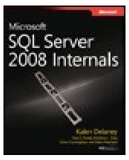


Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.
Microsoft Press. (c) 2009. Copying Prohibited.

Chapter 1: SQL Server 2008 Architecture and Configuration

Kalen Delaney

Microsoft SQL Server is Microsoft's premiere database management system, and SQL Server 2008 is the most powerful and feature-rich version yet. In addition to the core database engine, which allows you to store and retrieve large volumes of relational data, and the world-class Query Optimizer, which determines the fastest way to process your queries and access your data, dozens of other components increase the usability of your data and make your data and applications more available and more scalable. As you can imagine, no single book could cover all these features in depth. This book, *SQL Server 2008 Internals,* covers the main features of the core database engine.

Throughout this book, we'll delve into the details of specific features of the SQL Server Database Engine. In this first chapter, you'll get a high-level view of the components of that engine and how they work together. My goal is to help you understand how the topics covered in subsequent chapters fit into the overall operations of the engine.

In this chapter, however, we'll dig deeper into one big area of the SQL Server Database Engine that isn't covered later: the SQL operating system (SQLOS) and, in particular, the components related to memory management and scheduling. We'll also look at the metadata that SQL Server makes available to allow you to observe the engine behavior and data organization.

SQL Server Editions

Each version of SQL Server comes in various editions, which you can think of as a subset of the product features, with its own specific pricing and licensing requirements. Although we won't be discussing pricing and licensing in this book, some of the information about editions is important, because of the features that are available with each edition. The editions available and the feature list that each supports is described in detail in *SQL Server Books Online,* but here we will list the main editions. You can verify what edition you are running with the following query:

```
SELECT SERVERPROPERTY('Edition');
```

There is also a server property called *EngineEdition* that you can inspect, as follows:

```
SELECT SERVERPROPERTY('EngineEdition');
```

The *EngineEdition* property returns a value of 2, 3, or 4 (1 is not a possible value), and this value determines what features are available. A value of 3 indicates that your SQL Server edition is either Enterprise, Enterprise Evaluation, or Developer. These three editions have exactly the same features and functionality. If your *EngineEdition* value is 2, your edition is either Standard or Workgroup, and fewer features are available. The features and behaviors discussed in this book will be the ones available in one of these two engine editions. The features in Enterprise edition (as well as in Developer edition and Enterprise Evaluation edition) that are not in Standard edition generally relate to scalability and high-availability features, but there are other Enterprise-only features as well. When we discuss such features that are considered Enterprise-only, we'll let you know. For full details on what is in each edition, see the *SQL Server Books Online* topic "Features Supported by the Editions of SQL Server 2008." (A value of 4 for *EngineEdition* indicates that your SQL Server edition is an Express edition, which includes SQL Server Express, SQL Server Express with Advanced Services, or Windows Embedded SQL. None of these versions will be discussed specifically.) There is also a *SERVERPROPERTY* property called *EditionID*, which allows you to differentiate between the specific editions within each of the different *EngineEdition* values (that is, it allows you to differentiate between Enterprise, Enterprise Evaluation, and Developer editions).

SQL Server Metadata

SQL Server maintains a set of tables that store information about all the objects, data types, constraints, configuration options, and resources available to SQL Server. In SQL Server 2008, these tables are called the *system base tables*. Some of the system base tables exist only in the *master* database and contain system-wide information, and others exist in every database (including *master*) and contain information about the objects and resources belonging to that particular database. Beginning with SQL Server 2005, the system base tables are not always visible by default, in *master* or any other database. You won't see them when you expand the *tables* node in the Object Explorer in SQL Server Management Studio, and unless you are a system administrator, you won't see them when you execute the *sp_help* system procedure. If you log in as a system administrator and select from the catalog view (discussed shortly) called *sys.objects*, you can see the names of all the system tables. For example, the following query returns 58 rows of output on my SQL Server 2008 instance:

```
USE master;
SELECT name FROM sys.objects
WHERE type_desc = 'SYSTEM_TABLE';
```

But even as a system administrator, if you try to select data from one of the tables whose names are returned by the preceding query, you get a 208 error, indicating that the object name is invalid. The only way to see the data in the system base tables is to make a connection using the dedicated administrator connection (DAC), which we'll tell you about in the section entitled "The Scheduler," later in this chapter. Keep in mind that the system base tables are used for internal purposes only within the Database Engine and are not intended for general use. They are subject to change, and compatibility is not guaranteed. In SQL Server 2008, there are three types of system metadata objects. One type is Dynamic Management Objects, which we'll talk about later in this chapter when we discuss SQL Server scheduling and memory management. These Dynamic Management Objects don't really correspond to physical tables—they contain information gathered from internal structures to allow you to observe the current state of your SQL Server instance. The other two types of system objects are actually views built on top of the system base tables.

Compatibility Views

Although you were allowed to see data in the system tables in versions of SQL Server before 2005, you weren't encouraged to do this. Nevertheless, many people used system tables for developing their own troubleshooting and reporting tools and techniques, providing result sets that aren't available using the supplied system procedures. You might assume that due to the inaccessibility of the system base tables, you would have to use the DAC to utilize your homegrown tools when using SQL Server 2005 or 2008. However, you still might be disappointed. Many of the names and much of the content of the SQL Server 2000 system tables have changed, so any code that used them is completely unusable even with the DAC. The DAC is intended only for emergency access, and no support is provided for any other use of it. To save you from this grief, SQL Server 2005 and 2008 offer a set of compatibility views that allow you to continue to access a subset of the SQL Server 2000 system tables. These views are accessible from any database, although they are created in the hidden resource database.

Some of the compatibility views have names that might be quite familiar to you, such as *sysobjects, sysindexes, sysusers,* and *sysdatabases*. Others, like *sysmembers* and *sysmessages,* might be less familiar. For compatibility reasons, the views in SQL Server 2008 have the same names as their SQL Server 2000 counterparts, as well as the same column names, which means that any code that uses the SQL Server 2000 system tables won't break. However, when you select from these views, you are not guaranteed to get exactly the same results that you get from the corresponding tables in SQL Server 2000. In addition, the compatibility views do not contain any metadata related to new SQL Server 2005 or 2008 features, such as partitioning or the Resource Governor. You should consider the compatibility views to be for backward compatibility only; going forward, you should consider using other metadata mechanisms, such as the catalog view discussed in the next section. All these compatibility views will be removed in a future version of SQL Server.

More Info You can find a complete list of names and the columns in these views in *SQL Server Books Online*.

SQL Server 2005 and 2008 also provide compatibility views for the SQL Server 2000 pseudotables, such as *sysprocesses* and *syscacheobjects*. Pseudotables are tables that are not based on data stored on disk but are built as needed from internal structures and can be queried exactly as if they are tables. SQL Server 2005 replaced these pseudotables with Dynamic Management Objects. Note that there is not always a one-to-one correspondence between the SQL Server 2000 pseudotables and the SQL Server 2005 and SQL Server 2008 Dynamic Management Objects. For example, for SQL Server 2008 to retrieve all the information available in *sysprocesses*, you must access three Dynamic Management Objects: *sys.dm_exec_connections, sys.dm_exec_sessions,* and *sys.dm_exec_requests*.

Catalog Views

SQL Server 2005 introduced a set of catalog views as a general interface to the persisted system metadata. All the catalog views (as well as the Dynamic Management Objects and compatibility views) are in the *sys* schema, and you must reference the schema name when you access the objects. Some of the names are easy to remember because they are similar to the SQL Server 2000 system table names. For example, there is a catalog view called *objects* in the *sys* schema, so to reference the view, the following can be executed:

```
SELECT * FROM sys.objects;
```

Similarly, there are catalog views called *sys.indexes* and *sys.databases*, but the columns displayed for these catalog views are very different from the columns in the compatibility views. Because the output from these types of queries is too wide to reproduce, let me just suggest that you run these two queries yourself and observe the difference:

```
SELECT * FROM sys.databases;
SELECT * FROM sysdatabases;
```

The *sysdatabases* compatibility view is in the *sys* schema, so you can reference it as *sys.sysdatabases*. You can also reference it using *dbo.sysdatabases*. But again, for compatibility reasons, the schema name is not required, as it is for the catalog views. (That is, you cannot simply select from a view called *databases*; you must use the schema *sys* as a prefix.)

When you compare the output from the two preceding queries, you might notice that there are a lot more columns in the *sys.databases* catalog view. Instead of a bitmap *status* field that needs to be decoded, each possible database property has its own column in *sys.databases*. With SQL Server 2000, running the system procedure *sp_helpdb* decodes all these database options, but because *sp_helpdb* is a procedure, it is difficult to filter the results. As a view, *sys.databases* can be queried and filtered. For example, if you want to know which databases are in *simple* recovery mode, you can run the following:

```
SELECT name FROM sys.databases
WHERE recovery_model_desc = 'SIMPLE';
```

The catalog views are built on an inheritance model, so sets of attributes common to many objects don't have to be redefined internally. For example, *sys.objects* contains all the columns for attributes common to all types of objects, and the views *sys.tables* and *sys.views* contain all the same columns as *sys.objects*, as well as some additional columns that are relevant only to the particular type of objects. If you select from *sys.objects*, you get 12 columns, and if you then select from *sys.tables*, you get exactly the same 12 columns in the same order, plus 15 additional columns that aren't applicable to all types of objects but are meaningful for tables. In addition, although the base view *sys.objects* contains a subset of columns compared to the derived views such as *sys.tables*, it contains a superset of rows compared to a derived view. For example, the *sys.objects* view shows metadata for procedures and views in addition to that for tables, whereas the *sys.tables* view shows only rows for tables. So I can summarize the relationship between the base view and the derived views as follows: "The base views contain a subset of columns and a superset of rows, and the derived views contain a superset of columns and a subset of rows."

Just as in SQL Server 2000, some of the metadata appears only in the *master* database, and it keeps track of system-wide data, such as databases and logins. Other metadata is available in every database, such as objects and permissions. The *SQL Server Books Online* topic "Mapping System Tables to System Views" categorizes its objects into two lists—those appearing only in *master* and those appearing in all databases. Note that metadata appearing only in the *msdb* database is not available through catalog views but is still available in system tables, in the schema *dbo*. This includes metadata for backup and restore, replication, Database Maintenance Plans, Integration Services, log shipping, and SQL Server Agent.

As views, these metadata objects are based on an underlying Transact-SQL (T-SQL) definition. The most straightforward way to see the definition of these views is by using the *object_definition* function. (You can also see the definition of these system views by using *sp_helptext* or by selecting from the catalog view *sys.system_sql_modules*.) So to see the definition of *sys.tables*, you can execute the following:

```
SELECT object_definition (object_id('sys.tables'));
```

If you execute the preceding *SELECT* statement, you'll see that the definition of *sys.tables* references several completely undocumented system objects. On the other hand, some system object definitions refer only to objects that are documented. For example, the definition of the compatibility view *syscacheobjects* refers only to three Dynamic Management Objects (one view, *sys.dm_exec_cached_plans*, and two functions, *sys.dm_exec_sql_text* and *sys.dm_exec_plan_attributes*) that are fully documented.

The metadata with names starting with 'sys.dm_', such as the just-mentioned *sys.dm_exec_ cached_plans*, are considered Dynamic Management Objects, and we'll be discussing them in the next section when we discuss the SQL Server Database Engine's behavior.

Other Metadata

Although the catalog views are the recommended interface for accessing the SQL Server 2008 catalog, other tools are available as well.

Information Schema Views

Information schema views, introduced in SQL Server 7.0, were the original system table– independent view of the SQL

Server metadata. The information schema views included in SQL Server 2008 comply with the SQL-92 standard and all these views are in a schema called *INFORMATION_SCHEMA.* Some of the information available through the catalog views is available through the information schema views, and if you need to write a portable application that accesses the metadata, you should consider using these objects. However, the information schema views only show objects that are compatible with the SQL-92 standard. This means there is no information schema view for certain features, such as indexes, which are not defined in the standard. (Indexes are an implementation detail.) If your code does not need to be strictly portable, or if you need metadata about nonstandard features such as indexes, filegroups, the CLR, and SQL Server Service Broker, we suggest using the Microsoft-supplied catalog views. Most of the examples in the documentation, as well as in this and other reference books, are based on the catalog view interface.

System Functions

Most SQL Server system functions are property functions, which were introduced in SQL Server 7.0 and greatly enhanced in SQL Server 2000. SQL Server 2005 and 2008 have enhanced these functions still further. Property functions give us individual values for many SQL Server objects and also for SQL Server databases and the SQL Server instance itself. The values returned by the property functions are scalar as opposed to tabular, so they can be used as values returned by *SELECT* statements and as values to populate columns in tables. Here is the list of property functions available in SQL Server 2008:

- *SERVERPROPERTY*
- *COLUMNPROPERTY*
- *DATABASEPROPERTY*
- *DATABASEPROPERTYEX*
- *INDEXPROPERTY*
- *INDEXKEY_PROPERTY*
- *OBJECTPROPERTY*
- *OBJECTPROPERTYEX*
- *SQL_VARIANT_PROPERTY*
- *FILEPROPERTY*
- *FILEGROUPPROPERTY*
- *TYPEPROPERTY*
- *CONNECTIONPROPERTY*
- *ASSEMBLYPROPERTY*

The only way to find out what the possible property values are for the various functions is to check *SQL Server Books Online.*

Some of the information returned by the property functions can also be seen using the catalog views. For example, the *DATABASEPROPERTYEX* function has a property called *Recovery* that returns the recovery model of a database. To view the recovery model of a single database, you can use the property function as follows:

```
SELECT DATABASEPROPERTYEX('msdb', 'Recovery');
```

To view the recovery models of all our databases, you can use the *sys.databases* view:

```
SELECT name, recovery_model, recovery_model_desc
FROM sys.databases;
```

Note Columns with names ending in *_desc* are the so-called friendly name columns, and they are always paired with another column that is much more compact, but cryptic. In this case, the *recovery_model* column is a *tinyint* with a value of 1, 2, or 3. Both columns are available in the view because different consumers have different needs. For example, internally at Microsoft, the teams building the internal interfaces wanted to bind to more compact columns, whereas DBAs running adhoc queries might prefer the friendly names.

In addition to the property functions, the system functions include functions that are merely shortcuts for catalog view access. For example, to find out the database ID for the *AdventureWorks2008* database, you can either query the *sys.databases* catalog view or use the *DB_ID()* function. Both of the following *SELECT* statements should return the same result:

```
SELECT database_id
FROM sys.databases
WHERE name = 'AdventureWorks2008';

SELECT DB_ID('AdventureWorks2008');
```

System Stored Procedures

System stored procedures are the original metadata access tool, in addition to the system tables themselves. Most of the system stored procedures introduced in the very first version of SQL Server are still available. However, catalog views are a big improvement over these procedures: you have control over how much of the metadata you see because you can query the views as if they were tables. With the system stored procedures, you basically have to accept the data that it returns. Some of the procedures allow parameters, but they are very limited. So for the *sp_helpdb* procedure, you can pass a parameter to see just one database's information or not pass a parameter and see information for all databases. However, if you want to see only databases that the login *sue* owns, or just see databases that are in a lower compatibility level, you cannot do it using the supplied stored procedure. Using the catalog views, these queries are straightforward:

```
SELECT name FROM sys.databases
WHERE suser_sname(owner_sid) ='sue';

SELECT name FROM sys.databases
WHERE compatibility_level < 90;
```

Metadata Wrap-Up

Figure 1-1 shows the multiple layers of metadata available in SQL Server 2008, with the lowest layer being the system base tables (the actual catalog). Any interface that accesses the information contained in the system base tables is subject to the metadata security policies. For SQL Server 2008, that means that no users can see any metadata that they don't need to see or to which they haven't specifically been granted permissions. (There are a few exceptions, but they are very minor.) The "other metadata" refers to system information not contained in system tables, such as the internal information provided by the Dynamic Management Objects. Remember that the preferred interfaces to the system metadata are the catalog views and system functions. Although not all the compatibility views, *INFORMATION_SCHEMA* views, and system procedures are actually defined in terms of the catalog views, conceptually it is useful to think of them as another layer on top of the catalog view interface.

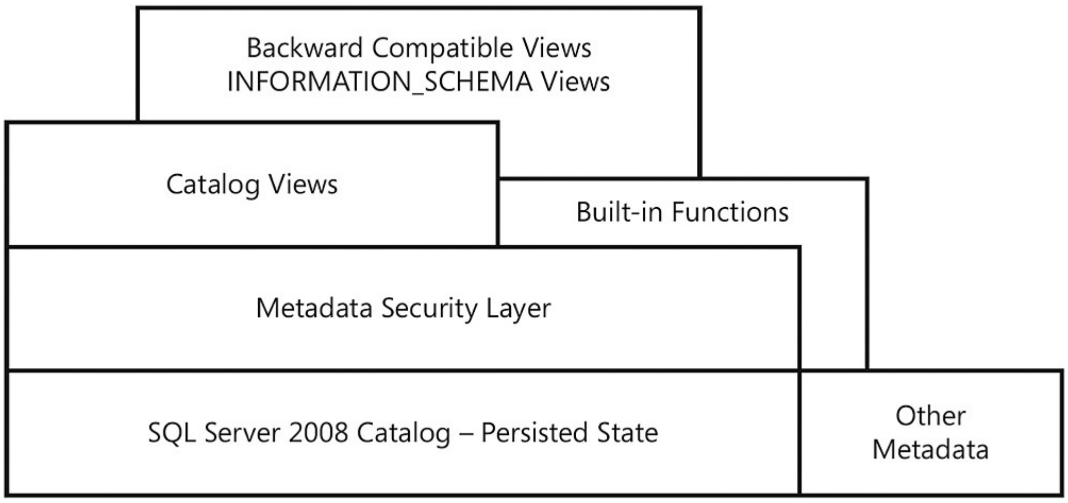


Figure 1-1: Layers of metadata in SQL Server 2008

Components of the SQL Server Engine

Figure 1-2 shows the general architecture of SQL Server, which has four major components. Three of those components, along with their subcomponents are shown in the figure: the relational engine (also called the *query processor*), the storage engine, and the SQLOS.

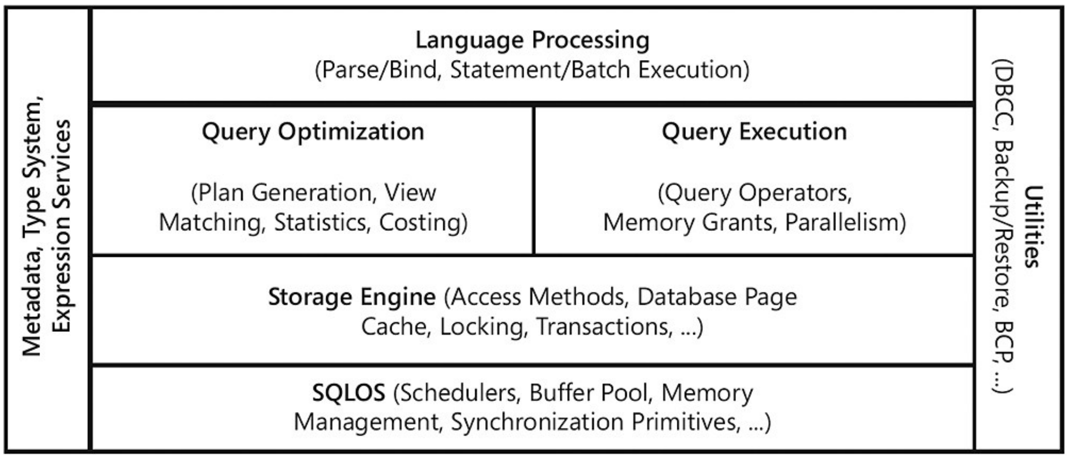

Language Processing
(Parse/Bind, Statement/Batch Execution)
Query Optimization
(Plan Generation, View Matching, Statistics, Costing)
Query Execution
(Query Operators, Memory Grants, Parallelism)
Storage Engine (Access Methods, Database Page Cache, Locking, Transactions, ...)
SQLOS (Schedulers, Buffer Pool, Memory Management, Synchronization Primitives, ...)
Metadata, Type System, Expression Services
Utilities (DBCC, Backup/Restore, BCP, ...)

