```

```
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE) VALUES(164,8,'United Arab
Emirates','AE',2602713,83600,'','',0)
INSERT into country
(COUNTRY_ID,REGION_ID,COUNTRY,CODE,POPULATION,AREA,FXCODE,CUR
RENCY,RATE)
VALUES(165,12,'Uzbekistan','UZ',27307134,447400,'','',0)
COMMIT TRAN

BEGIN TRAN


INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1000,'Ropley','The High St, Ropley,
Hampshire','Ken Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1001,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1002,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1003,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1004,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1005,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1006,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1007,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1008,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1009,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1010,'Urmston','Urmston,
Manchester','Beryl Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1011,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1012,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1013,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1014,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1015,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1016,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1017,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1018,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1019,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
```

```
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1020,'Ropley','The High St, Ropley,
Hampshire','Ken Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1021,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1022,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1023,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1024,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1025,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1026,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1027,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1028,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1029,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1030,'Urmston','Urmston,
Manchester','Beryl Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1031,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1032,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1033,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1034,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1035,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1036,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1037,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1038,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1039,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1040,'Urmston','Urmston,
Manchester','Beryl Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1041,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1042,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
```

```
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1043,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1044,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1045,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1046,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1047,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1048,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1049,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1050,'Ropley','The High St, Ropley,
Hampshire','Ken Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1051,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1052,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1053,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1054,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1055,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1056,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1057,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1058,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1059,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1060,'Urmston','Urmston,
Manchester','Beryl Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1061,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1062,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1063,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1064,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1065,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
```

```
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1066,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1067,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1068,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1069,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1070,'Urmston','Urmston,
Manchester','Beryl Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1071,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1072,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1073,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1074,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1075,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1076,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1077,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1078,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1079,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1080,'Urmston','Urmston,
Manchester','Beryl Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1081,'Epsom','The Main St, Epsom,
Surrey','Fred Stanley')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1082,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1083,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1084,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1085,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1086,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1087,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1088,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
```

```
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1089,'Beech','Beech, Alton,
Hampshire','Margaret Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1090,'Urmston','Urmston,
Manchester','Beryl Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1091,'Ropley','The High St, Ropley,
Hampshire','Ken Smith')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1092,'Chandlers','FordElm St,
Chandlers Ford, Hampshire','Lilian Jones')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1093,'Reading','Station St, Reading,
Berkshire','Bill Burns')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1094,'Bracknell','Oak Rd., Bracknell,
Berkshire','Pat Phillips')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1095,'Sandwich','West St., Sandwich,
Kent','Peter Hunt')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1096,'Poole','Water St., Poole,
Dorset','Andy James')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1097,'Gillingham','Church St.,
Gillingham, Dorset','Douglas Adams')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1098,'Hastings','St. Helens Rd.,
Hastings, East Sussex','Bill Burrows')
INSERT into
branches(country_id,branch_no,branch_name,branch_address,mana
gers_name) VALUES (92, 1099,'Beech','Beech, Alton,
Hampshire','Margaret Smith')

COMMIT TRAN
```

```
BEGIN TRAN

DECLARE @count INT

SELECT @count = 0
WHILE @count < 5000
BEGIN
    SELECT @count=@count+1
    INSERT customers VALUES (@count, 6, 'ABC' + CAST(@count
AS CHAR(37)),'A fine customer')
    INSERT accounts  VALUES (@count,@count,(@count%100)+1000,
ROUND(RAND() * 10000,2), 'Never overdrawn')
    INSERT accounts  VALUES
(@count+100000,@count,(@count%100)+1000, ROUND(RAND() *
10000,2),'Never overdrawn')
    IF(@count%1000) = 0
SELECT CONVERT(VARCHAR(10),@count)+ ' Customers Loaded'
END
COMMIT TRAN

SET NOCOUNT OFF
```

C

Performance Strategies and Tuning Checklist

This Appendix contains a few thoughts that might be useful as an *aide-memoir* when you are considering performance issues.

C.1 System resource use

The following apply to hardware resource:

- Establish trends. Use the System Monitor to monitor resources into a log file over a period of time. Get to know the normal ranges of the key counters.
- When using the System Monitor interactively, run the graphical user interface on a machine other than the server being monitored to minimize the System Monitor impact.
- Do not jump to conclusions. The performance problem may be caused by something you do not expect. It's easy to become convinced that something is causing a problem and to subconsciously twist the evidence to fit your theory.
- Remember that system resource bottlenecks may be a symptom of something else. A classic is a disk I/O bottleneck caused by paging due to a memory shortage.
- Ensure that you have sufficient page file space.
- Remove services and protocols you are not using from the server. Do not run a screen saver on the server.
- Try to run SQL Server on a dedicated server with no other applications running. It is much easier to optimize SQL Server in this situation. Try to avoid installing SQL Server on a Domain Controller (PDC).

- Place tempdb on a fast device. Use the System Monitor or Alert subsystem to track it if it expands dynamically. By default it will be reset to its initial size on SQL Server restart. It may be beneficial to manually expand it to the size to which it frequently grows.
- Use RAID for your database and transaction log. One approach would be to use a RAID 0 stripe set for the database and mirror it. Use a dedicated disk for the transaction log and mirror it. Hardware-based RAID is faster than software-based RAID.
- Use a good quality network card. A 32-bit network card has better throughput than a 16-bit card.

C.2 Choosing efficient indexes

It is likely that for all but the smallest of tables the database designer will need to define indexes. These will probably consist of a clustered index with a number of non-clustered indexes. Queries benefit from lots of indexes, but too many indexes will degrade the performance of Transact-SQL statements that change data, such as INSERT, UPDATE, and DELETE, since all the indexes will need to be maintained, which requires CPU and disk I/O. Even worse, many indexes being updated are likely to increase lock contention.

- Consider using a clustered index in the following situations:
  - The physical ordering supports the range retrievals of important queries—that is, queries that use BETWEEN and LIKE.
  - Few duplicate values mean that an equality test (=) returns few rows.
  - Many duplicate values mean that an equality test (=) returns many rows.
  - The clustered index key is used in the ORDER BY clause of critical queries.
  - The clustered index supports the GROUP BY clause of critical queries.
  - For a given row in the outer table of a join, there are few rows that match in the inner table. A clustered index on the join column in the inner table will be beneficial.
  - For a given row in the outer table of a join, there are many rows that match in the inner table. A clustered index on the join column in the inner table will be beneficial.
- When to avoid using a clustered index:

    - On a volatile column. A volatile column is a column that is updated frequently. This would result in the data row moving around the table repeatedly.

- Consider using a non-clustered index in the following situations:
    - Few duplicate values mean that an equality test (=) returns few rows.
    - The non-clustered index key is used in the ORDER BY clause of critical queries.
    - The non-clustered index supports the GROUP BY clause of critical queries.
    - For a given row in the outer table of a join, there are few rows that match in the inner table. A clustered index on the join column in the inner table will be beneficial.
    - A critical query can be efficiently covered.
    - Many applications will require the selection of a row by the primary key. This is a single-row selection and therefore would normally benefit from the creation of an index containing the same columns as the primary key. Since it is not common to request ranges of primary keys, a non-clustered index is probably the best option. If a primary key constraint is created, the index will be automatically created; it is recommended that this be a non-clustered index.

- Avoid using a non-clustered index:
    - When a query returns many rows, such as a range retrieval, or when there are many duplicate values returned by an equality test. Also, if, for a given row in the outer table of a join, there are many rows that match in the inner table, a non-clustered index on the join column in the inner table will not be beneficial.
    - Avoid using a non-clustered index on a volatile column. The result may not be as unfavorable as using a clustered index, since the data row will not move; however, the index will still have to be maintained.

- Some general guidelines are as follows:
    - Do not create an index on a column that is not very selective. An example of this would be a column that contained a status flag containing two or three values. It is unlikely that such an index would be used by the query optimizer.
    - Be careful when creating indexes with large keys. Fewer keys can be held in an index page, resulting in many index pages and

deeper indexes. Take care with a large key in a clustered index. This will be used as the pointer in all the non-clustered indexes on the table.

- Regularly check the levels of internal and external page fragmentation with DBCC SHOWCONTIG. Tidy up by rebuilding indexes. Make sure that there is enough free space in the database to rebuild clustered indexes. Another approach is to use the Database Maintenance Wizard.
- Consider using DBCC INDEXDEFRAG on tables where there is little opportunity for maintenance—for example, a 24 × 7 system.

C.3 Helping the Query Optimizer

Can you help the optimizer to improve query processing?

- Ensure that the UPDATE STATISTICS statement (or sp_updatestats) is run regularly.