Figure 1-2: The major components of the SQL Server Database Engine

(The fourth component is the protocol layer, which is not shown.) Every batch submitted to SQL Server for execution, from any client application, must interact with these four components. (For simplicity, I've made some minor omissions and simplifications and ignored certain "helper" modules among the subcomponents.)

The protocol layer receives the request and translates it into a form that the relational engine can work with, and it also takes the final results of any queries, status messages, or error messages and translates them into a form the client can understand before sending them back to the client. The relational engine layer accepts T-SQL batches and determines what to do with them. For T-SQL queries and programming constructs, it parses, compiles, and optimizes the request and oversees the process of executing the batch. As the batch is executed, if data is needed, a request for that data is passed to the storage engine. The storage engine manages all data access, both through transaction-based commands and bulk operations such as backup, bulk insert, and certain DBCC commands. The SQLOS layer handles activities that are normally considered to be operating system responsibilities, such as thread management (scheduling), synchronization primitives, deadlock detection, and memory management, including the buffer pool.

Observing Engine Behavior

SQL Server 2008 includes a suite of system objects that allow developers and database administrators to observe much of the internals of SQL Server. These metadata objects, introduced in SQL Server 2005, are called *Dynamic Management Objects*. These objects include both views and functions, but the vast majority are views. (Dynamic Management Objects are frequently referred to as Dynamic Management Views (DMVs) to reflect the fact that most of the objects are views.) You can access these metadata objects as if they reside in the *sys* schema, which exists in every SQL Server 2008 database, but they are not real tables that are stored on disk. They are similar to the pseudotables used in SQL Server 2000 for observing the active processes (*sysprocesses*) or the contents of the plan cache (*syscacheobjects*). However, the pseudotables in SQL Server 2000 do not provide any tracking of detailed resource usage and are not always directly usable to detect resource problems or state changes. Some of the DMVs allow tracking of detailed resource history, and there are more than 100 such objects that you can directly query and join with SQL *SELECT* statements, although not all of these objects are documented. The DMVs expose changing server state information that might span multiple sessions, multiple transactions, and multiple user requests. These objects can be used for diagnostics, memory and process tuning, and monitoring across all sessions in the server. They also provide much of the data available through the Management Data Warehouse's performance reports, which is a new feature in SQL Server 2008. (Note that *sysprocesses* and *syscacheobjects* are still available as compatibility views, which we mentioned in the section "SQL Server Metadata," earlier in this chapter.)

The DMVs aren't based on real tables stored in database files but are based on internal server structures, some of which we'll discuss in this chapter. We'll discuss further details about the DMVs in various places in this book, where the contents of one or more of the objects can illuminate the topics being discussed. The objects are separated into several categories based on the functional area of the information they expose. They are all in the *sys* schema and have a name that starts with *dm_*, followed by a code indicating the area of the server with which the object deals. The main categories we'll address are the following:

dm_exec_*

Contains information directly or indirectly related to the execution of user code and associated connections. For example, *sys.dm_exec_sessions* returns one row per authenticated session on SQL Server. This object contains much of the same information that *sysprocesses* contains but has even more information about the operating environment of each session.

dm_os_*

Contains low-level system information such as memory, locking, and scheduling. For example, *sys.dm_os_schedulers* is a DMV that returns one row per scheduler. It is primarily used to monitor the condition of a scheduler or to identify runaway tasks.

dm_tran_*

Contains details about current transactions. For example, *sys.dm_tran_locks* returns information about currently active lock resources. Each row represents a currently active request to the lock management component for a lock that has been granted or is waiting to be granted.

dm_io_*

Keeps track of I/O activity on networks and disks. For example, the function *sys.dm_io_virtual_file_stats* returns I/O statistics for data and log files.

dm_db_*

Contains details about databases and database objects such as indexes. For example, *sys.dm_db_index_physical_stats* is a function that returns size and fragmentation information for the data and indexes of the specified table or view.

SQL Server 2008 also has Dynamic Management Objects for many of its functional components; these include objects for monitoring full-text search catalogs, change data capture (CDC) information, service broker, replication, and the CLR.

Now let’s look at the major components of the SQL Server Database Engine.

Protocols

When an application communicates with the Database Engine, the application programming interfaces (APIs) exposed by the protocol layer formats the communication using a Microsoft-defined format called a *tabular data stream (TDS) packet*. The SQL Server Network Interface (SNI) protocol layer on both the server and client computers encapsulates the TDS packet inside a standard communication protocol, such as TCP/IP or Named Pipes. On the server side of the communication, the network libraries are part of the Database Engine. On the client side, the network libraries are part of the SQL Native Client. The configuration of the client and the instance of SQL Server determine which protocol is used.

SQL Server can be configured to support multiple protocols simultaneously, coming from different clients. Each client connects to SQL Server with a single protocol. If the client program does not know which protocols SQL Server is listening on, you can configure the client to attempt multiple protocols sequentially. The following protocols are available:

Shared Memory The simplest protocol to use, with no configurable settings. Clients using the Shared Memory protocol can connect only to a SQL Server instance running on the same computer, so this protocol is not useful for most database activity. Use this protocol for troubleshooting when you suspect that the other protocols are configured incorrectly. Clients using MDAC 2.8 or earlier cannot use the Shared Memory protocol. If such a connection is attempted, the client is switched to the Named Pipes protocol.

Named Pipes A protocol developed for local area networks (LANs). A portion of memory is used by one process to pass information to another process, so that the output of one is the input of the other. The second process can be local (on the same computer as the first) or remote (on a networked computer).

TCP/IP The most widely used protocol over the Internet. TCP/IP can communicate across interconnected networks of computers with diverse hardware architectures and operating systems. It includes standards for routing network traffic and offers advanced security features. Enabling SQL Server to use TCP/IP requires the most configuration effort, but most networked computers are already properly configured.

Virtual Interface Adapter (VIA) A protocol that works with VIA hardware. This is a specialized protocol; configuration details are available from your hardware vendor.

Tabular Data Stream Endpoints

SQL Server 2008 also allows you to create a TDS endpoint, so that SQL Server listens on an additional TCP port. During setup, SQL Server automatically creates an endpoint for each of the four protocols supported by SQL Server, and if the protocol is enabled, all users have access to it. For disabled protocols, the endpoint still exists but cannot be used. An additional endpoint is created for the DAC, which can be used only by members of the *sysadmin* fixed server role. (We'll discuss the DAC in more detail shortly.)

The Relational Engine

As mentioned earlier, the relational engine is also called the query processor. It includes the components of SQL Server that determine exactly what your query needs to do and the best way to do it. In Figure 1-2, the relational engine is shown as two primary components: Query Optimization and Query Execution. By far the most complex component of the query processor, and maybe even of the entire SQL Server product, is the Query Optimizer, which determines the best execution plan for the queries in the batch. The Query Optimizer is discussed in great detail in Chapter 8, "The Query Optimizer"; in this section, we'll give you just a high-level overview of the Query Optimizer as well as of the other components of the query processor.

The relational engine also manages the execution of queries as it requests data from the storage engine and processes the results returned. Communication between the relational engine and the storage engine is generally in terms of OLE DB row sets. (*Row set* is the OLE DB term for a *result set*.) The storage engine comprises the components needed to actually access and modify data on disk.

The Command Parser

The command parser handles T-SQL language events sent to SQL Server. It checks for proper syntax and translates T-SQL commands into an internal format that can be operated on. This internal format is known as a *query tree*. If the parser doesn't recognize the syntax, a syntax error is immediately raised that identifies where the error occurred. However, nonsyntax error messages cannot be explicit about the exact source line that caused the error. Because only the command parser can access the source of the statement, the statement is no longer available in source format when the command is actually executed.

The Query Optimizer

The Query Optimizer takes the query tree from the command parser and prepares it for execution. Statements that can't be optimized, such as flow-of-control and Data Definition Language (DDL) commands, are compiled into an internal form. The statements that are optimizable are marked as such and then passed to the Query Optimizer. The Query Optimizer is mainly concerned with the Data Manipulation Language (DML) statements *SELECT, INSERT, UPDATE,* and *DELETE,* which can be processed in more than one way, and it is the Query Optimizer's job to determine which of the many possible ways is the best. It compiles an entire command batch, optimizes queries that are optimizable, and checks security. The query optimization and compilation result in an execution plan.

The first step in producing such a plan is to *normalize* each query, which potentially breaks down a single query into multiple, fine-grained queries. After the Query Optimizer normalizes a query, it *optimizes* it, which means that it determines a plan for executing that query. Query optimization is cost-based; the Query Optimizer chooses the plan that it determines would cost the least based on internal metrics that include estimated memory requirements, CPU utilization, and number of required I/Os. The Query Optimizer considers the type of statement requested, checks the amount of data in the various tables affected, looks at the indexes available for each table, and then looks at a sampling of the data values kept for each index or column referenced in the query. The sampling of the data values is called *distribution statistics*. (Statistics will be discussed in detail in Chapter 8.) Based on the available information, the Query Optimizer considers the various access methods and processing strategies that it could use to resolve a query and chooses the most cost-effective plan.

The Query Optimizer also uses pruning heuristics to ensure that optimizing a query doesn't take longer than it would take to simply choose a plan and execute it. The Query Optimizer doesn't necessarily perform exhaustive optimization. Some products consider every possible plan and then choose the most cost-effective one. The advantage of this exhaustive optimization is that the syntax chosen for a query theoretically never causes a performance difference, no matter what syntax the user employed. But with a complex query, it could take much longer to estimate the cost of every conceivable plan than it would to accept a good plan, even if it is not the best one, and execute it.

After normalization and optimization are completed, the normalized tree produced by those processes is compiled into the execution plan, which is actually a data structure. Each command included in it specifies exactly which table will be affected, which indexes will be used (if any), which security checks must be made, and which criteria (such as equality to a specified value) must evaluate to TRUE for selection. This execution plan might be considerably more complex than is immediately apparent. In addition to the actual commands, the execution plan includes all the steps necessary to ensure that constraints are checked. Steps for calling a trigger are slightly different from those for verifying constraints. If a trigger is included for the action being taken, a call to the procedure that comprises the trigger is appended. If the trigger is an *instead-of* trigger, the call to the trigger's plan replaces the actual data modification command. For *after* triggers, the trigger's plan is branched to right after the plan for the modification statement that fired the trigger, before that modification is committed. The specific steps for the trigger are not compiled into the execution plan, unlike those for constraint verification.

A simple request to insert one row into a table with multiple constraints can result in an execution plan that requires many other tables to be accessed or expressions to be evaluated as well. In addition, the existence of a trigger can cause many more steps to be executed. The step that carries out the actual *INSERT* statement might be just a small part of the total execution plan necessary to ensure that all actions and constraints associated with adding a row are carried out.

The Query Executor

The query executor runs the execution plan that the Query Optimizer produced, acting as a dispatcher for all the commands in the execution plan. This module steps through each command of the execution plan until the batch is complete. Most of the commands require interaction with the storage engine to modify or retrieve data and to manage transactions and locking. More information on query execution, and execution plans, is available on the companion Web site, *http://www.SQLServerInternals.com/companion.*

The Storage Engine

The SQL Server storage engine includes all the components involved with the accessing and managing of data in your database. In SQL Server 2008, the storage engine is composed of three main areas: access methods, locking and transaction services, and utility commands.

Access Methods

When SQL Server needs to locate data, it calls the access methods code. The access methods code sets up and requests scans of data pages and index pages and prepares the OLE DB row sets to return to the relational engine. Similarly, when data is to be inserted, the access methods code can receive an OLE DB row set from the client. The access methods code contains components to open a table, retrieve qualified data, and update data. The access methods code doesn't actually retrieve the pages; it makes the request to the buffer manager, which ultimately serves up the page in its cache or reads it to cache from disk. When the scan starts, a look-ahead mechanism qualifies the rows or index entries on a page. The retrieving of rows that meet specified criteria is known as a *qualified retrieval*. The access methods code is employed not only for *SELECT* statements but also for qualified *UPDATE* and *DELETE* statements (for example, *UPDATE* with a WHERE clause) and for any data modification operations that need to modify index entries. Some types of access methods are listed below.

Row and Index Operations You can consider row and index operations to be components of the access methods code because they carry out the actual method of access. Each component is responsible for manipulating and maintaining its respective on-disk data structures—namely, rows of data or B-tree indexes, respectively. They understand and manipulate information on data and index pages.

The row operations code retrieves, modifies, and performs operations on individual rows. It performs an operation within a row, such as "retrieve column 2" or "write this value to column 3." As a result of the work performed by the access methods code, as well as by the lock and transaction management components (discussed shortly), the row is found and appropriately locked as part of a transaction. After formatting or modifying a row in memory, the row operations code inserts or deletes a row. There are special operations that the row operations code needs to handle if the data is a Large Object (LOB) data type—*text, image,* or *ntext*—or if the row is too large to fit on a single page and needs to be stored as overflow data. We'll look at the different types of data storage structures in Chapters 5, "Tables," 6, "Indexes: Internals and Management," and 7, "Special Storage."

The index operations code maintains and supports searches on B-trees, which are used for SQL Server indexes. An index is structured as a tree, with a root page and intermediate-level and lower-level pages. (If the tree is very small, there might not be intermediate-level pages.) A B-tree groups records that have similar index keys, thereby allowing fast access to data by searching on a key value. The B-tree's core feature is its ability to balance the index tree. (*B* stands for *balanced*.) Branches of the index tree are spliced together or split apart as necessary so that the search for any given record always traverses the same number of levels and therefore requires the same number of page accesses.

Page Allocation Operations The allocation operations code manages a collection of pages for each database and keeps track of which pages in a database have already been used, for what purpose they have been used, and how much space is available on each page. Each database is a collection of 8-KB disk pages that are spread across one or more physical files. (In Chapter 3, "Databases and Database Files," you'll find more details about the physical organization of databases.)

SQL Server uses 13 types of disk pages. The ones we'll be discussing in this book are data pages, two types of LOB pages, row-overflow pages, index pages, Page Free Space (PFS) pages, Global Allocation Map and Shared Global Allocation Map (GAM and SGAM) pages, Index Allocation Map (IAM) pages, Bulk Changed Map (BCM) pages, and Differential Changed Map (DCM) pages.

All user data is stored on data or LOB pages, and all index rows are stored on index pages. PFS pages keep track of which pages in a database are available to hold new data. Allocation pages (GAMs, SGAMs, and IAMs) keep track of the other pages. They contain no database rows and are used only internally. BCM and DCM pages are used to make backup and recovery more efficient. We'll explain these types of pages in Chapters 3 and 4, "Logging and Recovery."

Versioning Operations Another type of data access, which was added to the product in SQL Server 2005, is access through the version store. Row versioning allows SQL Server to maintain older versions of changed rows. The row versioning technology in SQL Server supports Snapshot isolation as well as other features of SQL Server 2008, including online index builds and triggers, and it is the versioning operations code that maintains row versions for whatever purpose they are needed.

Chapters 3, 5, 6, and 7 deal extensively with the internal details of the structures that the access methods code works with: databases, tables, and indexes.

Transaction Services

A core feature of SQL Server is its ability to ensure that transactions are *atomic*—that is, all or nothing. In addition, transactions must be durable, which means that if a transaction has been committed, it must be recoverable by SQL Server no matter what—even if a total system failure occurs one millisecond after the commit was acknowledged. There are actually four properties that transactions must adhere to: *atomicity, consistency, isolation,* and *durability,* called the *ACID properties*. we'll discuss all four of these properties in Chapter 10, "Transactions and Concurrency," when we discuss transaction management and concurrency issues.

In SQL Server, if work is in progress and a system failure occurs before the transaction is committed, all the work is rolled back to the state that existed before the transaction began. Write-ahead logging makes it possible to always roll back work in progress or roll forward committed work that has not yet been applied to the data pages. Write-ahead logging ensures that the record of each transaction's changes is captured on disk in the transaction log before a transaction is acknowledged as committed, and that the log records are always written to disk before the data pages where the changes were actually made are written. Writes to the transaction log are always synchronous—that is, SQL Server must wait for them to complete. Writes to the data pages can be asynchronous because all the effects can be reconstructed from the log if necessary. The transaction management component coordinates logging, recovery, and buffer management. These topics are discussed later in this book; at this point, we'll just look briefly at transactions themselves.

The transaction management component delineates the boundaries of statements that must be grouped together to form an operation. It handles transactions that cross databases within the same SQL Server instance, and it allows nested transaction sequences. (However, nested transactions simply execute in the context of the first-level transaction; no special action occurs when they are committed. And a rollback specified in a lower level of a nested transaction undoes the entire transaction.) For a distributed transaction to another SQL Server instance (or to any other resource manager), the transaction management component coordinates with the Microsoft Distributed Transaction Coordinator (MS DTC) service using operating system remote procedure calls. The transaction management component marks *save points*—points you designate within a transaction at which work can be partially rolled back or undone.

The transaction management component also coordinates with the locking code regarding when locks can be released, based on the isolation level in effect. It also coordinates with the versioning code to determine when old versions are no longer needed and can be removed from the version store. The isolation level in which your transaction runs determines how sensitive your application is to changes made by others and consequently how long your transaction must hold locks or maintain versioned data to protect against those changes. SQL Server 2008 supports two concurrency models for guaranteeing the ACID properties of transactions: optimistic concurrency and pessimistic concurrency. Pessimistic concurrency guarantees correctness and consistency by locking data so that it cannot be changed; this is the concurrency model that every version of SQL Server prior to SQL Server 2005 used exclusively, and it is the default in both SQL Server

2005 and SQL Server 2008. SQL Server 2005 introduced optimistic concurrency, which provides consistent data by keeping older versions of rows with committed values in an area of *tempdb* called the *version store*. With optimistic concurrency, readers do not block writers and writers do not block readers, but writers still block writers. The cost of these nonblocking reads and writes must be considered. To support optimistic concurrency, SQL Server needs to spend more time managing the version store. In addition, administrators have to pay close attention to the *tempdb* database and plan for the extra maintenance it requires.

Five isolation-level semantics are available in SQL Server 2008. Three of them support only pessimistic concurrency: Read Uncommitted, Repeatable Read, and Serializable. Snapshot isolation level supports optimistic concurrency. The default isolation level, Read Committed, can support either optimistic or pessimistic concurrency, depending on a database setting.

The behavior of your transactions depends on the isolation level and the concurrency model you are working with. A complete understanding of isolation levels also requires an understanding of locking because the topics are so closely related. The next section gives an overview of locking; you'll find more detailed information on isolation, transactions, and concurrency management in Chapter 10.

Locking Operations Locking is a crucial function of a multiuser database system such as SQL Server, even if you are operating primarily in the Snapshot isolation level with optimistic concurrency. SQL Server lets you manage multiple users simultaneously and ensures that the transactions observe the properties of the chosen isolation level. Even though readers do not block writers and writers do not block readers in Snapshot isolation, writers do acquire locks and can still block other writers, and if two writers try to change the same data concurrently, a conflict occurs that must be resolved. The locking code acquires and releases various types of locks, such as share locks for reading, exclusive locks for writing, intent locks taken at a higher granularity to signal a potential "plan" to perform some operation, and extent locks for space allocation. It manages compatibility between the lock types, resolves deadlocks, and escalates locks if needed. The locking code controls table, page, and row locks as well as system data locks.

> **Note** Concurrency, with locks or row versions, is an important aspect of SQL Server. Many developers are keenly interested in it because of its potential effect on application performance. Chapter 10 is devoted to the subject, so we won't go into it further here.

Other Operations

Also included in the storage engine are components for controlling utilities such as bulk-load, DBCC commands, full-text index population and management, and backup and restore operations. DBCC is discussed in detail in Chapter 11, "DBCC Internals." The log manager makes sure that log records are written in a manner to guarantee transaction durability and recoverability; we'll go into detail about the transaction log and its role in backup and restore operations in Chapter 4.

The SQLOS

The SQLOS is a separate application layer at the lowest level of the SQL Server Database Engine, that both SQL Server and SQL Reporting Services run atop. Earlier versions of SQL Server have a thin layer of interfaces between the storage engine and the actual operating system through which SQL Server makes calls to the operating system for memory allocation, scheduler resources, thread and worker management, and synchronization objects. However, the services in SQL Server that needed to access these interfaces can be in any part of the engine. SQL Server requirements for managing memory, schedulers, synchronization objects, and so forth have become more complex. Rather than each part of the engine growing to support the increased functionality, a single application layer has been designed to manage all operating system resources that are specific to SQL Server.

The two main functions of SQLOS are scheduling and memory management, both of which we'll talk about in detail later in this section. Other functions of SQLOS include the following:

Synchronization Synchronization objects include spinlocks, mutexes, and special reader/ writer locks on system resources.

Memory Brokers Memory brokers distribute memory allocation between various components within SQL Server, but do not perform any allocations, which are handled by the Memory Manager.

SQL Server Exception Handling Exception handling involves dealing with user errors as well as system-generated errors.

Deadlock Detection The deadlock detection mechanism doesn't just involve locks, but checks for any tasks holding onto resources, that are mutually blocking each other. We'll talk about deadlocks involving locks (by far the most common kind) in Chapter 10.

Extended Events Tracking extended events is similar to the SQL Trace capability, but is much more efficient because the tracking runs at a much lower level than SQL Trace. In addition, because the extended event layer is so low and deep, there are many more types of events that can be tracked. The SQL Server 2008 Resource Governor manages resource usage using extended events. We'll talk about extended events in Chapter 2, "Change Tracking, Tracing, and Extended Events." (In a future version, all tracing will be handled at this level by extended events.)

Asynchronous IO The difference between asynchronous and synchronous is what part of the system is actually waiting for an unavailable resource. When SQL Server requests a synchronous I/O, if the resource is not available the Windows kernel will put the thread on a wait queue until the resource becomes available. For asynchronous I/O, SQL Server requests that Windows initiate an I/O. Windows starts the I/O operation and doesn't stop the thread from running. SQL Server will then place the server session in an I/O wait queue until it gets the signal from Windows that the resource is available.

NUMA Architecture

SQL Server 2008 is NUMA–aware, and both scheduling and memory management can take advantage of NUMA hardware by default. You can use some special configurations when you work with NUMA, so we'll provide some general background here before discussing scheduling and memory.

The main benefit of NUMA is scalability, which has definite limits when you use symmetric multiprocessing (SMP) architecture. With SMP, all memory access is posted to the same shared memory bus. This works fine for a relatively small number of CPUs, but problems appear when you have many CPUs competing for access to the shared memory bus. The trend in hardware has been to have more than one system bus, each serving a small set of processors. NUMA limits the number of CPUs on any one memory bus. Each group of processors has its own memory and possibly its own I/O channels. However, each CPU can access memory associated with other groups in a coherent way, and we'll discuss this a bit more later in the chapter. Each group is called a *NUMA node,* and the nodes are linked to each other by a high-speed interconnection. The number of CPUs within a NUMA node depends on the hardware vendor. It is faster to access local memory than the memory associated with other NUMA nodes. This is the reason for the name *Non-Uniform Memory Access*. Figure 1-3 shows a NUMA node with four CPUs.

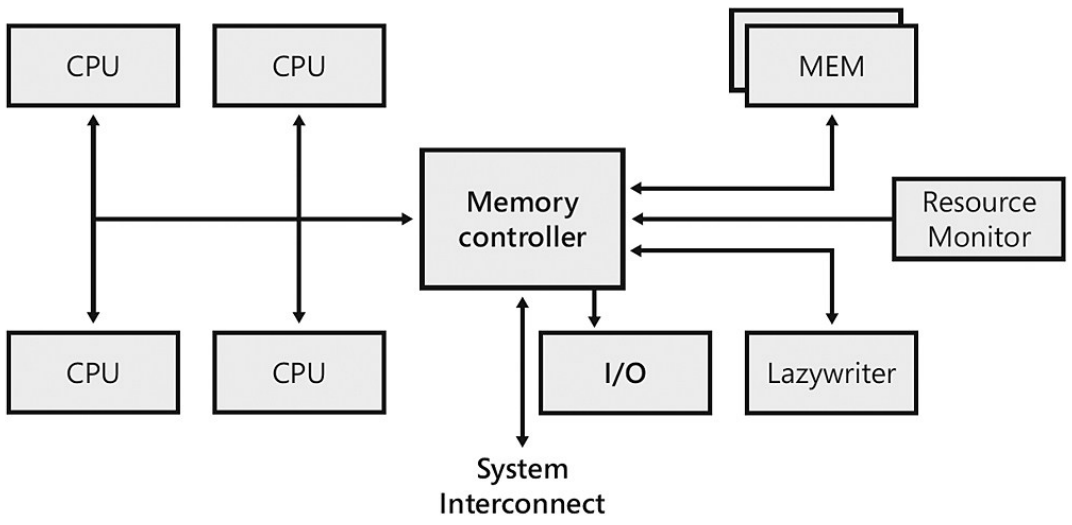


Figure 1-3: A NUMA node with four CPUs

SQL Server 2008 allows you to subdivide one or more physical NUMA nodes into smaller NUMA nodes, referred to as *software NUMA* or *soft-NUMA*. You typically use *soft-NUMA* when you have many CPUs and do not have hardware NUMA because soft-NUMA allows only for the subdividing of CPUs but not memory. You can also use soft-NUMA to subdivide hardware NUMA nodes into groups of fewer CPUs than is provided by the hardware NUMA. Your soft-NUMA nodes can also be configured to listen on their own ports.

Only the SQL Server scheduler and SNI are soft-NUMA–aware. Memory nodes are created based on hardware NUMA and are therefore not affected by soft-NUMA.

TCP/IP, VIA, Named Pipes, and shared memory can take advantage of NUMA round-robin scheduling, but only TCP and VIA can affinitize to a specific set of NUMA nodes. See *SQL Server Books Online* for how to use the SQL Server Configuration Manager to set a TCP/IP address and port to single or multiple nodes.

The Scheduler

Prior to SQL Server 7.0, scheduling depended entirely on the underlying Microsoft Windows operating system. Although this meant that SQL Server could take advantage of the hard work done by Windows engineers to enhance scalability and efficient processor use, there were definite limits. The Windows scheduler knew nothing about the needs of a relational database system, so it treated SQL Server worker threads the same as any other process running on the operating system. However, a high-performance system such as SQL Server functions best when the scheduler can meet its special needs. SQL Server 7.0 and all subsequent versions are designed to handle their own scheduling to gain a number of advantages, including the following:

- A private scheduler can support SQL Server tasks using fibers as easily as it supports using threads.
- Context switching and switching into kernel mode can be avoided as much as possible.

Note The scheduler in SQL Server 7.0 and SQL Server 2000 was called the *User Mode Scheduler* (UMS) to reflect the fact that it ran primarily in user mode, as opposed to kernel mode. SQL Server 2005 and 2008 call the scheduler the SOS Scheduler and improve on UMS even more.

One major difference between the SOS scheduler and the Windows scheduler is that the SQL Server scheduler runs as a cooperative rather than a preemptive scheduler. This means that it relies on the workers, threads, or fibers to yield voluntarily often enough so one process or thread doesn’t have exclusive control of the system. The SQL Server product team has to make sure that its code runs efficiently and voluntarily yields the scheduler in appropriate places; the reward for this is much greater control and scalability than is possible with the Windows scheduler.

Even though the scheduler is not preemptive, the SQL Server scheduler still adheres to a concept of a quantum. Instead of SQL Server tasks being forced to give up the CPU by the operating system, SQL Server tasks can request to be put on a wait queue periodically, and if they have exceeded the internally defined quantum, and they are not in the middle of an operation that cannot be stopped, they will voluntarily relinquish the CPU.

SQL Server Workers

You can think of the SQL Server scheduler as a logical CPU used by SQL Server workers. A worker can be either a thread or a fiber that is bound to a logical scheduler. If the Affinity Mask Configuration option is set, each scheduler is affinitized to a particular CPU. (We’ll talk about configuration later in this chapter.) Thus, each worker is also associated with a single CPU. Each scheduler is assigned a worker limit based on the configured Max Worker Threads and the number of schedulers, and each scheduler is responsible for creating or destroying workers as needed. A worker cannot move from one scheduler to another, but as workers are destroyed and created, it can appear as if workers are moving between schedulers.

Workers are created when the scheduler receives a request (a task to execute) and there are no idle workers. A worker can be destroyed if it has been idle for at least 15 minutes, or if SQL Server is under memory pressure. Each worker can use at least half a megabyte of memory on a 32-bit system and at least 2 MB on a 64-bit system, so destroying multiple workers and freeing their memory can yield an immediate performance improvement on memory-starved systems. SQL Server actually handles the worker pool very efficiently, and you might be surprised to know that even on very large systems with hundreds or even thousands of users, the actual number of SQL Server workers might be much lower than the configured value for Max Worker Threads. Later in this section, we’ll tell you about some of the Dynamic Management Objects that let you see how many workers you actually have, as well as scheduler and task information (discussed in the next section).

SQL Server Schedulers

In SQL Server 2008, each actual CPU (whether hyperthreaded or physical) has a scheduler created for it when SQL Server starts. This is true even if the affinity mask option has been configured so that SQL Server is set to not use all the available physical CPUs. In SQL Server 2008, each scheduler is set to either ONLINE or OFFLINE based on the affinity mask settings, and the default is that all schedulers are ONLINE. Changing the affinity mask value can change the status of one or more schedulers to OFFLINE, and you can do this without having to restart your SQL Server. Note that when a scheduler is switched from ONLINE to OFFLINE due to a configuration change, any work already assigned to the scheduler is first completed and no new work is assigned.

SQL Server Tasks

The unit of work for a SQL Server worker is a *request*, or a *task*, which you can think of as being equivalent to a single batch sent from the client to the server. Once a request is received by SQL Server, it is bound to a worker, and that worker processes the entire request before handling any other request. This holds true even if the request is blocked for some reason, such as while it waits for a lock or for I/O to complete. The particular worker does not handle any new requests but waits until the blocking condition is resolved and the request can be completed. Keep in mind that a session ID (SPID) is not the same as a task. A SPID is a connection or channel over which requests can be sent, but there is not always an active request on any particular SPID.

In SQL Server 2008, a SPID is not bound to a particular scheduler. Each SPID has a preferred scheduler, which is the scheduler that most recently processed a request from the SPID. The SPID is initially assigned to the scheduler with the lowest load. (You can get some insight into the load on each scheduler by looking at the *load_factor* column in the DMV *sys.dm_os_schedulers*.) However, when subsequent requests are sent from the same SPID, if another scheduler has a load factor that is less than a certain percentage of the average of the scheduler's entire load factor, the new task is given to the scheduler with the smallest load factor. There is a restriction that all tasks for one SPID must be processed by schedulers on the same NUMA node. The exception to this restriction is when a query is being executed as a parallel query across multiple CPUs. The optimizer can decide to use more CPUs that are available on the NUMA node processing the query, so other CPUs (and other schedulers) can be used.

Threads vs. Fibers

As mentioned earlier, the UMS was designed to work with workers running on either threads or fibers. Windows fibers have less overhead associated with them than threads do, and multiple fibers can run on a single thread. You can configure SQL Server to run in fiber mode by setting the Lightweight Pooling option to 1. Although using less overhead and a "lightweight" mechanism sounds like a good idea, you should evaluate the use of fibers carefully.

Certain components of SQL Server don't work, or don't work well, when SQL Server runs in fiber mode. These components include SQLMail and SQLXML. Other components, such as heterogeneous and CLR queries, are not supported at all in fiber mode because they need certain thread-specific facilities provided by Windows. Although it is possible for SQL Server to switch to thread mode to process requests that need it, the overhead might be greater than the overhead of using threads exclusively. Fiber mode was actually intended just for special niche situations in which SQL Server reaches a limit in scalability due to spending too much time switching between thread contexts or switching between user mode and kernel mode. In most environments, the performance benefit gained by fibers is quite small compared to the benefits you can get by tuning in other areas. If you're certain you have a situation that could benefit from fibers, be sure to test thoroughly before you set the option on a production server. In addition, you might even want to contact Microsoft Customer Support Services (*http://support.microsoft.com/ph/2855*) just to be certain.

NUMA and Schedulers

With a NUMA configuration, every node has some subset of the machine's processors and the same number of schedulers. If the machine is configured for hardware NUMA, the number of processors on each node will be preset, but for soft-NUMA that you configure yourself, you can decide how many processors are assigned to each node. There is still the same number of schedulers as processors, however. When SPIDs are first created, they are assigned to nodes on a round-robin basis. The Scheduler Monitor then assigns the SPID to the least loaded scheduler on that node. As mentioned earlier, if the SPID is moved to another scheduler, it stays on the same node. A single processor or SMP machine will be treated as a machine with a single NUMA node. Just like on an SMP machine, there is no hard mapping between schedulers and a CPU with NUMA, so any scheduler on an individual node can run on any CPU on that node. However, if you have set the Affinity Mask Configuration option, each scheduler on each node will be fixed to run on a particular CPU.

Every NUMA node has its own lazywriter (which we'll talk about in the section entitled "Memory," later in this chapter) as well as its own I/O Completion Port (IOCP), which is the network listener. Every node also has its own Resource Monitor, which is managed by a hidden scheduler. You can see the hidden schedulers in *sys.dm_os_schedulers*. Each Resource Monitor has its own SPID, which you can see by querying the *sys.dm_exec_requests* and *sys.dm_os_workers* DMVs, as shown here:

```
SELECT session_id,
    CONVERT (varchar(10), t1.status) AS status,
    CONVERT (varchar(20), t1.command) AS command,
    CONVERT (varchar(15), t2.state) AS worker_state
FROM sys.dm_exec_requests AS t1 JOIN sys.dm_os_workers AS t2
ON  t2.task_address = t1.task_address
```

```
WHERE command = 'RESOURCE MONITOR';
```

Every node has its own Scheduler Monitor, which can run on any SPID and runs in a preemptive mode. The Scheduler Monitor is a thread that wakes up periodically and checks each scheduler to see if it has yielded since the last time the Scheduler Monitor woke up (unless the scheduler is idle). The Scheduler Monitor raises an error (17883) if a nonidle thread has not yielded. The 17883 error can occur when an application other than SQL Server is monopolizing the CPU. The Scheduler Monitor knows only that the CPU is not yielding; it can't ascertain what kind of task is using it. The Scheduler Monitor is also responsible for sending messages to the schedulers to help them balance their workload.

Dynamic Affinity

In SQL Server 2008 (in all editions except SQL Server Express), processor affinity can be controlled dynamically. When SQL Server starts up, all scheduler tasks are started on server startup, so there is one scheduler per CPU. If the affinity mask has been set, some of the schedulers are then marked as offline and no tasks are assigned to them.

When the affinity mask is changed to include additional CPUs, the new CPU is brought online. The Scheduler Monitor then notices an imbalance in the workload and starts picking workers to move to the new CPU. When a CPU is brought offline by changing the affinity mask, the scheduler for that CPU continues to run active workers, but the scheduler itself is moved to one of the other CPUs that are still online. No new workers are given to this scheduler, which is now offline, and when all active workers have finished their tasks, the scheduler stops.

Binding Schedulers to CPUs

Remember that normally, schedulers are not bound to CPUs in a strict one-to-one relationship, even though there is the same number of schedulers as CPUs. A scheduler is bound to a CPU only when the affinity mask is set. This is true even if you specify that the affinity mask use all the CPUs, which is the default setting. For example, the default Affinity Mask Configuration value is 0, which means to use all CPUs, with no hard binding of scheduler to CPU. In fact, in some cases when there is a heavy load on the machine, Windows can run two schedulers on one CPU.

For an eight-processor machine, an affinity mask value of 3 (bit string 00000011) means that only CPUs 0 and 1 are used and two schedulers are bound to the two CPUs. If you set the affinity mask to 255 (bit string 11111111), all the CPUs are used, just as with the default. However, with the affinity mask set, the eight CPUs will be bound to the eight schedulers.

In some situations, you might want to limit the number of CPUs available but not bind a particular scheduler to a single CPU—for example, if you are using a multiple-CPU machine for server consolidation. Suppose that you have a 64-processor machine on which you are running eight SQL Server instances and you want each instance to use eight of the processors. Each instance has a different affinity mask that specifies a different subset of the 64 processors, so you might have affinity mask values 255 (0xFF), 65280 (0xFF00), 16711680 (0xFF0000), and 4278190080 (0xFF000000). Because the affinity mask is set, each instance has hard binding of scheduler to CPU. If you want to limit the number of CPUs but still not constrain a particular scheduler to running on a specific CPU, you can start SQL Server with trace flag 8002. This lets you have CPUs mapped to an instance, but within the instance, schedulers are not bound to CPUs.

Observing Scheduler Internals

SQL Server 2008 has several Dynamic Management Objects that provide information about schedulers, workers, and tasks. These are primarily intended for use by Microsoft Customer Support Services, but you can use them to gain a greater appreciation for the information that SQL Server monitors.

> **Note** All these objects (as well as most of the other Dynamic Management Objects) require a permission called *View Server State.* By default, only a SQL Server administrator has that permission, but it can be granted to others. For each of the objects, we will list some of the more useful or interesting columns and provide the description of each column taken from *SQL Server 2008 Books Online.* For the full list of columns, most of which are useful only to support personnel, you can refer to *SQL Server Books Online,* but even then, you'll find that some of the columns are listed as "for internal use only."

These Dynamic Management Objects are as follows:

sys.dm_os_schedulers This view returns one row per scheduler in SQL Server. Each scheduler is mapped to an individual processor in SQL Server. You can use this view to monitor the condition of a scheduler or to identify runaway tasks. Interesting columns include the following:

parent_node_id The ID of the node that the scheduler belongs to, also known as the *parent node*. This represents a NUMA node.

scheduler_id The ID of the scheduler. All schedulers that are used to run regular queries have IDs of less than 255. Those with IDs greater than or equal to 255, such as the dedicated administrator connection scheduler, are used internally by SQL Server.

cpu_id The ID of the CPU with which this scheduler is associated. If SQL Server is configured to run with affinity, the value is the ID of the CPU on which the scheduler is supposed to run. If the affinity mask has not been specified, the *cpu_id* will be 255.

is_online If SQL Server is configured to use only some of the available processors on the server, this can mean that some schedulers are mapped to processors that are not in the affinity mask. If that is the case, this column returns 0. This means the scheduler is not being used to process queries or batches.

current_tasks_count The number of current tasks associated with this scheduler, including the following. (When a task is completed, this count is decremented.)

- Tasks that are waiting on a resource to be acquired before proceeding
- Tasks that are currently running or that are runnable and waiting to be executed

runnable_tasks_count The number of tasks waiting to run on the scheduler.

current_workers_count The number of workers associated with this scheduler, including workers that are not assigned any task.

active_workers_count The number of workers that have been assigned a task.

work_queue_count The number of tasks waiting for a worker. If *current_workers_count* is greater than *active_workers_count*, this work queue count should be 0 and the work queue should not grow.

pending_disk_io_count The number of pending I/Os. Each scheduler has a list of pending I/Os that are checked every time there is a context switch to determine whether they have been completed. The count is incremented when the request is inserted. It is decremented when the request is completed. This number does not indicate the state of the I/Os.

load_factor The internal value that indicates the perceived load on this scheduler. This value is used to determine whether a new task should be put on this scheduler or another scheduler. It is useful for debugging purposes when schedulers appear not to be evenly loaded. In SQL Server 2000, a task is routed to a particular scheduler. In SQL Server 2008, the routing decision is based on the load on the scheduler. SQL Server 2008 also uses a load factor of nodes and schedulers to help determine the best location to acquire resources. When a task is added to the queue, the load factor increases. When a task is completed, the load factor decreases. Using load factors helps the SQLOS balance the work load better.

sys.dm_os_workers This view returns a row for every worker in the system. Interesting columns include the following:

is_preemptive A value of 1 means that the worker is running with preemptive scheduling. Any worker running external code is run under preemptive scheduling.

is_fiber A value of 1 means that the worker is running with lightweight pooling.

sys.dm_os_threads This view returns a list of all SQLOS threads that are running under the SQL Server process. Interesting columns include the following:

started_by_sqlserver Indicates the thread initiator. A 1 means that SQL Server started the thread and 0 means that another component, such as an extended procedure from within SQL Server, started the thread.

creation_time The time when this thread was created.

stack_bytes_used The number of bytes that are actively being used on the thread.

affinity The CPU mask on which this thread is supposed to be running. This depends on the value in the *sp_configure* “affinity mask.”

locale The cached locale LCID for the thread.

sys.dm_os_tasks This view returns one row for each task that is active in the instance of SQL Server. Interesting columns include the following:

task_state The state of the task. The value can be one of the following:

- PENDING: Waiting for a worker thread
- RUNNABLE: Runnable but waiting to receive a quantum
- RUNNING: Currently running on the scheduler
- SUSPENDED: Has a worker but is waiting for an event
- DONE: Completed
- SPINLOOP: Processing a spinlock, as when waiting for a signal

context_switches_count The number of scheduler context switches that this task has completed.

pending_io_count The number of physical I/Os performed by this task.

pending_io_byte_count The total byte count of I/Os performed by this task.

pending_io_byte_average The average byte count of I/Os performed by this task.

scheduler_id The ID of the parent scheduler. This is a handle to the scheduler information for this task.

session_id The ID of the session associated with the task.

sys.dm_os_waiting_tasks This view returns information about the queue of tasks that are waiting on some resource. Interesting columns include the following:

session_id The ID of the session associated with the task.

exec_context_id The ID of the execution context associated with the task.

wait_duration_ms The total wait time for this wait type, in milliseconds. This time is inclusive of *signal_wait_time.*

wait_type The name of the wait type.

resource_address The address of the resource for which the task is waiting.

blocking_task_address The task that is currently holding this resource.

blocking_session_id The ID of the session of the blocking task.

blocking_exec_context_id The ID of the execution context of the blocking task.

resource_description The description of the resource that is being consumed.