- Set the database options to allow automatic statistics updating and creation.
- Always test query performance on representative data. Data distributions that do not reflect live data in the production database and tables that are smaller than those in the production database could result in query plans different from those used when the application goes live.
- Make sure that join conditions are not omitted. Always check in the case of joins involving many tables that *N* tables must have a minimum of *N* – 1 join conditions. Better still, use the ANSI SQL-92 join syntax.
- Try to establish a standard so that program documentation includes an attached showplan output. This has a number of advantages. First, it forces the SQL developer to actually run the query and obtain a showplan output, which otherwise may not have happened. Second, it allows the person responsible for database performance to quickly scan the showplan output for obvious problems. Third, if the query performance suddenly degrades in the future, it is easy to check if the query optimizer has adopted a new query plan. Attaching statistics IO output is also recommended.
- Use query optimizer hints only if it is absolutely necessary. Revisit them to check if the plan they force is still the most efficient.

- Ensure that stored procedures are not being passed a range of parameters such that a highly inefficient query plan is being used for some values.
- The use of order by, distinct, and union in a query results in SQL Server having to do more work. If they can be avoided, do so. It might be that you know there are no duplicates, or a sort may be performed elsewhere, perhaps on the client.

C.4 Avoiding lock contention

No matter how well the database is tuned to minimize disk I/O, all the database designer's efforts will be wasted if lock contention is prevalent in the database. SQL Server's locking mechanisms were described in Chapter 10, and we will now look at some general guidelines to follow when designing a database. Remember that in most multiuser systems that make changes to data some lock contention is unavoidable. The secret is to minimize both the locking hot spots and the length of time for which locks are held. There are a number of guidelines to adhere to:

- **Rule 1: Keep transactions as short as possible.** If a transaction has placed an exclusive lock on a row, page, or table, it will keep that lock until it ends with a commit or rollback. This is also true with shared locks if the REPEATABLE, SERIALIZABLE, or HOLDLOCK hints are used or the repeatable read or serializable isolation level is used. The longer the lock is held, the more chance there will be that the lock blocks another user. This has a cascade effect, with the blocked user blocking other users. Minimize the time the locks are held. Do not perform work inside a transaction that can be performed outside of it.
- **Rule 2: Do not hold locks across user interactions.** This follows from Rule 1. Unless special considerations apply, you have a real need to, and you know what you are doing, this rule should be adhered to at all costs in a multiuser environment. What does this mean? It means that transactions should be completed before control is passed back to the user, and the transaction should not be active while the user is staring at the screen. The reasons are obvious. The computer may process a transaction's workload in less than a second, and if that transaction then completes, another transaction will only have waited a fraction of a second before it acquired its locks. If, however, a trans-

action places locks on rows, pages, or tables, and the transaction is left active while the application returns to the user, it will keep its locks while the user stares at the screen, scratches his or her head, chats with a colleague, or, worse still, goes to lunch! This could be, and usually is, disastrous for system throughput, and it is more commonplace that one might imagine! I know of instances where businesses have stopped trading for critical periods of time because a user went to lunch while a screen prompt sat on his or her workstation. This is not the user's fault. Blame resides with the application designer. If it becomes necessary to retrieve data in the database for later modification, it is usually far better to choose an option where locks are not held on database objects and an optimistic locking approach is taken—that is, the retrieved rows are not locked and, when updates are eventually performed, a check is made in the application to see if another user has changed the data since the data was read. SQL Server provides the row version data type to assist the developer.

- **Rule 3: Try not to interleave updates and reads.** If a transaction changes data when it starts, it will hold exclusive locks until it finishes. Try not to change data and then spend time reading data. If possible read the data, save all of the updates until the end of the transaction, and then issue them in one short burst. This minimizes the length of time that exclusive locks are held.
- **Rule 4: Help the query optimizer to choose indexed access.** The query optimizer chooses whether a table scan or index is used to retrieve data. Judicious use of indexes and care when writing Transact-SQL statements will help the query optimizer to choose an indexed access. From a locking contention viewpoint this is preferable to a table scan, since a table scan may lock at the table or page level if shared locks are to be held.
- **Rule 5: Only lock as strictly as is necessary to meet your integrity requirements.** Only hold shared locks if you require that the row you have read must not be changed by anyone else before your transaction ends.
- **Rule 6: Update tables in the same order throughout the application.** If one program updates table A and then updates table B, and another program updates table B and then updates table A, there is potential for deadlock. It is better to settle on some simple application development standard, such as always updating tables in alphabetical order wherever possible. In this case, the first program will

cause the second program to wait cleanly and avoid the potential deadlock scenario.