The Dedicated Administrator Connection (DAC)

Under extreme conditions such as a complete lack of available resources, it is possible for SQL Server to enter an abnormal state in which no further connections can be made to the SQL Server instance. Prior to SQL Server 2005, this situation meant that an administrator could not get in to kill any troublesome connections or even begin to diagnose the possible cause of the problem. SQL Server 2005 introduced a special connection called the *DAC* that was designed to be accessible even when no other access can be made.

Access via the DAC must be specially requested. You can connect to the DAC using the command-line tool SQLCMD, and specifying the *-A* (or */A*) flag. This method of connection is recommended because it uses fewer resources than the graphical user interface (GUI).

Through Management Studio, you can specify that you want to connect using DAC by preceding the name of your SQL Server with *ADMIN:* in the Connection dialog box.

For example, to connect to the default SQL Server instance on my machine, TENAR, we would enter **ADMIN:TENAR**. To connect to a named instance called SQL2008 on the same machine, we would enter **ADMIN:TENAR\SQL2008**.

The DAC is a special-purpose connection designed for diagnosing problems in SQL Server and possibly resolving them. It is not meant to be used as a regular user connection. Any attempt to connect using the DAC when there is already an active DAC connection results in an error. The message returned to the client says only that the connection was rejected; it does not state explicitly that it was because there already was an active DAC. However, a message is written to the error log indicating the attempt (and failure) to get a second DAC connection. You can check whether a DAC is in use by running the following query. If there is an active DAC, the query will return the SPID for the DAC; otherwise, it will return no rows.

```
SELECT s.session_id
FROM sys.tcp_endpoints as e JOIN sys.dm_exec_sessions as s
   ON e.endpoint_id = s.endpoint_id
WHERE e.name='Dedicated Admin Connection';
```

You should keep the following points in mind about using the DAC:

- By default, the DAC is available only locally. However, an administrator can configure SQL Server to allow remote connection by using the configuration option called *Remote Admin Connections.*
- The user logon to connect via the DAC must be a member of the *sysadmin* server role.
- There are only a few restrictions on the SQL statements that can be executed on the DAC. (For example, you cannot run *BACKUP* or *RESTORE* using the DAC.) However, it is recommended that you do not run any resource-intensive queries that might exacerbate the problem that led you to use the DAC. The DAC connection is created primarily for troubleshooting and diagnostic purposes. In general, you'll use the DAC for running queries against the Dynamic Management Objects, some of which you've seen already and many more of which we'll discuss later in this book.
- A special thread is assigned to the DAC that allows it to execute the diagnostic functions or queries on a separate scheduler. This thread cannot be terminated. You can kill only the DAC session, if needed. The DAC scheduler always uses the *scheduler_id* value of 255, and this thread has the highest priority. There is no lazywriter thread for the DAC, but the DAC does have its own IOCP, a worker thread, and an idle thread.

You might not always be able to accomplish your intended tasks using the DAC. Suppose you have an idle connection that is holding on to a lock. If the connection has no active task, there is no thread associated with it, only a connection ID. Suppose further that many other processes are trying to get access to the locked resource, and that they are blocked. Those connections still have an incomplete task, so they do not release their worker. If 255 such processes (the default number of worker threads) try to get the same lock, all available workers might get used up and no more connections can be made to SQL Server. Because the DAC has its own scheduler, you can start it, and the expected solution would be to kill the connection that is holding the lock but not do any further processing to release the lock. But if you try to use the DAC to kill the process holding the lock, the attempt fails. SQL Server would need to give a worker to the task to kill it, and no workers are available. The only solution is to kill several of the (blameless) blocked processes that still have workers associated with them.

Note To conserve resources, SQL Server 2008 Express edition does not support a DAC connection unless started with a trace flag 7806.

The DAC is not guaranteed to always be usable, but because it reserves memory and a private scheduler and is implemented as a separate node, a connection probably is possible when you cannot connect in any other way.

Memory

Memory management is a huge topic, and to cover every detail of it would require a whole book in itself. My goal in this section is twofold: first, to provide enough information about how SQL Server uses its memory resources so you can determine whether memory is being managed well on your system; and second, to describe the aspects of memory management that you have control over so you can understand when to exert that control.

By default, SQL Server 2008 manages its memory resources almost completely dynamically. When allocating memory, SQL Server must communicate constantly with the operating system, which is one of the reasons the SQLOS layer of the engine is so important.

The Buffer Pool and the Data Cache

The main memory component in SQL Server is the buffer pool. All memory not used by another memory component remains in the buffer pool to be used as a data cache for pages read in from the database files on disk. The buffer manager manages disk I/O functions for bringing data and index pages into the data cache so data can be shared among users. When other components require memory, they can request a buffer from the buffer pool. A buffer is a page in memory that's the same size as a data or index page. You can think of it as a page frame that can hold one page from a database. Most of the buffers taken from the buffer pool for other memory components go to other kinds of memory caches, the largest of which is typically the cache for procedure and query plans, which is usually called the *plan cache*.

Occasionally, SQL Server must request contiguous memory in larger blocks than the 8-KB pages that the buffer pool can provide, so memory must be allocated from outside the buffer pool. Use of large memory blocks is typically kept to a minimum, so direct calls to the operating system account for a small fraction of SQL Server memory usage.

Access to In-Memory Data Pages

Access to pages in the data cache must be fast. Even with real memory, it would be ridiculously inefficient to scan the whole data cache for a page when you have gigabytes of data. Pages in the data cache are therefore hashed for fast access. *Hashing* is a technique that uniformly maps a key via a hash function across a set of hash buckets. A *hash table* is a structure in memory that contains an array of pointers (implemented as a linked list) to the buffer pages. If all the pointers to buffer pages do not fit on a single hash page, a *linked list* chains to additional hash pages.

Given a *dbid-fileno-pageno* identifier (a combination of the database ID, file number, and page number), the hash function converts that key to the hash bucket that should be checked; in essence, the hash bucket serves as an index to the specific page needed. By using hashing, even when large amounts of memory are present, SQL Server can find a specific data page in cache with only a few memory reads. Similarly, it takes only a few memory reads for SQL Server to determine that a desired page is not in cache and that it must be read in from disk.

> **Note** Finding a data page might require that multiple buffers be accessed via the hash buckets chain (linked list). The hash function attempts to uniformly distribute the *dbid-fileno-pageno* values throughout the available hash buckets. The number of hash buckets is set internally by SQL Server and depends on the total size of the buffer pool.

Managing Pages in the Data Cache

You can use a data page or an index page only if it exists in memory. Therefore, a buffer in the data cache must be available for the page to be read into. Keeping a supply of buffers available for immediate use is an important performance optimization. If a buffer isn't readily available, many memory pages might have to be searched simply to locate a buffer to free up for use as a workspace.

In SQL Server 2008, a single mechanism is responsible both for writing changed pages to disk and for marking as free those pages that have not been referenced for some time. SQL Server maintains a linked list of the addresses of free pages, and any worker needing a buffer page uses the first page of this list.

Every buffer in the data cache has a header that contains information about the last two times the page was referenced and some status information, including whether the page is dirty (that is, it has been changed since it was read into disk). The reference information is used to implement the page replacement policy for the data cache pages, which uses an algorithm called *LRU-K*, which was introduced by Elizabeth O'Neil, Patrick O'Neil, and Gerhard Weikum (in the Proceedings of the ACM SIGMOD Conference, May 1993). This algorithm is a great improvement over a strict Least Recently Used (LRU) replacement policy, which has no knowledge of how recently a page was used. It is also an improvement over a Least Frequently Used (LFU) policy involving reference counters because it requires far fewer adjustments by the engine and much less bookkeeping overhead. An LRU-K algorithm keeps track of the last *K* times a page was referenced and can differentiate between types of pages, such as index and data pages, with different levels of frequency. It can actually simulate the effect of assigning pages to different buffer pools of specifically tuned sizes. SQL Server 2008 uses a *K* value of 2, so it keeps track of the two most recent accesses of each buffer page.

The data cache is periodically scanned from the start to the end. Because the buffer cache is all in memory, these scans are quick and require no I/O. During the scan, a value is associated with each buffer based on its usage history. When the value gets low enough, the dirty page indicator is checked. If the page is dirty, a write is scheduled to write the modifications to disk. Instances of SQL Server use a write-ahead log so the write of the dirty data page is blocked while the log page recording the modification is first written to disk. (We'll discuss logging in much more detail in Chapter 4.) After the modified page has been flushed to disk, or if the page was not dirty to start with, the page is freed. The association between the buffer page and the data page that it contains is removed by deleting information about the buffer from the hash table, and the buffer is put on the free list.

Using this algorithm, buffers holding pages that are considered more valuable remain in the active buffer pool whereas buffers holding pages not referenced often enough eventually return to the free buffer list. The instance of SQL Server determines internally the size of the free buffer list, based on the size of the buffer cache. The size cannot be configured.

The Free Buffer List and the Lazywriter

The work of scanning the buffer pool, writing dirty pages, and populating the free buffer list is primarily performed by the individual workers after they have scheduled an asynchronous read and before the read is completed. The worker gets the address of a section of the buffer pool containing 64 buffers from a central data structure in the SQL Server Database Engine. Once the read has been initiated, the worker checks to see whether the free list is too small. (Note that this process has consumed one or more pages of the list for its own read.) If so, the worker searches for buffers to free up, examining all 64 buffers, regardless of how many it actually finds to free up in that group of 64. If a write must be performed for a dirty buffer in the scanned section, the write is also scheduled.

Each instance of SQL Server also has a thread called *lazywriter* for each NUMA node (and every instance has at least one) that scans through the buffer cache associated with that node. The lazywriter thread sleeps for a specific interval of time, and when it wakes up, it examines the size of the free buffer list. If the list is below a certain threshold, which depends on the total size of the buffer pool, the lazywriter thread scans the buffer pool to repopulate the free list. As buffers are added to the free list, they are also written to disk if they are dirty.

When SQL Server uses memory dynamically, it must constantly be aware of the amount of free memory. The lazywriter for each node queries the system periodically to determine the amount of free physical memory available. The lazywriter expands or shrinks the data cache to keep the operating system's free physical memory at 5 MB (plus or minus 200 KB) to prevent paging. If the operating system has less than 5 MB free, the lazywriter releases memory to the operating system instead of adding it to the free list. If more than 5 MB of physical memory is free, the lazywriter recommits memory to the buffer pool by adding it to the free list. The lazywriter recommits memory to the buffer pool only when it repopulates the free list; a server at rest does not grow its buffer pool.

SQL Server also releases memory to the operating system if it detects that too much paging is taking place. You can tell when SQL Server increases or decreases its total memory use by using one of SQL Server's tracing mechanisms to monitor Server Memory Change events (in the Server Event category). An event is generated whenever memory in SQL Server increases or decreases by 1 MB or 5 percent of the maximum server memory, whichever is greater. You can look at the value of the data element, called *Event Sub Class,* to see whether the change was an increase or a decrease. An *Event Sub Class* value of 1 means a memory increase; a value of 2 means a memory decrease. Tracing will be covered in detail in Chapter 2.

Checkpoints

The checkpoint process also scans the buffer cache periodically and writes any dirty data pages for a particular database to disk. The difference between the checkpoint process and the lazywriter (or the worker threads' management of pages) is that the checkpoint process never puts buffers on the free list. The only purpose of the checkpoint process is to ensure that pages written before a certain time are written to disk, so that the number of dirty pages in memory is always kept to a minimum, which in turn ensures that the length of time SQL Server requires for recovery of a database after a failure is kept to a minimum. In some cases, checkpoints may find few dirty pages to write to disk if most of the dirty pages have been written to disk by the workers or the lazywriters in the period between two checkpoints.

When a checkpoint occurs, SQL Server writes a checkpoint record to the transaction log, which lists all the transactions that are active. This allows the recovery process to build a table containing a list of all the potentially dirty pages. Checkpoints occur automatically at regular intervals but can also be requested manually.

Checkpoints are triggered when any of the following occurs:

- A database owner (or backup operator) explicitly issues a *CHECKPOINT* command to perform a checkpoint in that database. In SQL Server 2008, you can run multiple checkpoints (in different databases) concurrently by using the *CHECKPOINT* command.

- The log is getting full (more than 70 percent of capacity) and the database is in autotruncate mode. (We'll tell you about autotruncate mode in Chapter 4.) A checkpoint is triggered to truncate the transaction log and free up space. However, if no space can be freed up, perhaps because of a long-running transaction, no checkpoint occurs.
- A long recovery time is estimated. When recovery time is predicted to be longer than the Recovery Interval configuration option, a checkpoint is triggered. SQL Server 2008 uses a simple metric to predict recovery time because it can recover, or redo, in less time than it took the original operations to run. Thus, if checkpoints are taken about as often as the recovery interval frequency, recovery completes within the interval. A recovery interval setting of 1 means that checkpoints occur about every minute so long as transactions are being processed in the database. A minimum amount of work must be done for the automatic checkpoint to fire; this is currently 10 MB of logs per minute. In this way, SQL Server doesn't waste time taking checkpoints on idle databases. A default recovery interval of 0 means that SQL Server chooses an appropriate value; for the current version, this is one minute.
- An orderly shutdown of SQL Server is requested, without the NOWAIT option. A checkpoint operation is then run in each database on the instance. An orderly shutdown occurs when you explicitly shut down SQL Server, unless you do so by using the *SHUTDOWN WITH NOWAIT* command. An orderly shutdown also occurs when the SQL Server service is stopped through Service Control Manager or the net stop command from an operating system prompt.

You can also use the *sp_configure* Recovery Interval option to influence checkpointing frequency, balancing the time to recover vs. any impact on run-time performance. If you're interested in tracing when checkpoints actually occur, you can use the SQL Server extended events *sqlserver.checkpoint_begin* and *sqlserver.checkpoint_end* to monitor checkpoint activity. (Details on extended events can be found in Chapter 2.)

The checkpoint process goes through the buffer pool, scanning the pages in a nonsequential order, and when it finds a dirty page, it looks to see whether any physically contiguous (on the disk) pages are also dirty so that it can do a large block write. But this means that it might, for example, write buffers 14, 200, 260, and 1,000 when it sees that buffer 14 is dirty. (Those pages might have contiguous disk locations even though they're far apart in the buffer pool. In this case, the noncontiguous pages in the buffer pool can be written as a single operation called a *gather-write*.) The process continues to scan the buffer pool until it gets to page 1,000. In some cases, an already written page could potentially be dirty again, and it might need to be written out to disk a second time.

The larger the buffer pool, the greater the chance that a buffer that has already been written will be dirty again before the checkpoint is done. To avoid this, SQL Server uses a bit associated with each buffer called a *generation number*. At the beginning of a checkpoint, all the bits are toggled to the same value, either all 0's or all 1's. As a checkpoint checks a page, it toggles the generation bit to the opposite value. When the checkpoint comes across a page whose bit has already been toggled, it doesn't write that page. Also, any new pages brought into cache during the checkpoint process get the new generation number so they won't be written during that checkpoint cycle. Any pages already written because they're in proximity to other pages (and are written together in a gather write) aren't written a second time.

In some cases checkpoints may issue a substantial amount of I/O, causing the I/O subsystem to get inundated with write requests which can severely impact read performance. On the other hand, there may be periods of relatively low I/O activity that could be utilized. SQL Server 2008 includes a command-line option that allows throttling of checkpoint I/Os. You can use the SQL Server Configuration Manager, and add the *–k* parameter, followed by a decimal number, to the list of startup parameters for the SQL Server service. The value specified indicates the number of megabytes per second that the checkpoint process can write. By using this *–k* option, the I/O overhead of checkpoints can be spread out and have a more measured impact. Remember that by default, the checkpoint process makes sure that SQL Server can recover databases within the recovery interval that you specify. If you enable this option, the default behavior changes, resulting in a long recovery time if you specify a very low value for the parameter. Backups may take a slightly longer time to finish because a checkpoint process that a backup initiates is also delayed. Before enabling this option on a production system, you should make sure that you have enough hardware to sustain the I/O requests that are posted by SQL Server and that you have thoroughly tested your applications on the system.

Managing Memory in Other Caches

Buffer pool memory that isn't used for the data cache is used for other types of caches, primarily the plan cache. The page replacement policy, as well as the mechanism by which freeable pages are searched for, are quite a bit different than for the data cache.

SQL Server 2008 uses a common caching framework that is used by all caches except the data cache. The framework consists of a set of stores and the Resource Monitor. There are three types of stores: cache stores, user stores (which don't actually have anything to do with users), and object stores. The plan cache is the main example of a cache store, and the metadata cache is the prime example of a user store. Both cache stores and user stores use the same LRU mechanism and the same costing algorithm to determine which pages can stay and which can be freed. Object stores, on the other hand, are just pools of memory blocks and don't require LRU or costing. One example of the use of an object store is the SNI, which uses the object store for pooling network buffers. For the rest of this section, my discussion of stores refers only to cache stores and user stores.

The LRU mechanism used by the stores is a straightforward variation of the clock algorithm. Imagine a clock hand sweeping through the store, looking at every entry; as it touches each entry, it decreases the cost. Once the cost of an entry reaches 0, the entry can be removed from the cache. The cost is reset whenever an entry is reused.

Memory management in the stores takes into account both global and local memory management policies. Global policies consider the total memory on the system and enable the running of the clock algorithm across all the caches. Local policies involve looking at one store or cache in isolation and making sure it is not using a disproportionate amount of memory.

To satisfy global and local policies, the SQL Server stores implement two hands: external and internal. Each store has two clock hands, and you can observe these by examining the DMV *sys.dm_os_memory_cache_clock_hands*. This view contains one internal and one external clock hand for each cache store or user store. The external clock hands implement the global policy, and the internal clock hands implement the local policy. The Resource Monitor is in charge of moving the external hands whenever it notices memory pressure. There are many types of memory pressure, and it is beyond the scope of this book to go into all the details of detecting and troubleshooting memory problems. However, if you take a look at the DMV *sys.dm_os_memory_cache_clock_hands*, specifically at the *removed_last_round_count* column, you can look for a value that is very large compared to other values. If you notice that value increasing dramatically, that is a strong indication of memory pressure. The companion Web site for this book contains a comprehensive white paper called "Troubleshooting Performance Problems in SQL Server 2008," which includes many details on tracking down and dealing with memory problems.

The internal clock moves whenever an individual cache needs to be trimmed. SQL Server attempts to keep each cache reasonably sized compared to other caches. The internal clock hands move only in response to activity. If a worker running a task that accesses a cache notices a high number of entries in the cache or notices that the size of the cache is greater than a certain percentage of memory, the internal clock hand for that cache starts to free up memory for that cache.

The Memory Broker

Because memory is needed by so many components in SQL Server, and to make sure each component uses memory efficiently, SQL Server uses a Memory Broker, whose job is to analyze the behavior of SQL Server with respect to memory consumption and to improve dynamic memory distribution. The Memory Broker is a centralized mechanism that dynamically distributes memory between the buffer pool, the query executor, the Query Optimizer, and all the various caches, and it attempts to adapt its distribution algorithm for different types of workloads. You can think of the Memory Broker as a control mechanism with a feedback loop. It monitors memory demand and consumption by component, and it uses the information that it gathers to calculate the optimal memory distribution across all components. It can broadcast this information to the component, which then uses the information to adapt its memory usage. You can monitor Memory Broker behavior by querying the Memory Broker ring buffer as follows:

```
SELECT * FROM sys.dm_os_ring_buffers
WHERE ring_buffer_type =
'RING_BUFFER_MEMORY_BROKER';
```

The ring buffer for the Memory Broker is updated only when the Memory Broker wants the behavior of a given component to change—that is, to grow, shrink, or remain stable (if it has previously been growing or shrinking).

Sizing Memory

When we talk about SQL Server memory, we are actually talking about more than just the buffer pool. SQL Server memory is actually organized into three sections, and the buffer pool is usually the largest and most frequently used. The buffer pool is used as a set of 8-KB buffers, so any memory that is needed in chunks larger than 8 KB is managed separately.

The DMV called *sys.dm_os_memory_clerks* has a column called *multi_pages_kb* that shows how much space is used by a memory component outside the buffer pool:

```
SELECT type, sum(multi_pages_kb)
FROM sys.dm_os_memory_clerks
WHERE multi_pages_kb != 0
GROUP BY type;
```

If your SQL Server instance is configured to use Address Windowing Extensions (AWE) memory, that can be considered a third memory area. AWE is an API that allows a 32-bit application to access physical memory beyond the 32-bit address limit. Although AWE memory is measured as part of the buffer pool, it must be kept track of separately because only data cache pages can use AWE memory. None of the other memory components, such as the plan cache, can use AWE memory.

Note If AWE is enabled, the only way to get information about the actual memory consumption of SQL Server is by using SQL Server–specific counters or DMVs inside the server; you won't get this information from operating system–level performance counters.

Sizing the Buffer Pool

When SQL Server starts, it computes the size of the virtual address space (VAS) of the SQL Server process. Each process running on Windows has its own VAS. The set of all virtual addresses available for process use constitutes the size of the VAS. The size of the VAS depends on the architecture (32- or 64-bit) and the operating system. VAS is just the set of all possible addresses; it might be much greater than the physical memory on the machine.

A 32-bit machine can directly address only 4 GB of memory and, by default, Windows itself reserves the top 2 GB of address space for its own use, which leaves only 2 GB as the maximum size of the VAS for any application, such as SQL Server. You can increase this by enabling a */3GB* flag in the system's Boot.ini file, which allows applications to have a VAS of up to 3 GB. If your system has more than 3 GB of RAM, the only way a 32-bit machine can get to it is by enabling AWE. One benefit of using AWE in SQL Server 2008 is that memory pages allocated through the AWE mechanism are considered locked pages and can never be swapped out.

On a 64-bit platform, the AWE Enabled configuration option is present, but its setting is ignored. However, the Windows policy option Lock Pages in Memory is available, although it is disabled by default. This policy determines which accounts can make use of a Windows feature to keep data in physical memory, preventing the system from paging the data to virtual memory on disk. It is recommended that you enable this policy on a 64-bit system.

On 32-bit operating systems, you have to enable the Lock Pages in Memory option when using AWE. It is recommended that you don't enable the Lock Pages in Memory option if you are not using AWE. Although SQL Server ignores this option when AWE is not enabled, other processes on the system may be affected.

Note Memory management is much more straightforward on a 64-bit machine, both for SQL Server, which has so much more VAS to work with, and for an administrator, who doesn't have to worry about special operating system flags or even whether to enable AWE. Unless you are working only with very small databases and do not expect to need more than a couple of gigabytes of RAM, you should definitely consider running a 64-bit edition of SQL Server 2008.

Table 1-1 shows the possible memory configurations for various editions of SQL Server 2008.

Table 1-1: SQL Server 2008 Memory Configurations

Configuration	VAS	Maximum Physical Memory	AWE/Locked Pages Support
Native 32-bit on 32-bit operating system with */3GB* boot parameter	2 GB	64 GB	AWE
	3 GB	16 GB	AWE
32-bit on x64 operating system (Windows on Windows)	4 GB	64 GB	AWE
Native 64-bit on x64 operating system	8 terabyte	1 terabyte	Locked Pages
Native 64-bit on IA64 operating system	7 terabyte	1 terabyte	Locked Pages

In addition to the VAS size, SQL Server also calculates a value called *Target Memory,* which is the number of 8-KB pages that it expects to be able to allocate. If the configuration option Max Server Memory has been set, Target Memory is the lesser of these two values. Target Memory is recomputed periodically, particularly when it gets a memory notification from Windows. A decrease in the number of target pages on a normally loaded server might indicate a response to external physical memory pressure. You can see the number of target pages by using the Performance Monitor—examine the Target Server Pages counter in the *SQL Server: Memory Manager* object. There is also a DMV called *sys.dm_os_sys_info* that contains one row of general-purpose SQL Server configuration information, including the following columns:

physical_memory_in_bytes The amount of physical memory available.

virtual_memory_in_bytes The amount of virtual memory available to the process in user mode. You can use this value to determine whether SQL Server was started by using a 3-GB switch.

bpool_commited The total number of buffers with pages that have associated memory. This does not include virtual memory.

bpool_commit_target The optimum number of buffers in the buffer pool.

bpool_visible The number of 8-KB buffers in the buffer pool that are directly accessible in the process virtual address space. When not using AWE, when the buffer pool has obtained its memory target (*bpool_committed* = *bpool_commit_target*), the value of *bpool_visible* equals the value of *bpool_committed*. When using AWE on a 32-bit version of SQL Server, *bpool_visible* represents the size of the AWE mapping window used to access physical memory allocated by the buffer pool. The size of this mapping window is bound by the process address space and, therefore, the visible amount will be smaller than the committed amount and can be reduced further by internal components consuming memory for purposes other than database pages. If the value of *bpool_visible* is too low, you might receive out-of-memory errors.

Although the VAS is reserved, the physical memory up to the target amount is committed only when that memory is required for the current workload that the SQL Server instance is handling. The instance continues to acquire physical memory as needed to support the workload, based on the users connecting and the requests being processed. The SQL Server instance can continue to commit physical memory until it reaches its target or the operating system indicates that there is no more free memory. If SQL Server is notified by the operating system that there is a shortage of free memory, it frees up memory if it has more memory than the configured value for Min Server Memory. Note that SQL Server does not commit memory equal to Min Server Memory initially. It commits only what it needs and what the operating system can afford. The value for Min Server Memory comes into play only after the buffer pool size goes above that amount, and then SQL Server does not let memory go below that setting.

As other applications are started on a computer running an instance of SQL Server, they consume memory, and SQL Server might need to adjust its target memory. Normally, this should be the only situation in which target memory is less than commit memory, and it should stay that way only until memory can be released. The instance of SQL Server adjusts its memory consumption, if possible. If another application is stopped and more memory becomes available, the instance of SQL Server increases the value of its target memory, allowing the memory allocation to grow when needed. SQL Server adjusts its target and releases physical memory only when there is pressure to do so. Thus, a server that is busy for a while can commit large amounts of memory that will not necessarily be released if the system becomes quiescent.

Note There is no special handling of multiple SQL Server instances on the same machine; there is no attempt to balance memory across all instances. They all compete for the same physical memory, so to make sure none of the instances becomes starved for physical memory, you should use the Min and Max Server Memory option on all SQL Server instances on a multiple-instance machine.

Observing Memory Internals

SQL Server 2008 includes several Dynamic Management Objects that provide information about memory and the various caches. Like the Dynamic Management Objects containing information about the schedulers, these objects are intended primarily for use by Customer Support Services to see what SQL Server is doing, but you can use them for the same purpose. To select from these objects, you must have the View Server State permission. Once again, we will list some of the more useful or interesting columns for each object; most of these descriptions are taken from *SQL Server Books Online:*

sys.dm_os_memory_clerks This view returns one row per memory clerk that is currently active in the instance of SQL Server. You can think of a clerk as an accounting unit. Each store described earlier is a clerk, but some clerks are not stores, such as those for the CLR and for full-text search. The following query returns a list of all the types of clerks:

```
SELECT DISTINCT type FROM sys.dm_os_memory_clerks;
```

Interesting columns include the following:

single_pages_kb The amount of single-page memory allocated, in kilobytes. This is the amount of memory allocated by using the single-page allocator of a memory node. This single-page allocator steals pages directly from the buffer pool.

multi_pages_kb The amount of multiple-page memory allocated, in kilobytes. This is the amount of memory allocated by using the multiple-page allocator of the memory nodes. This memory is allocated outside the buffer pool and takes advantage of the virtual allocator of the memory nodes.

virtual_memory_reserved_kb The amount of virtual memory reserved by a memory clerk. This is the amount of memory reserved directly by the component that uses this clerk. In most situations, only the buffer pool reserves VAS directly by using its memory clerk.

virtual_memory_committed_kb The amount of memory committed by the clerk. The amount of committed memory should always be less than the amount of Reserved Memory.

awe_allocated_kb The amount of memory allocated by the memory clerk by using AWE. In SQL Server, only buffer pool clerks (MEMORYCLERK_SQLBUFFERPOOL) use this mechanism, and only when AWE is enabled.

sys.dm_os_memory_cache_counters This view returns a snapshot of the health of each cache of type userstore and cachestore. It provides run-time information about the cache entries allocated, their use, and the source of memory for the cache entries. Interesting columns include the following:

single_pages_kb The amount of single-page memory allocated, in kilobytes. This is the amount of memory allocated by using the single-page allocator. This refers to the 8-KB pages that are taken directly from the buffer pool for this cache.

multi_pages_kb The amount of multiple-page memory allocated, in kilobytes. This is the amount of memory allocated by using the multiple-page allocator of the memory node. This memory is allocated outside the buffer pool and takes advantage of the virtual allocator of the memory nodes.

multi_pages_in_use_kb The amount of multiple-page memory being used, in kilobytes.

single_pages_in_use_kb The amount of single-page memory being used, in kilobytes.

entries_count The number of entries in the cache.

entries_in_use_count The number of entries in use in the cache.

sys.dm_os_memory_cache_hash_tables This view returns a row for each active cache in the instance of SQL Server. This view can be joined to *sys.dm_os_memory_cache_counters* on the *cache_address* column. Interesting columns include the following:

buckets_count The number of buckets in the hash table.

buckets_in_use_count The number of buckets currently being used.

buckets_min_length The minimum number of cache entries in a bucket.

buckets_max_length The maximum number of cache entries in a bucket.

buckets_avg_length The average number of cache entries in each bucket. If this number gets very large, it might indicate that the hashing algorithm is not ideal.

buckets_avg_scan_hit_length The average number of examined entries in a bucket before the searched-for item was found. As above, a big number might indicate a less-than-optimal cache. You might consider running *DBCC FREESYSTEMCACHE* to remove all unused entries in the cache stores. You can get more details on this command in *SQL Server Books Online.*

sys.dm_os_memory_cache_clock_hands This DMV, discussed earlier, can be joined to the other cache DMVs using the *cache_address* column. Interesting columns include the following:

clock_hand The type of clock hand, either external or internal. Remember that there are two clock hands for every store.

clock_status The status of the clock hand: suspended or running. A clock hand runs when a corresponding policy kicks in.

rounds_count The number of rounds the clock hand has made. All the external clock hands should have the same (or close to the same) value in this column.

removed_all_rounds_count The number of entries removed by the clock hand in all rounds.

NUMA and Memory

As mentioned earlier, one major reason for implementing NUMA is to handle large amounts of memory efficiently. As clock speed and the number of processors increase, it becomes increasingly difficult to reduce the memory latency required to use this additional processing power. Large L3 caches can help alleviate part of the problem, but this is only a limited solution. NUMA is the scalable solution of choice. SQL Server 2008 has been designed to take advantage of NUMA-based computers without requiring any application changes. Keep in mind that the NUMA memory nodes depend completely on the hardware NUMA configuration. If you define your own soft-NUMA, as discussed earlier, you will not affect the number of NUMA memory nodes. So, for example, if you have an SMP computer with eight CPUs and you create four soft-NUMA nodes with two CPUs each, you have only one MEMORY node serving all four NUMA nodes. Soft-NUMA does not provide memory to CPU affinity. However, there is a network I/O thread and a lazywriter thread for each NUMA node, either hard or soft.

The principal reason for using soft-NUMA is to reduce I/O and lazywriter bottlenecks on computers with many CPUs and no hardware NUMA. For instance, on a computer with eight CPUs and no hardware NUMA, you have one I/O thread and one lazywriter thread that could be a bottleneck. Configuring four soft-NUMA nodes provides four I/O threads and four lazywriter threads, which could definitely help performance.

If you have multiple NUMA memory nodes, SQL Server divides the total target memory evenly among all the nodes. So if you have 10 GB of physical memory and four NUMA nodes and SQL Server determines a 10-GB target memory value, all nodes eventually allocate and use 2.5 GB of memory as if it were their own. In fact, if one of the nodes has less memory than another, it must use memory from another one to reach its 2.5-GB allocation. This memory is called *foreign memory*. Foreign memory is considered local, so if SQL Server has readjusted its target memory and each node needs to release some, no attempt will be made to free up foreign pages first. In addition, if SQL Server has been configured to run on a subset of the available NUMA nodes, the target memory will *not* be limited automatically to the memory on those nodes. You must set the Max Server Memory value to limit the amount of memory.

In general, the NUMA nodes function largely independently of each other, but that is not always the case. For example, if a worker running on a node *N1* needs to access a database page that is already in node *N2*'s memory, it does so by accessing *N2*'s memory, which is called *nonlocal memory*. Note that nonlocal is not the same as foreign memory.

Read-Ahead

SQL Server supports a mechanism called *read-ahead,* whereby the need for data and index pages can be anticipated and pages can be brought into the buffer pool before they're actually needed. This performance optimization allows large amounts of data to be processed effectively. Read-ahead is managed completely internally, and no configuration adjustments are necessary.

There are two kinds of read-ahead: one for table scans on heaps and one for index ranges. For table scans, the table's allocation structures are consulted to read the table in disk order. Up to 32 extents (32 * 8 pages/extent * 8,192 bytes/page = 2 MB) of read-ahead may be outstanding at a time. Four extents (32 pages) at a time are read with a single 256-KB scatter read. If the table is spread across multiple files in a file group, SQL Server attempts to distribute the read-ahead activity across the files evenly.

For index ranges, the scan uses level 1 of the index structure (the level immediately above the leaf) to determine which pages to read ahead. When the index scan starts, read-ahead is invoked on the initial descent of the index to minimize the number of reads performed. For instance, for a scan of *WHERE state = 'WA'*, read-ahead searches the index for *key = 'WA'*, and it can tell from the level-1 nodes how many pages must be examined to satisfy the scan. If the anticipated number of pages is small, all the pages are requested by the initial read-ahead; if the pages are noncontiguous, they're fetched in scatter reads. If the range contains a large number of pages, the initial read-ahead is performed and thereafter, every time another 16 pages are consumed by the scan, the index is consulted to read in another 16 pages. This has several interesting effects:

- Small ranges can be processed in a single read at the data page level whenever the index is contiguous.

- The scan range (for example, *state = 'WA'*) can be used to prevent reading ahead of pages that won't be used because this information is available in the index.
- Read-ahead is not slowed by having to follow page linkages at the data page level. (Read-ahead can be done on both clustered indexes and nonclustered indexes.)

As you can see, memory management in SQL Server is a huge topic, and I've provided you with only a basic understanding of how SQL Server uses memory. This information should give you a start in interpreting the wealth of information available through the DMVs and troubleshooting. The companion Web site includes a white paper that offers many more troubleshooting ideas and scenarios.

SQL Server Resource Governor

Having sufficient memory and scheduler resources available is of paramount importance in having a system that runs well. Although SQL Server and the SQLOS have many built-in algorithms to distribute these resources equitably, you often understand your resource needs better than the SQL Server Database Engine does.

Resource Governor Overview

SQL Server 2008 Enterprise Edition provides you with an interface for assigning scheduler and memory resources to groups of processes based on your determination of their needs. This interface is called the *Resource Governor,* which has the following goals:

- Allow monitoring of resource consumption per workload, where a workload can be defined as a group of requests.
- Enable workloads to be prioritized.
- Provide a means to specify resource boundaries between workloads to allow predictable execution of those workloads where there might otherwise be resource contention
- Prevent or reduce the probability of runaway queries.

The Resource Governor's functionality is based on the concepts of workloads and resource pools, which are set up by the DBA. Using just a few basic DDL commands, you can define a set of workload groups, create a classifier function to determine which user sessions are members of which groups, and set up pools of resources to allow each workload group to have minimum and maximum settings for the amount of memory and the percentage of CPU resources that they can use.

Figure 1-4 illustrates a sample relationship between the classifier function applied to each session, workload groups, and resource pools. More details about groups and pools are provided throughout this section, but you can see in the figure that each new session is placed in a workload group based on the result of the classifier function. Also notice that there is a many-to-one relationship between groups and pools. Many workload groups can be assigned to the same pool, but each workload group only belongs on one pool.

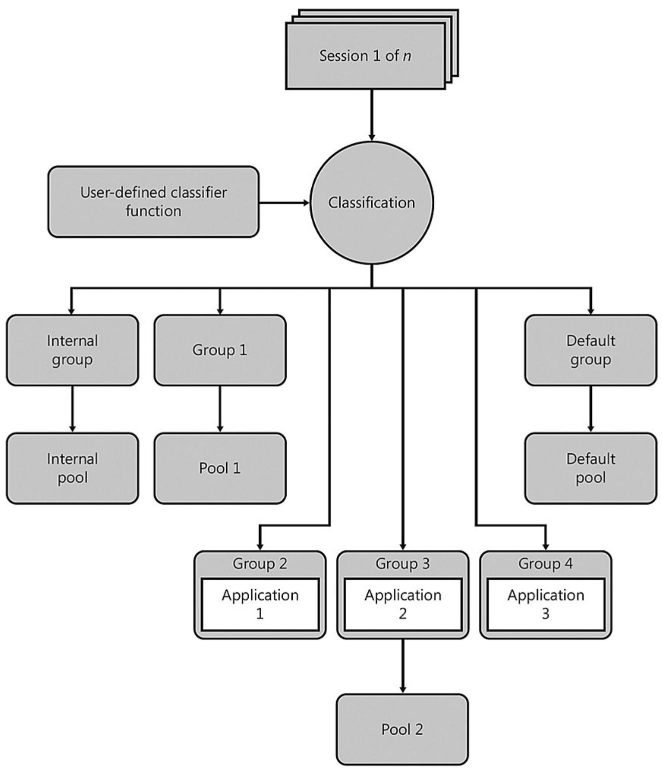


Figure 1-4: Resource Governor components

Enabling the Resource Governor

The Resource Governor is enabled using the DDL statement *ALTER RESOURCE GOVERNOR.* Using this statement, you can specify a classifier function to be used to assign sessions to a workload, enable or disable the Resource Governor, or reset the statistics being kept on the Resource Governor.

Classifier Function

Once a classifier function has been defined and the Resource Governor enabled, the function is applied to each new session to determine the name of the workload group to which the session will be assigned. The session stays in the same group until its termination, unless it is assigned explicitly to a different group. There can only be a maximum of one classifier function active at any given time, and if no classifier function has been defined, all new sessions are assigned to a default group. The classifier function is typically based on properties of a connection, and determines the workload group based on system functions such as *SUSER_NAME(), SUSER_SNAME(), IS_SRVROLEMEMBER(),* and *IS_MEMBER(),* and on property functions like *LOGINPROPERTY* and *CONNECTIONPROPERTY.*

Workload Groups

A *workload group* is just a name defined by a DBA to allow multiple connections to share the same resources. There are two predefined workload groups in every SQL Server instance:

- **Internal group** This group is used for the internal activities of SQL Server. Users are not able to add sessions to the internal group or affect its resource usage. However, the internal group can be monitored.
- **Default group** All sessions are classified into this group when no other classifier rules could be applied. This includes situations where the classifier function resulted in a nonexistent group or when there was a failure of the classifier function.