- **Rule 7: Perform multiuser testing before the application goes live**. This is often forgotten or left to the last minute. Whether you use sophisticated multiuser testing products or you persuade your users to stay late in the evening—do it!

We could add more rules but we have found that if the above seven are adhered to, lock contention should be minimized.

C.5 Database integrity

Integrity is the natural enemy of performance:

- The greater the data consistency requirements the more the impact on performance.
- Do not implement your data integrity checks at the last minute before you go live. It does not matter whether you have used triggers or constraints, your performance is likely to suddenly drop.
- Remember that if you do not index your foreign key column(s), you are likely to experience bad performance if you delete a row from the referenced table, since a table scan will probably be performed on the child table.
- A table that has many foreign key constraints defined on it will have degraded insert performance, since many lookups will be performed against the referenced tables.

C.6 Database administration activities

Database administration can obstruct normal activity:

- Avoid running DBCC statements, UPDATE STATISTICS, and backups during periods of high user activity.
- Consider creating a reporting database to off-load reporting and ad hoc querying. This could be kept up-to-date by replication or log shipping if required.

- When loading a table using Data Transformation Services, the BULK INSERT statement, or BCP, be aware of the logging impact of the different SQL Server recovery models.
- Put the file to be loaded on the same server as the database and data file to avoid network traffic.
- Creating indexes will usually impact performance on the server, so it is better to perform index rebuilds during a quiet period.
- Creating a non-clustered index has less impact than creating a clustered index. Clustered index creation uses an exclusive table lock, whereas non-clustered index creation uses a shared table lock.
- Use the DROP_EXISTING clause of the CREATE INDEX statement when rebuilding a clustered index to minimize the impact on the non-clustered indexes on the table.
- Consider using the SORT_IN_TEMPDB option on the CREATE INDEX statement to spread the I/O load across multiple disk drives.
- When creating a database, try to set a realistic initial size to avoid multiple file extensions.
- It might be better to switch variable-length datatypes to fixed-length datatypes in some cases to avoid the potential use of forwarding pointers.
- Consider shrinking database files at periodic intervals.

C.7 Archiving data

This is a requirement that usually gets left until the last minute. The fact remains, however, that the larger a database gets, the more performance is likely to degrade. Many database administration tasks will also take longer: database backups, the update of statistics, DBCC checks, and index builds.

The reasons that performance degrades include the following:

- Larger tables mean longer table scans.
- Larger tables mean deeper indexes—hence, more I/O to reach the table row.
- Longer table scans and index traversals mean locks may be held longer.

Ensure that there is an archiving strategy in place before the database gets too large.

C.8 Read only report databases

Reporting has different requirements, to that of OLTP databases:

- If we consider a typical OLTP production system comprised of many users, we would probably expect to find that the system included many short transactions that updated the tables in the database in real-time. In reality, we would also find that there was a requirement to run long and perhaps complex reports against portions of the database. The fast-response time requirements of the lightweight online transactions and the data-hungry requirements of the heavyweight report transactions often do not mix well. The report transactions can severely impact the response times of the online transactions in the production system and in the worst case may cause lock conflict.
- One option is to separate these two different workloads into their own databases on their own server. This can never, in reality, be done completely, since there is usually no clear break between the requirements of the two systems. However, there is a case for off-loading as much reporting work as possible to another database. This also means that there will be a natural frozen cut-off point. If the report database is only updated overnight, then it will hold the close of day position all the following day, which can be a useful asset.
- A separate report database can also have extra indexes added to it that would have been unacceptable in the production database for performance reasons.
- Updating information in the report database could be a simple matter of restoring it from last night's backup of the OLTP database, or the replication capabilities present in SQL Server could be used. Whatever the method, consider the approach of separating the different workloads, since this can greatly help performance and increase flexibility.
- If the report database is created from last night's backup, there are also two more added bonuses. First, the fact that you are restoring your backup means that you can feel confident that your backup/restore scripts work. Second, since the database is identical to the

OLTP database, those lengthy DBCC integrity checks can be run on the report database instead of the OLTP database.

C.9 Denormalization

Denormalization removes granularity from a data model:

- Before considering denormalization, a fully normalized database design should be your starting point. A fully normalized database design helps to avoid data redundancy and possible update anomalies, but it usually results in a design that requires tables to be joined frequently.
- Possible approaches to denormalization include the duplication of columns from one or more tables into another to avoid the join in a critical query. For columns that are volatile, this can make updates more complex.
- Another denormalization technique is to store derived data in the database. Transactions that change data can, usually by means of triggers, modify the derived data column. This can save query time, since the work has already been done calculating the derived column.

Index