Many sessions can be assigned to the same workload group, and each session can start multiple sequential tasks (or batches). Each batch can be composed of multiple statements, and some of those statements, such as stored procedure calls, can be broken down further. Figure 1-5 illustrates this relationship between workload groups, sessions, batches, and statements.

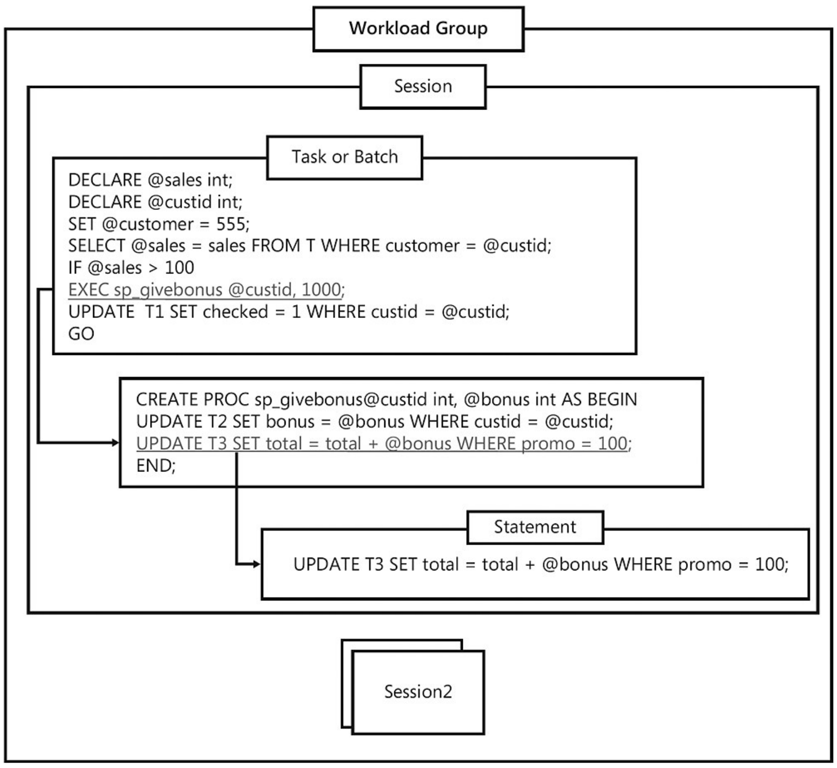

Workload Group
Session
Task or Batch
DECLARE @sales int;
DECLARE @custid int;
SET @customer = 555;
SELECT @sales = sales FROM T WHERE customer = @custid;
IF @sales > 100
EXEC sp_givebonus @custid, 1000;
UPDATE T1 SET checked = 1 WHERE custid = @custid;
GO
CREATE PROC sp_givebonus@custid int, @bonus int AS BEGIN
UPDATE T2 SET bonus = @bonus WHERE custid = @custid;
UPDATE T3 SET total = total + @bonus WHERE promo = 100;
END;
Statement
UPDATE T3 SET total = total + @bonus WHERE promo = 100;
Session2

Figure 1-5: Workload groups, sessions, batches, and statements

When you create a workload group, you give it a name and then supply values for up to six specific properties of the group. For any properties that aren't specified, there is a default value. In addition to the properties of the group, the group is assigned to a resource pool; and if no pool is specified, the default group is assumed. The six properties that can be specified are the following:

1. **IMPORTANCE** Each workload group can have an importance of *low, medium*, or *high* within their resource pool. Medium is the default. This value determines the relative ratio of CPU bandwidth available to the group in a preset proportion (which is subject to change in future versions or services packs). Currently the weighting is low = 1, medium =3, and high = 9. This means that a scheduler tries to execute runnable sessions from high-priority workload groups three times more often than sessions from groups with medium importance, and nine times more often than sessions from groups with low importance. It's up to the DBA to make sure not to have too many sessions in the groups with high importance, or not to assign a high importance to too many groups. If you have nine times as many sessions from groups with high importance than from groups with low importance, the end result will be that all the sessions will get equal time on the scheduler.

2. **REQUEST_MAX_MEMORY_GRANT_PERCENT** This value specifies the maximum amount of memory that a single task from this group can take from the resource pool. This is the percent relative to the pool size specified by the pool's MAX_MEMORY_PERCENT value, not the actual amount of memory being used. This amount refers only to memory granted for query execution, and not for data buffers or cached plans, which can be shared by many requests. The default value is 25 percent, which means a single request can consume one-fourth of the pool's memory.

3. **REQUEST_MAX_CPU_TIME_SEC** This value is the maximum amount of CPU time in seconds that can be consumed by any one request in the workload group. The default setting is 0, which means there is no limit on the CPU time.

4. **REQUEST_MEMORY_GRANT_TIMEOUT_SEC** This value is the maximum time in seconds that a query waits for a resource to become available. If the resource does not become available, it may fail with a timeout error. (In some cases, the query may not fail, but it may run with substantially reduced resources.) The default value is 0, which means the server will calculate the timeout based on the query cost.

5. **MAX_DOP** This value specifies the maximum degree of parallelism (DOP) for a parallel query, and the value takes precedence over the max degree of parallelism configuration option and any query hints. The actual run-time DOP is also bound by number of schedulers and availability of parallel threads. This MAX_DOP setting is a maximum limit

only, meaning that the server is allowed to run the query using fewer processors than specified. The default setting is 0, meaning that the server handles the value globally. You should be aware of the following details about working with the MAX_DOP value:

- MAXDOP as query hint is honored so long as it does not exceed the workload group MAX_DOP value.
- MAXDOP as query hint always overrides the Max Degree of Parallelism configuration option.
- If the query is marked as serial at compile time, it cannot be changed back to parallel at run time regardless of workload group or configuration setting.
- Once the degree of parallelism is decided, it can be lowered only when memory pressure occurs. Workload group reconfiguration will not be seen for tasks waiting in the grant memory queue.

6. **GROUP_MAX_REQUESTS** This value is the maximum number of requests allowed to be simultaneously executing in the workload group. The default is 0, which means unlimited requests.

Any of the properties of a workload group can be changed by using *ALTER WORKLOAD GROUP.*

Resource Pools

A resource pool is a subset of the physical resources of the server. Each pool has two parts. One part does not overlap with other pools, which enables you to set a minimum value for the resource. The other part of the pool is shared with other pools, and this allows you to define the maximum possible resource consumption. The pool resources are set by specifying one of the following for each resource:

- MIN or MAX for CPU
- MIN or MAX for memory percentage

MIN represents the minimum guaranteed resource availability for CPU or memory and MAX represents the maximum size of the pool for CPU or memory.

The shared part of the pool is used to indicate where available resources can go if resources are available. However, when resources are consumed, they go to the specified pool and are not shared. This may improve resource utilization in cases where there are no requests in a given pool and the resources configured to the pool can be freed up for other pools.

Here are more details about the four values that can be specified for each resource pool:

1. **MIN_CPU_PERCENT** This is a guaranteed average CPU bandwidth for all requests in the pool when there is CPU contention. SQL Server attempts to distribute CPU bandwidth between individual requests as fairly as possible and takes the *IMPORTANCE* property for each workload group into account. The default value is 0, which means there is no minimum value.
2. **MAX_CPU_PERCENT** This is the maximum CPU bandwidth that all requests in the pool receive when there is CPU contention. The default value is 100, which means there is no maximum value. If there is no contention for CPU resources, a pool can consume up to 100 percent of CPU bandwidth.
3. **MIN_MEMORY_PERCENT** This value specifies the amount of memory reserved for this pool that cannot be shared with other pools. If there are no requests in the pool but the pool has a minimum memory value set, this memory cannot be used for requests in other pools and is wasted. Within a pool, distribution of memory between requests is on a first-come-first-served basis. Memory for a request can also be affected by properties of the workload group, such as REQUEST_MAX_MEMORY_GRANT_PERCENT. The default value of 0 means that there is no minimum memory reserved.
4. **MAX_MEMORY_PERCENT** This value specifies the percent of total server memory that can be used by all requests in the specified pool. This amount can go up to 100 percent, but the actual amount is reduced by memory already reserved by the MIN_MEMORY_PERCENT value specified by other pools. MAX_MEMORY_PERCENT is always greater than or equal to MIN_MEMORY_PERCENT. The amount of memory for an individual request will be affected by workload group policy, for example, REQUEST_MAX_MEMORY_GRANT_PERCENT. The default setting of 100 means that all the server memory can be used for one pool. This setting cannot be exceeded, even if it means that the server will be underutilized.

Some extreme cases of pool configuration are the following:

- All pools define minimums that add up to 100 percent of the server resources. This is equivalent to dividing the server resources into nonoverlapping pieces regardless of the resources consumed inside any given pool.
- All pools have no minimums. All the pools compete for available resources, and their final sizes are based on resource consumption in each pool.

Resource Governor has two predefined resource pools for each SQL Server instance:

Internal pool This pool represents the resources consumed by the SQL Server itself. This pool always contains only the internal workload group and is not alterable in any way. There are no restrictions on the resources used by the internal pool. You are not able to affect the resource usage of the internal pool or add workload groups to it. However, you are able to monitor the resources used by the internal group.

Default pool Initially, the default pool contains only the default workload group. This pool cannot be dropped, but it can be altered and other workload groups can be added to it. Note that the default group cannot be moved out of the default pool.

Pool Sizing

Table 1-2, taken from *SQL Server 2008 Books Online*, illustrates the relationships between the MIN and MAX values in several pools and how the effective MAX values are computed. The table shows the settings for the internal pool, the default pool, and two user-defined pools. The following formulas are used for calculating the effective MAX % and the shared %:

- Min(X,Y) means the smaller value of *X* and *Y.*
- Sum(X) means the sum of value *X* across all pools.
- Total shared % = 100 – sum(MIN %).
- Effective MAX % = min(X,Y).
- Shared % = Effective MAX % – MIN %.

Table 1-2: MIN and MAX Values for Workload Groups

Pool Name	MIN % Setting	MAX % Setting	Calculated Effective MAX %	Calculated Shared %	Comment
internal	0	100	100	0	Effective MAX % and shared % are not applicable to the internal pool.
default	0	100	30	30	The effective MAX value is calculated as min (100,100–(20+50)) = 30. The calculated shared % is effective MAX – MIN = 30.
Pool 1	20	100	50	30	The effective MAX value is calculated as min (100,100–50) = 50. The calculated shared % is effective MAX – MIN = 30.
Pool 2	50	70	70	20	The effective MAX value is calculated as min (70,100–20) = 70. The calculated shared % is effective MAX – MIN = 20.

Table 1-3, also taken from *SQL Server Books Online*, shows how the values above can change when a new pool is created. This new pool is Pool 3 and has a MIN % setting of 5.

Table 1-3: MIN and MAX Values for Resource Pools

Pool Name	MIN % Setting	MAX % Setting	Calculated Effective MAX %	Calculated Shared %	Comment
internal	0	100	100	0	Effective MAX % and shared % are not applicable to the internal pool.
default	0	100	25	30	The effective MAX value is calculated as min (100,100–(20+50+5)) = 25. The calculated shared %

					is effective MAX – MIN = 25.
Pool 1	20	100	45	25	The effective MAX value is calculated as min (100,100–55) = 45. The calculated shared % is effective MAX – MIN = 30.
Pool 2	50	70	70	20	The effective MAX value is calculated as min (70,100–25) = 70. The calculated shared % is effective MAX – MIN = 20.
Pool 3	5	100	30	25	The effective MAX value is calculated as min (100,100–70) = 30. The calculated shared % is effective MAX – MIN = 25.

Example

This section includes a few syntax examples of the Resource Governor DDL commands, to give a further idea of how all these concepts work together. This is not a complete discussion of all the possible DDL command options; for that, you need to refer to *SQL Server Books Online.*

```
--- Create a resource pool for production processing
--- and set limits.
USE master;
GO
CREATE RESOURCE POOL pProductionProcessing
WITH
(
     MAX_CPU_PERCENT = 100,
     MIN_CPU_PERCENT = 50
);
GO
--- Create a workload group for production processing
--- and configure the relative importance.
CREATE WORKLOAD GROUP gProductionProcessing
WITH
(
     IMPORTANCE = MEDIUM
)
--- Assign the workload group to the production processing
--- resource pool.
USING pProductionProcessing;
GO
--- Create a resource pool for off-hours processing
--- and set limits.
CREATE RESOURCE POOL pOffHoursProcessing
WITH
(
     MAX_CPU_PERCENT = 50,
     MIN_CPU_PERCENT = 0
);
GO
--- Create a workload group for off-hours processing
--- and configure the relative importance.
CREATE WORKLOAD GROUP gOffHoursProcessing
WITH
(
     IMPORTANCE = LOW
)
--- Assign the workload group to the off-hours processing
--- resource pool.
USING pOffHoursProcessing;
GO
--- Any changes to workload groups or resource pools require that the
--- resource governor be reconfigured.
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
USE master;
GO
CREATE TABLE tblClassifierTimeTable (
     strGroupName     sysname          not null,
     tStartTime       time              not null,
     tEndTime         time              not null
);
GO
--- Add time values that the classifier will use to
--- determine the workload group for a session.
INSERT into tblClassifierTimeTable
     VALUES('gProductionProcessing', '6:35 AM', '6:15 PM');
GO
--- Create the classifier function
CREATE FUNCTION fnTimeClassifier()
RETURNS sysname
WITH SCHEMABINDING
AS
BEGIN
     DECLARE @strGroup sysname
     DECLARE @loginTime time
     SET @loginTime = CONVERT(time,GETDATE())
     SELECT TOP 1 @strGroup = strGroupName
     FROM dbo.tblClassifierTimeTable
     WHERE tStartTime <= @loginTime and tEndTime >= @loginTime
     IF(@strGroup is not null)
     BEGIN
          RETURN @strGroup
     END
--- Use the default workload group if there is no match
--- on the lookup.
     RETURN N'gOffHoursProcessing'
END;
GO
--- Reconfigure the Resource Governor to use the new function
ALTER RESOURCE GOVERNOR with (CLASSIFIER_FUNCTION = dbo.fnTimeClassifier);
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

Resource Governor Controls

The actual limitations of resources are controlled by your pool settings. In SQL Server 2008, you can control memory and CPU resources, but not I/O. It's possible that in a future version, more resource controls will become available. There is an important difference between the way that memory and CPU resources limits are applied.

You can think of the memory specifications for a pool as hard limits, and no pool will ever use more than its maximum memory setting. In addition, SQL Server always reserves the minimum memory for each pool, so that if no sessions in workload groups are assigned to a pool, its minimum memory reservation is unusable by other sessions.

However, CPU limits are soft limits, and unused scheduler bandwidth can be used by other sessions. The maximum values are also not always fixed upper limits. For example, if there are two pools, one with a maximum of 25 percent and the other with a maximum of 50 percent, as soon as the first pool has used its 25 percent of the scheduler, sessions from groups in the other pool can use all the remaining CPU resources. As soft limits, they can make CPU usage not quite as predictable as memory usage. Each session is assigned to a scheduler, as described in the previous section, with no regard to the workload group that the session is in. Assume a minimal situation with only two sessions running on a dual CPU instance. Each will most likely be assigned to a different scheduler, and the two sessions may be in two different workload groups in two different resource pools.

Assume that the session on CPU1 is from a workload group in the first pool that has a maximum CPU setting of 80 percent, and that the second session, on CPU2, is from a group in the second pool with a maximum CPU setting of 20 percent. Because these are only two sessions, they each use 100 percent of their scheduler or 50 percent of the total CPU resources on the instance. If CPU1 is then assigned another task from a workload group from the 20 percent pool, the situation changes. Tasks using the 20 percent pool have 20 percent of CPU1 but still have 100 percent of CPU2, and tasks using the 80 percent pool still have only 80 percent of CPU1. This means tasks running from the 20 percent pool have 60 percent of the total CPU resources, and the one task from the 80 percent pool has only 40 percent of the total CPU resources. Of course, as more and more tasks are assigned to the schedulers, this anomaly may work itself out, but because of the way that scheduler resources are managed across multiple CPUs, there is much less explicit control.

For testing and troubleshooting purposes, there may be times you want to be able to turn off all Resource Governor functionality easily. You can disable the Resource Governor with the command *ALTER RESOURCE GOVERNOR*

DISABLE. You can then re-enable the Resource Governor with the command *ALTER RESOURCE GOVERNOR RECONFIGURE.* If you want to make sure the Resource Governor stays disabled, you can start your SQL Server instance with trace flag 8040 in this situation. When this trace flag is used, Resource Governor stays in the OFF state at all times and all attempts to reconfigure it fails. The same behavior results if you start your SQL Server instance in single-user mode using the *–m* and *–f* flags. If the Resource Governor is disabled, you should notice the following behaviors:

- Only the *internal* workload group and resource pool exist.
- Resource Governor configuration metadata are not loaded into memory.
- Your classifier function is never executed automatically.
- The Resource Governor metadata is visible and can be manipulated.

Resource Governor Metadata

There are three specific catalog views that you'll want to take a look at when working with the Resource Governor.

sys.resource_governor_configuration This view returns the stored Resource Governor state.

sys.resource_governor_resource_pools This view returns the stored resource pool configuration. Each row of the view determines the configuration of an individual pool.

sys.resource_governor_workload_groups This view returns the stored workload group configuration.

There are also three DMVs devoted to the Resource Governor:

- **sys.dm_resource_governor_workload_groups** This view returns workload group statistics and the current in-memory configuration of the workload group.
- **sys.dm_resource_governor_resource_pools** This view returns information about the current resource pool state, the current configuration of resource pools, and resource pool statistics.
- **sys.dm_resource_governor_configuration** This view returns a row that contains the current in-memory configuration state for the Resource Governor.

Finally, six other DMVs contain information related to the Resource Governor:

- **sys.dm_exec_query_memory_grants** This view returns information about the queries that have acquired a memory grant or that still require a memory grant to execute. Queries that do not have to wait for a memory grant do not appear in this view. The following columns are added for the Resource Governor: *group_id, pool_id, is_small, ideal_memory_kb.*
- **sys.dm_exec_query_resource_semaphores** This view returns the information about the current query-resource semaphore status. It provides general query-execution memory status information and allows you to determine whether the system can access enough memory. The *pool_id* column has been added for the Resource Governor.
- **sys.dm_exec_sessions** This view returns one row per authenticated session on SQL Server. The *group_id* column has been added for the Resource Governor.
- **sys.dm_exec_requests** This view returns information about each request that is executing within SQL Server. The *group_id* column is added for the Resource Governor.
- **sys.dm_exec_cached_plans** This view returns a row for each query plan that is cached by SQL Server for faster query execution. The *pool_id* column is added for the Resource Governor.
- **sys.dm_os_memory_brokers** This view returns information about allocations that are internal to SQL Server, which use the SQL Server memory manager. The following columns are added for the Resource Governor: *pool_id, allocations_db_per_sec, predicated_allocations_kb, overall_limit_kb.*

Although at first glance it may seem like the setup of the Resource Governor is unnecessarily complex, hopefully you'll find that being able to specify properties for both workload groups and resource pools provides you with the maximum control and flexibility. You can think of the workload groups as tools that give control to your developers, and the resource pools as administrator tools for limiting what the developers can do.

SQL Server 2008 Configuration

In the second part of this chapter, we'll look at the options for controlling how SQL Server 2008 behaves. One main method of controlling the behavior of the Database Engine is to adjust configuration option settings, but you can configure behavior in a few other ways as well. We'll first look at using SQL Server Configuration Manager to control network protocols and SQL Server–related services. We'll then look at other machine settings that can affect the behavior of SQL Server. Finally, we'll examine some specific configuration options for controlling server-wide settings in SQL Server.

Using SQL Server Configuration Manager

Configuration Manager is a tool for managing the services associated with SQL Server, configuring the network protocols used by SQL Server, and managing the network connectivity configuration from client computers connecting to SQL Server. It is installed as part of SQL Server. Configuration Manager is available by right-clicking the registered server in Management Studio, or you can add it to any other Microsoft Management Console (MMC) display.

Configuring Network Protocols

A specific protocol must be enabled on both the client and the server for the client to connect and communicate with the server. SQL Server can listen for requests on all enabled protocols at once. The underlying operating system network protocols (such as TCP/IP) should already be installed on the client and the server. Network protocols are typically installed during Windows setup; they are not part of SQL Server setup. A SQL Server network library does not work unless its corresponding network protocol is installed on both the client and the server.

On the client computer, the SQL Native Client must be installed and configured to use a network protocol enabled on the server; this is usually done during Client Tools Connectivity setup. The SQL Native Client is a standalone data access API used for both OLE DB and ODBC. If the SQL Native Client is available, any network protocol can be configured for use with a particular client connecting to SQL Server. You can use SQL Server Configuration Manager to enable a single protocol or to enable multiple protocols and specify an order in which they should be attempted. If the Shared Memory protocol setting is enabled, that protocol is always tried first, but, as mentioned earlier in this chapter, it is available for communication only when the client and the server are on the same machine.

The following query returns the protocol used for the current connection, using the DMV *sys.dm_exec_connections*:

```
SELECT net_transport
FROM sys.dm_exec_connections
WHERE session_id = @@SPID;
```

Default Network Configuration

The network protocols that can be used to communicate with SQL Server 2008 from another computer are not all enabled for SQL Server during installation. To connect from a particular client computer, you might need to enable the desired protocol. The Shared Memory protocol is enabled by default on all installations, but because it can be used to connect to the Database Engine only from a client application on the same computer, its usefulness is limited.

TCP/IP connectivity to SQL Server 2008 is disabled for new installations of the Developer, Evaluation, and SQL Express editions. OLE DB applications connecting with MDAC 2.8 cannot connect to the default instance on a local server using ".", "(local)", or (<blank>) as the server name. To resolve this, supply the server name or enable TCP/IP on the server. Connections to local named instances are not affected, nor are connections using the SQL Native Client. Installations in which a previous installation of SQL Server is present might not be affected.

Table 1-4 describes the default network configuration settings.

Table 1-4: SQL Server 2008 Default Network Configuration Settings

SQL Server Edition	Type of Installation	Shared Memory	TCP/IP	Named Pipes	VIA
Enterprise	New	Enabled	Enabled	Disabled (available only locally)	Disabled
Enterprise (clustered)	New	Enabled	Enabled	Enabled	Disabled

Developer	New	Enabled	Disabled	Disabled (available only locally)	Disabled
Standard	New	Enabled	Enabled	Disabled (available only locally)	Disabled
Workgroup	New	Enabled	Enabled	Disabled (available only locally)	Disabled
Evaluation	New	Enabled	Disabled	Disabled (available only locally)	Disabled
Web	New	Enabled	Enabled	Disabled (available only locally)	Disabled
SQL Server Express	New	Enabled	Disabled	Disabled (available only locally)	Disabled
All editions	Upgrade or side-by-side installation	Enabled	Settings preserved from the previous installation	Settings preserved from the previous installation	Disabled

Managing Services

You can use configuration Manager to start, pause, resume, or stop SQL Server–related services. The services available depend on the specific components of SQL Server you have installed, but you should always have the SQL Server service itself and the SQL Server Agent service. Other services might include the SQL Server Full-Text Search service and SQL Server Integration Services (SSIS). You can also use Configuration Manager to view the current properties of the services, such as whether the service is set to start automatically. Configuration Manager is the preferred tool for changing service properties rather than using Windows service management tools. When you use a SQL Server tool such as Configuration Manager to change the account used by either the SQL Server or SQL Server Agent service, the SQL Server tool automatically makes additional configurations, such as setting permissions in the Windows Registry so that the new account can read the SQL Server settings. Password changes using Configuration Manager take effect immediately without requiring you to restart the service.

SQL Server Browser

One other related service that deserves special attention is the SQL Server Browser service. This service is particularly important if you have named instances of SQL Server running on a machine. SQL Server Browser listens for requests to access SQL Server resources and provides information about the various SQL Server instances installed on the computer where the Browser service is running.

Prior to SQL Server 2000, only one installation of SQL Server could be on a machine at one time, and there really was no concept of an “instance.” SQL Server always listened for incoming requests on port 1433, but any port can be used by only one connection at a time. When SQL Server 2000 introduced support for multiple instances of SQL Server, a new protocol called *SQL Server Resolution Protocol (SSRP)* was developed to listen on UDP port 1434. This listener could reply to clients with the names of installed SQL Server instances, along with the port numbers or named pipes used by the instance. SQL Server 2005 replaced SSRP with the SQL Server Browser service, which is still used in SQL Server 2008.

If the SQL Server Browser service is not running on a computer, you cannot connect to SQL Server on that machine unless you provide the correct port number. However, if the SQL Server Browser service is not running, the following connections will not work:

- Connecting to a named instance without providing the port number or pipe
- Using the DAC to connect to a named instance or the default instance if it us not using TCP/IP port 1433
- Enumerating servers in Management Studio, Enterprise Manager, or Query Analyzer

It is recommended that the Browser Service be set to start automatically on any machine on which SQL Server will be accessed using a network connection.

SQL Server System Configuration

You can configure the machine that SQL Server runs on, as well as the Database Engine itself, in several ways and through a variety of interfaces. We’ll first look at some operating system–level settings that can affect the behavior of SQL Server. Next, we’ll see some SQL Server options that can affect behavior that aren’t especially considered to be configuration options. Finally, we'll examine the configuration options for controlling the behavior of SQL Server 2008, which are set primarily using a stored procedure interface called *sp_configure*.

Operating System Configuration

For your SQL Server to run well, it must be running on a tuned operating system, on a machine that has been properly configured to run SQL Server. Although it is beyond the scope of this book to discuss operating system and hardware configuration and tuning, there are a few issues that are very straightforward but can have a major impact on the performance of SQL Server, and we will describe them here.

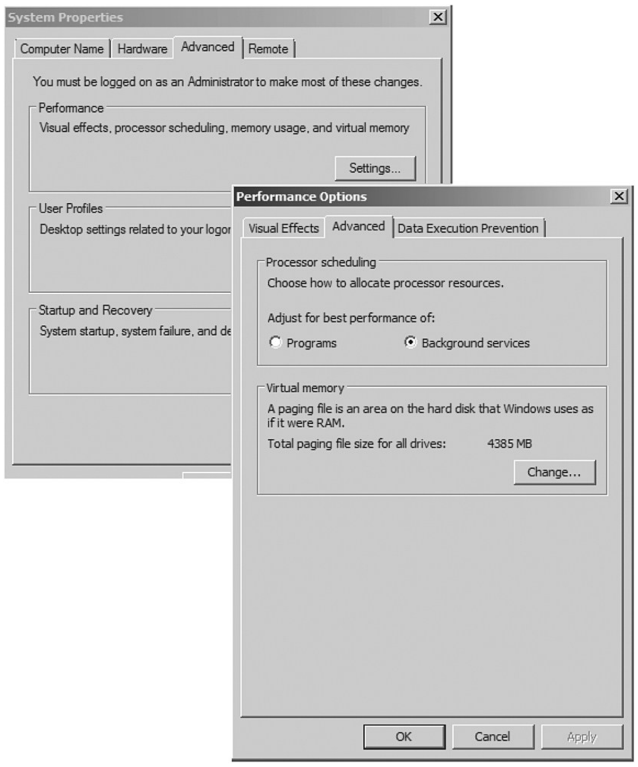
Task Management

As you saw in the first part of this chapter, the operating system schedules all threads in the system for execution. Each thread of every process has a priority, and Windows executes the next available thread with the highest priority. By default, the operating system gives active applications a higher priority, but this priority setting may not be appropriate for a server application running in the background, such as SQL Server 2008. To remedy this situation, the SQL Server installation program modifies the priority setting to eliminate the favoring of foreground applications.

It's not a bad idea to double-check this priority setting periodically in case someone has set it back. You'll need to open the Advanced tab of the Performance Options dialog box.

If you're using Windows XP or Windows Server 2003, click the Start menu, right-click My Computer, and choose Properties. The System Properties dialog box opens. On the Advanced tab, click the Settings button in the Performance area. Again, select the Advanced tab.

If you're using Windows Server 2008, click the Start menu, right-click Computer, and choose Properties. The System information screen opens. Select Advanced System Settings from the list on the left to open the System Properties dialog box. Just as for Windows XP and Windows Server 2003, on the Advanced tab, click the Settings button in the Performance area. Again, select the Advanced tab. You should see the Performance Options dialog box, shown in Figure 1-6.



You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logo

Startup and Recovery
System startup, system failure, and de

Processor scheduling
Choose how to allocate processor resources.
Adjust for best performance of:
Programs [✓ Background services]

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: 4385 MB

Figure 1-6: Configuration of priority for background services

The first set of options is for specifying how to allocate processor resources, and you can adjust for the best performance of either programs or background services. Select Background Services so that all programs (both background and foreground) receive equal processor resources. If you plan to connect to SQL Server 2008 from a local client (that is, a client running on the same computer as the server), you can improve processing time by using this setting.

System Paging File Location

If possible, you should place the operating system paging file on a different drive than the files used by SQL Server. This is vital if your system will be paging. However, a better approach is to add memory or change the SQL Server memory configuration to effectively eliminate paging. In general, SQL Server is designed to minimize paging, so if your memory configuration values are appropriate for the amount of physical memory on the system, such a small amount of page-file activity will occur that the file's location is irrelevant.

Nonessential Services

You should disable any services that you don't need. In Windows Server 2003, you can right-click My Computer and choose Manage. Expand the Services And Applications node in the Computer Management tool, and click Services. In the right-hand pane, you see a list of all the services available on the operating system. You can change a service's startup property by right-clicking its name and choosing Properties. Unnecessary services add overhead to the system and use resources that could otherwise go to SQL Server. No unnecessary services should be marked for automatic startup. Avoid using a server that's running SQL Server as a domain controller, the group's file or print server, the Web server, or the Dynamic Host Configuration Protocol (DHCP) server. You should also consider disabling the Alerter, ClipBook, Computer Browser, Messenger, Network Dynamic Data Exchange (DDE), and Task Scheduler services, which are enabled by default but are not needed by SQL Server.

Connectivity

You should run only the network protocols that you actually need for connectivity. You can use the SQL Server Configuration Manager to disable unneeded protocols, as described earlier in this chapter.

Firewall Setting

Improper firewall settings are another system configuration issue that can inhibit SQL Server connectivity across your network. Firewall systems help prevent unauthorized access to computer resources and are usually desirable, but to access an instance of SQL Server through a firewall, you'll need to configure the firewall on the computer running SQL Server to allow access. Many firewall systems are available, and you'll need to check the documentation for your system for the exact details of how to configure it. In general, you'll need to carry out the following steps:

1. Configure the SQL Server instance to use a specific TCP/IP port. Your default SQL Server uses port 1433 by default, but that can be changed. Named instances use dynamic ports by default, but that can also be changed using the SQL Server Configuration Manager.

2. Configure your firewall to allow access to the specific port for authorized users or computers.

3. As an alternative to configuring SQL Server to listen on a specific port and then opening that port, you can list the SQL Server executable (Sqlservr.exe) and the SQL Browser executable (Sqlbrowser.exe) when requiring a connection to named instances, as exceptions to the blocked programs. You can use this method when you want to continue to use dynamic ports.

Trace Flags

SQL Server Books Online lists only about a dozen trace flags that are fully supported. You can think of trace flags as special switches that you can turn on or off to change the behavior of SQL Server. There are actually many dozens, if not hundreds, of trace flags. However, most were created for the SQL Server development team's internal testing of the product and were never intended for use by anybody outside Microsoft.

You can set trace flags on or off by using the *DBCC TRACEON* or *DBCC TRACEOFF* command or by specifying them on the command line when you start SQL Server using Sqlservr.exe. You can also use the SQL Server Configuration Manager to enable one or more trace flags every time the SQL Server service is started. (You can read about how to do that in *SQL Server Books Online.*) Trace flags enabled with *DBCC TRACEON* are valid only for a single connection unless you specified an additional parameter of –1, in which case they are active for all connections, even ones opened before you ran *DBCC TRACEON*. Trace flags enabled as part of starting the SQL Server service are enabled for all sessions.

A few of the trace flags are particularly relevant to topics covered in this book, and we will discuss particular ones when we describe topics that they are related to. For example, we already mentioned trace flag 8040 in conjunction with the Resource Governor.

Caution Because trace flags change the way SQL Server behaves, they can actually cause trouble if used inappropriately. Trace flags are not harmless features that you can experiment with just to see what happens, especially not on a production system. Using them effectively requires a thorough understanding of SQL Server default behavior (so that you know exactly what you'll be changing) and extensive testing to determine that your system really will benefit from the use of the trace flag.

SQL Server Configuration Settings

If you choose to have SQL Server automatically configure your system, it dynamically adjusts the most important configuration options for you. It's best to accept the default configuration values unless you have a good reason to change them. A poorly configured system can destroy performance. For example, a system with an incorrectly configured memory setting can break an application.

In certain cases, tweaking the settings rather than letting SQL Server dynamically adjust them might lead to a tiny performance improvement, but your time is probably better spent on application and database designing, indexing, query tuning, and other such activities, which we'll talk about later in this book. You might see only a 5 percent improvement in performance by moving from a reasonable configuration to an ideal configuration, but a badly configured system can kill your application's performance.

SQL Server 2008 has 68 server configuration options that you can query using the catalog view *sys.configurations*.

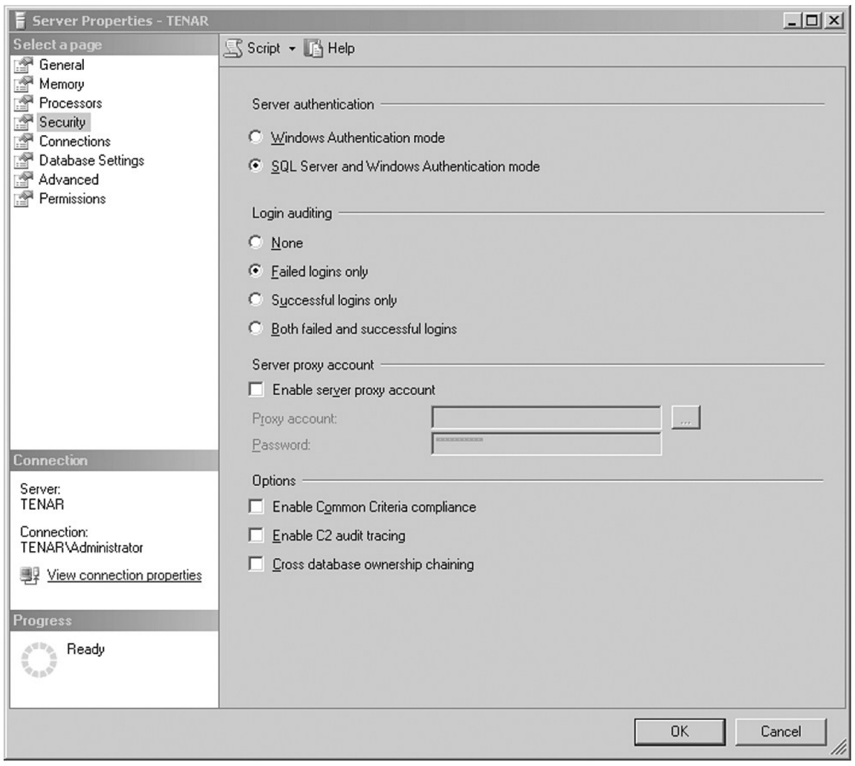
You should change configuration options only when you have a clear reason for doing so, and you should closely monitor the effects of each change to determine whether the change improved or degraded performance. Always make and monitor changes one at a time. The server-wide options discussed here can be changed in several ways. All of them can be set via the *sp_configure* system stored procedure. However, of the 68 options, all but 16 are considered advanced options and are not manageable by default using *sp_configure*. You'll first need to change the Show Advanced Options option to be 1, as shown here:

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
```

To see which options are advanced, you can again query the *sys.configurations* view and examine a column called *is_advanced,* which lets you see which options are considered advanced:

```
SELECT * FROM sys.configurations
WHERE is_advanced = 1;
GO
```

Many of the configuration options can also be set from the Server Properties dialog box in the Object Explorer window of Management Studio, but there is no single dialog box from which all configuration settings can be seen or changed. Most of the options that you can change from the Server Properties dialog box are controlled from one of the property pages that you reach by right-clicking the name of your SQL Server instance from Management Studio. You can see the list of property pages in Figure 1-7.



Server Properties - TENAR

Select a page: General, Memory, Processors, Security, Connections, Database Settings, Advanced, Permissions

Server authentication
- Windows Authentication mode
- [✓ SQL Server and Windows Authentication mode]

Login auditing
- None
- [✓ Failed logins only]
- Successful logins only
- Both failed and successful logins

Server proxy account
- [ ] Enable server proxy account
- Proxy account:
- Password: **********

Options
- [ ] Enable Common Criteria compliance
- [ ] Enable C2 audit tracing
- [ ] Cross database ownership chaining

Connection
Server: TENAR
Connection: TENAR\Administrator

Progress: Ready

Figure 1-7: List of server property pages in Management Studio

If you use the *sp_configure* stored procedure, no changes take effect until the *RECONFIGURE* command runs. In some cases, you might have to specify *RECONFIGURE WITH OVERRIDE* if you are changing an option to a value outside the recommended range. Dynamic changes take effect immediately upon reconfiguration, but others do not take effect until the server is restarted. If after running *RECONFIGURE,* an option's *run_value* and *config_value* as displayed by *sp_configure* are different, or if the value and value_in_use in *sys.configurations* are different, you must restart the SQL Server service for the new value to take effect. You can use the *sys.configurations* view to determine which options are dynamic:

```
SELECT * FROM sys.configurations
WHERE is_dynamic = 1;
GO
```

We won't look at every configuration option here—only the most interesting ones or ones that are related to SQL Server performance. In most cases, I'll discuss options that you should *not* change. Some of these are resource settings that relate to performance only in that they consume memory (for example, Locks). But if they are configured too high, they can rob a system of memory and degrade performance. We'll group the configuration settings by functionality. Keep in mind that SQL Server sets almost all these options automatically, and your applications work well without you ever looking at them.

Memory Options

In the preceding section, you saw how SQL Server uses memory, including how it allocates memory for different uses and when it reads data from or writes data to disk. However, we did not discuss how to control how much memory SQL Server actually uses for these purposes.

Min Server Memory and Max Server Memory By default, SQL Server adjusts the total amount of the memory resources it will use. However, you can use the Min Server Memory and Max Server Memory configuration options to take manual control. The default setting for Min Server Memory is 0 MB, and the default setting for Max Server Memory is 2147483647. If you use the *sp_configure* stored procedure to change both of these options to the same value, you basically take full control and tell SQL Server to use a fixed memory size. The absolute maximum of 2147483647 MB is actually the largest value that can be stored in the integer field of the underlying system table. It is not related to the actual resources of your system. The Min Server Memory option does not force SQL Server to acquire a minimum amount of memory at startup. Memory is allocated on demand based on the database workload. However, once the Min Server Memory threshold is reached, SQL Server does not release memory if it would be left with less than that amount. To ensure that each instance has allocated memory at least equal to the Min Server Memory value, therefore, we recommend that you execute a database server load shortly after startup. During normal server activity, the memory available per instance varies, but there is never less than the Min Server Memory value available for each instance.

Set Working Set Size The configuration option Set Working Set Size is a setting from earlier versions, and it has been deprecated. This setting is ignored in SQL Server 2008, even though you do not receive an error message when you try to use this value.

AWE Enabled This option enables the use of the AWE API to support large memory sizes on 32-bit systems. With AWE enabled, SQL Server 2008 can use as much memory as the Enterprise, Developer, or Standard editions allow. When running on Windows Server 2003 or Windows Server 2008, SQL Server reserves only a small portion of AWE-mapped memory when it starts. As additional AWE-mapped memory is required, the operating system dynamically allocates it to SQL Server. Similarly, if fewer resources are required, SQL Server can return AWE-mapped memory to the operating system for use by other processes or applications.

Use of AWE, in either Windows Server 2003 or Windows Server 2008, locks the pages in memory so that they cannot be written to the paging file. Windows has to swap out other applications if additional physical memory is needed, so the performance of those applications might suffer. You should therefore set a value for Max Server Memory when you have also enabled AWE.

If you are running multiple instances of SQL Server on the same computer, and each instance uses AWE-mapped memory, you should ensure that the instances perform as expected. Each instance should have a Min Server Memory setting. Because AWE-mapped memory cannot be swapped out to the page file, the sum of the Min Server Memory values for all instances should be less than the total physical memory on the computer.

If your SQL Server is set up for failover clustering and is configured to use AWE memory, you must ensure that the sum of the Max Server Memory settings for all the instances is less than the least physical memory available on any of the servers in the cluster. If the failover node has less physical memory than the original node, the instances of SQL Server may fail to start.

User Connections SQL Server 2008 dynamically adjusts the number of simultaneous connections to the server if the User Connections configuration setting is left at its default of 0. Even if you set this value to a different number, SQL Server does not actually allocate the full amount of memory needed for each user connection until a user actually connects. When SQL Server starts, it allocates an array of pointers with as many entries as the configured value for User Connections. If you must use this option, do not set the value too high because each connection takes approximately 28 KB of overhead regardless of whether the connection is being used. However, you also don't want to set it too low because if you exceed the maximum number of user connections, you receive an error message and cannot connect until another connection becomes available. (The exception is the DAC connection, which can be used.) Keep in mind that the User Connections value is not the same as the number of users; one user, through one application, can open multiple connections to SQL Server. Ideally, you should let SQL Server dynamically adjust the value of the User Connections option.

Important The Locks configuration option is a setting from earlier versions, and it has been deprecated. This setting is ignored in SQL Server 2008, even though you do not receive an error message when you try to use this value.

Scheduling Options

As described previously, SQL Server 2008 has a special algorithm for scheduling user processes using the SQLOS, which manages one scheduler per logical processor and makes sure that only one process can run on a scheduler at any given time. The SQLOS manages the assignment of user connections to workers to keep the number of users per CPU as balanced as possible. Five configuration options affect the behavior of the scheduler: Lightweight Pooling, Affinity Mask, Affinity64 Mask, Priority Boost, and Max Worker Threads.

Affinity Mask and Affinity64 Mask From an operating system point of view, the ability of Windows to move process threads among different processors is efficient, but this activity can reduce SQL Server performance because each processor cache is reloaded with data repeatedly. By setting the Affinity Mask option, you can allow SQL Server to assign processors to specific threads and thus improve performance under heavy load conditions by eliminating processor reloads and reducing thread migration and context switching across processors. Setting an affinity mask to a *non-0* value not only controls the binding of schedulers to processors, but it also allows you to limit which processors are used for executing SQL Server requests.

The value of an affinity mask is a 4-byte integer, and each bit controls one processor. If you set a bit representing a processor to 1, that processor is mapped to a specific scheduler. The 4-byte affinity mask can support up to 32 processors. For example, to configure SQL Server to use processors 0 through 5 on an eight-way box, you would set the affinity mask to 63, which is equivalent to a bit string of 00111111. To enable processors 8 through 11 on a 16-way box, you would set the affinity mask to 3840, or 0000111100000000. You might want to do this on a machine supporting multiple instances, for example. You would set the affinity mask of each instance to use a different set of processors on the computer.

To cover more than 32 CPUs, you configure a 4-byte affinity mask for the first 32 CPUs and up to a 4-byte Affinity64 mask for the remaining CPUs. Note that affinity support for servers with 33 to 64 processors is available only on 64-bit operating systems.

You can configure the affinity mask to use all the available CPUs. For an eight-way machine, an Affinity Mask setting of 255 means that all CPUs will be enabled. This is not exactly the same as a setting of 0 because with the *nonzero* value, the schedulers are bound to a specific CPU, and with the *0* value, they are not.

Lightweight Pooling By default, SQL Server operates in thread mode, which means that the workers processing SQL Server requests are threads. As we described earlier, SQL Server also lets user connections run in fiber mode. Fibers are less expensive to manage than threads. The Lightweight Pooling option can have a value of 0 or 1; 1 means that SQL Server should run in fiber mode. Using fibers may yield a minor performance advantage, particularly when you have eight or more CPUs and all of the available CPUs are operating at or near 100 percent. However, the trade-off is that certain operations, such as running queries on linked servers or executing extended stored procedures, must run in thread mode and therefore need to switch from fiber to thread. The cost of switching from fiber to thread mode for those connections can be noticeable and in some cases offsets any benefit of operating in fiber mode.

If you're running in an environment using a high percentage of total CPU resources, and if System Monitor shows a lot of context switching, setting Lightweight Pooling to 1 might yield some performance benefit.

Priority Boost If the Priority Boost setting is enabled, SQL Server runs at a higher scheduling priority. The result is that the priority of *every* thread in the server process is set to a priority of 13 in Windows 2000 and Windows Server 2003. Most processes run at the normal priority, which is 7. The net effect is that if the server is running a very resource-intensive workload and is getting close to maxing out the CPU, these normal priority processes are effectively starved.

The default Priority Boost setting is 0, which means that SQL Server runs at normal priority whether or not you're running it on a single-processor machine. There are probably very few sites or applications for which setting this option makes much difference, but if your machine is totally dedicated to running SQL Server, you might want to enable this option (setting it to 1) to see for yourself. It can potentially offer a performance advantage on a heavily loaded, dedicated system. As with most of the configuration options, you should use it with care. Raising the priority too high might affect the core operating system and network operations, resulting in problems shutting down SQL Server or running other operating system tasks on the server.

Max Worker Threads SQL Server uses the operating system's thread services by keeping a pool of workers (threads or fibers) that take requests from the queue. It attempts to divide the worker threads evenly among the SQLOS schedulers so that the number of threads available to each scheduler is the Max Worker Threads setting divided by the number of CPUs. With 100 or fewer users, there are usually as many worker threads as active users (not just connected users who are idle). With more users, it often makes sense to have fewer worker threads than active users. Although some user requests have to wait for a worker thread to become available, total throughput increases because less context switching occurs.

The Max Worker Threads default value of 0 means that the number of workers is configured by SQL Server, based on the number of processors and machine architecture. For example, for a four-way 32-bit machine running SQL Server, the default is 256 workers. This does not mean that 256 workers are created on startup. It means that if a connection is waiting to be serviced and no worker is available, a new worker is created if the total is currently below 256. If this setting is configured to 256 and the highest number of simultaneously executing commands is, say, 125, the actual number of workers will not exceed 125. It might be even smaller than that because SQL Server destroys and trims away workers that are no longer being used. You should probably leave this setting alone if your system is handling 100 or fewer simultaneous connections. In that case, the worker thread pool will not be greater than 100.

Table 1-5 lists the default number of workers given your machine architecture and number of processors. (Note that Microsoft recommends 1024 as the maximum for 32-bit operating systems.)

Table 1-5: Default Settings for Max Worker Threads

CPU	32-Bit Computer	64-Bit Computer
Up to 4 processors	256	512
8 processors	288	576

16 processors	352	704
32 processors	480	960

Even systems that handle 4,000 or more connected users run fine with the default setting. When thousands of users are simultaneously connected, the actual worker pool is usually well below the Max Worker Threads value set by SQL Server because from the perspective of the database, most connections are idle even if the user is doing plenty of work on the client.

Disk I/O Options

No options are available for controlling the disk read behavior of SQL Server. All the tuning options to control read-ahead in previous versions of SQL Server are now handled completely internally. One option is available to control disk write behavior. This option controls how frequently the checkpoint process writes to disk.

Recovery Interval The Recovery Interval option can be configured automatically. SQL Server setup sets it to 0, which means autoconfiguration. In SQL Server 2008, this means that the recovery time should be less than one minute. This option lets the database administrator control the checkpoint frequency by specifying the maximum number of minutes that recovery should take, per database. SQL Server estimates how many data modifications it can roll forward in that recovery time interval. SQL Server then inspects the log of each database (every minute, if the recovery interval is set to the default of 0) and issues a checkpoint for each database that has made at least that many data modification operations since the last checkpoint. For databases with relatively small transaction logs, SQL Server issues a checkpoint when the log becomes 70 percent full, if that is less than the estimated number.

The Recovery Interval option does not affect the time it takes to undo long-running transactions. For example, if a long-running transaction takes two hours to perform updates before the server becomes disabled, the actual recovery takes considerably longer than the Recovery Interval value.

The frequency of checkpoints in each database depends on the amount of data modifications made, not on a time-based measure. So a database that is used primarily for read operations will not have many checkpoints issued. To avoid excessive checkpoints, SQL Server tries to make sure that the value set for the recovery interval is the minimum amount of time between successive checkpoints.

As discussed previously, most writing to disk doesn't actually happen during checkpoint operations. Checkpoints are just a way to guarantee that all dirty pages not written by other mechanisms are still written to the disk in a timely manner. For this reason, you should keep the Recovery Interval value set at 0 (self-configuring).

Affinity I/O Mask and Affinity64 I/O Mask These two options control the affinity of a processor for I/O operations and work in much the same way as the two options for controlling processing affinity for workers. Setting a bit for a processor in either of these bit masks means that the corresponding processor is used *only* for I/O operations. You probably never need to set this option. However, if you do decide to use it, perhaps just for testing purposes, you should use it in conjunction with the Affinity Mask or Affinity64 Mask option and make sure the bits set do not overlap. You should thus have one of the following combinations of settings: 0 for both Affinity I/O Mask and Affinity Mask for a CPU, 1 for the Affinity I/O Mask option and 0 for Affinity Mask, or 0 for Affinity I/O Mask and 1 for Affinity Mask.

Backup Compression Default Backup Compression is a new feature in SQL Server 2008, and for backward compatibility, the default value for backup compression is 0, meaning that backups are not compressed. Although only Enterprise edition instances can create a compressed backup, any edition of SQL Server 2008 can restore a compressed backup. When Backup Compression is enabled, the compression is performed on the server prior to writing, so it can greatly reduce the size of the backups and the I/O required to write the backups to the external device. The amount of space reduction depends on many factors, including the following:

- **The type of data in the backup** For example, character data compresses more than other types of data.
- **Whether the data is encrypted** Encrypted data compresses significantly less than equivalent unencrypted data. If transparent data encryption is used to encrypt an entire database, compressing backups might not reduce their size by much, if at all.

After the backup has been performed, you can inspect the *backupset* table in the *msdb* database to determine the compression ratio, using a statement like the following:

```
SELECT backup_size/compressed_backup_size FROM msdb..backupset;
```

Although compressed backups can use significantly fewer I/O resources, it can significantly increase CPU usage when performing the compression. This additional load can affect other operations occurring concurrently. To minimize this impact, you can consider using the Resource Governor to create a workload group for sessions performing backups and assign the group to a resource pool with a limit on its maximum CPU utilization.

The configured value is the instance-wide default for Backup Compression, but it can be overridden for a particular backup operation, by specifying WITH COMPRESSION or WITH NO_COMPRESSION. Compression can be used for any type of backup: full, log, differential or partial (file or filegroup).

Note The algorithm used for compressing backups is very different than the database compression algorithms. Backup Compression uses an algorithm very similar to zip, where it is just looking for patterns in the data. Data compression will be discussed in Chapter 7.

Filestream Access Level Filestream integrates the Database Engine with your NTFS file system by storing BLOB data as files on the file system and allowing you to access this data either using T-SQL or Win32 file system interfaces to provide streaming access to the data. Filestream uses the Windows system cache for caching file data to help reduce any effect that filestream data might have on SQL Server performance. The SQL Server buffer pool is not used so that filestream does not reduce the memory available for query processing.

Prior to setting this configuration option to indicate the access level for filestream data, you must enable *FILESTREAM* externally using the SQL Server Configuration Manager (if you haven't enabled *FILESTREAM* during SQL Server setup). Using the SQL Server Configuration Manager, you can right-click the name of the SQL Server service and choose properties. The dialog box has a separate tab for *FILESTREAM* options. You must check the top box to enable *FILESTREAM* for T-SQL access, and then you can choose to enable *FILESTREAM* for file I/O streaming if you want.

After enabling *FILESTREAM* for your SQL Server instance, you then set the configuration value. The following values are allowed:

- **0 Disables FILESTREAM** support for this instance
- **1 Enables FILESTREAM** for T-SQL access
- **2 Enables FILESTREAM** for T-SQL and Win32 streaming access

Databases that store filestream data must have a special filestream filegroup. We'll discuss filegroups in Chapter 3. More details about filestream storage will be covered in Chapter 7.

Query Processing Options

SQL Server has several options for controlling the resources available for processing queries. As with all the other tuning options, your best bet is to leave the default values unless thorough testing indicates that a change might help.

Min Memory Per Query When a query requires additional memory resources, the number of pages that it gets is determined partly by the Min Memory Per Query option. This option is relevant for sort operations that you specifically request using an ORDER BY clause, and it also applies to internal memory needed by merge-join operations and by hash-join and hash-grouping operations. This configuration option allows you to specify a minimum amount of memory (in kilobytes) that any of these operations should be granted before they are executed. Sort, merge, and hash operations receive memory in a very dynamic fashion, so you rarely need to adjust this value. In fact, on larger machines, your sort and hash queries typically get much more than the Min Memory Per Query setting, so you shouldn't restrict yourself unnecessarily. If you need to do a lot of hashing or sorting, however, and you have few users or a lot of available memory, you might improve performance by adjusting this value. On smaller machines, setting this value too high can cause virtual memory to page, which hurts server performance.

Query Wait The Query Wait option controls how long a query that needs additional memory waits if that memory is not available. A setting of –1 means that the query waits 25 times the estimated execution time of the query, but it always waits at least 25 seconds with this setting. A value of 0 or more specifies the number of seconds that a query waits. If the wait time is exceeded, SQL Server generates error 8645:

```
Server: Msg 8645, Level 17, State 1, Line 1
A time out occurred while waiting for memory resources to execute the query. Re-run the
query.
```

Even though memory is allocated dynamically, SQL Server can still run out of memory if the memory resources on the machine are exhausted. If your queries time out with error 8645, you can try increasing the paging file size or even adding more physical memory. You can also try tuning the query by creating more useful indexes so that hash or merge operations aren't needed. Keep in mind that this option affects only queries that have to wait for memory needed by hash and merge operations. Queries that have to wait for other reasons are not affected.

Blocked Process Threshold This option allows an administrator to request a notification when a user task has been blocked for more than the configured number of seconds. When Blocked Process Threshold is set to 0, no notification is given. You can set any value up to 86,400 seconds. When the deadlock monitor detects a task that has been waiting longer than the configured value, an internal event is generated. You can choose to be notified of this event in one of two ways. You can use SQL Trace to create a trace and capture event of type Blocked process report, which you can find in the Errors and Warnings category on the Events Select screen in SQL Server Profiler. So long as a resource stays blocked on a deadlock-detectable resource, the event is raised every time the deadlock monitor checks for a deadlock. An Extensible Markup Language (XML) string is captured in the Text Data column of the trace that describes the blocked resource and the resource being waited on. More information about deadlock detection is in Chapter 10.

Alternatively, you can use event notifications to send information about events to a service broker service. Event notifications can provide a programming alternative to defining a trace, and they can be used to respond to many of the same events that SQL Trace can capture. Event notifications, which execute asynchronously, can be used to perform an action inside an instance of SQL Server 2008 in response to events with very little consumption of memory resources. Because event notifications execute asynchronously, these actions do not consume any resources defined by the immediate transaction.

Index Create Memory The Min Memory Per Query option applies only to sorting and hashing used during query execution; it does not apply to the sorting that takes place during index creation. Another option, Index Create Memory, lets you allocate a specific amount of memory for index creation. Its value is specified in kilobytes.

Query Governor Cost Limit You can use the Query Governor Cost Limit option to specify the maximum number of seconds that a query can run. If you specify a nonzero, non-negative value, SQL Server disallows execution of any query that has an estimated cost exceeding that value. Specifying 0 (the default) for this option turns off the query governor, and all queries are allowed to run without any time limit.

Max Degree Of Parallelism and Cost Threshold For Parallelism SQL Server 2008 lets you run certain kinds of complex queries simultaneously on two or more processors. The queries must lend themselves to being executed in sections. Here's an example:

```
SELECT AVG(charge_amt), category
FROM charge
GROUP BY category
```

If the charge table has 1,000,000 rows and there are 10 different values for *category*, SQL Server can split the rows into groups and have only a subset of the groups processed on each processor. For example, with a four-CPU machine, categories 1 through 3 can be averaged on the first processor, categories 4 through 6 can be averaged on the second processor, categories 7 and 8 can be averaged on the third, and categories 9 and 10 can be averaged on the fourth. Each processor can come up with averages for only its groups, and the separate averages are brought together for the final result.

During optimization, the Query Optimizer always finds the cheapest possible serial plan before considering parallelism. If this serial plan costs less than the configured value for the Cost Threshold For Parallelism option, no parallel plan is generated. Cost Threshold For Parallelism refers to the cost of the query in seconds; the default value is 5. If the cheapest serial plan costs more than this configured threshold, a parallel plan is produced based on assumptions about how many processors and how much memory will actually be available at runtime. This parallel plan cost is compared with the serial plan cost, and the cheaper one is chosen. The other plan is discarded.

A parallel query execution plan can use more than one thread; a serial execution plan, which is used by a nonparallel query, uses only a single thread. The actual number of threads used by a parallel query is determined at query plan execution initialization and is the DOP. The decision is based on many factors, including the Affinity Mask setting, the Max Degree Of Parallelism setting, and the available threads when the query starts executing.

You can observe when SQL Server is executing a query in parallel by querying the DMV *sys.dm_os_tasks*. A query that is running on multiple CPUs has one row for each thread, as follows:

```
SELECT
    task_address,
    task_state,
    context_switches_count,
    pending_io_count,
    pending_io_byte_count,
    pending_io_byte_average,
    scheduler_id,
    session_id,
    exec_context_id,
    request_id,
    worker_address,
    host_address
FROM sys.dm_os_tasks
ORDER BY session_id, request_id;
```

Be careful when you use the Max Degree Of Parallelism and Cost Threshold For Parallelism options—they have server-wide impact.

There are other configuration options that we will not mention, most of which deal with aspects of SQL Server that are beyond the scope of this book. These include options for configuring remote queries, replication, SQL Agent, C2 auditing, and full-text search. There is a Boolean option to disallow use of the CLR in programming SQL Server objects; it is off (0) by default. The Allow Updates option still exists but has no effect in SQL Server 2008. A few of the configuration options deal with programming issues, and you can get more details in *Inside SQL Server 2008: TSQL Programming*. These options include ones for dealing with recursive and nested triggers, cursors, and accessing objects across databases.

The Default Trace

One final option that doesn't seem to fit into any of the other categories is called *Default Trace Enabled*. We mention it because the default value is 1, which means that as soon as SQL Server starts, it runs a server-side trace, capturing a predetermined set of information into a predetermined location. None of the properties of this default trace can be changed; the only thing you can do is turn it off.

You can compare the default trace to the blackbox trace which has been available since SQL Server 7 (and is still available in SQL Server 2008), but the blackbox trace takes a few steps to create, and it takes even more steps to have it start automatically when your SQL Server starts. This default trace is so lightweight that you might find little reason to disable it. If you're not familiar with SQL Server tracing, you'll probably need to spend some time reading about tracing in Chapter 2.

The default trace output file is stored in the same directory in which you installed SQL Server, in the \Log subdirectory. So if you've installed SQL Server in the default location, the captured trace information for a default instance will be in the file C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSSERVER\MSSQL\LOG\Log.trc. Every time you stop and restart SQL Server, or reach the maximum file size of 20 MB, a new trace file is created with a sequential numerical suffix, so the second trace file would be Log_01.trc, followed by Log_02. trc, and so on. If all the trace log files are removed or renamed, the next trace file starts at log.trc again. SQL Server will keep no more than five trace files per instance, so when the sixth file is created, the earliest one is deleted.

You can open the trace files created through the default trace mechanism by using the SQL Server Profiler, just as you can any other trace file, or you can copy to a table by using the system function *fn_trace_gettable* and view the current contents of the trace while the trace is still running. As with any server-side trace that writes to a file, the writing is done in 128-KB blocks. Thus, on a very low-use SQL Server instance, it might look like nothing is being written to the file for quite some time. You need 128 KB of data for any writes to the physical file to occur. In addition, when the SQL Server service is stopped, whatever events have accumulated for this trace will be written out to the file.

Unlike the blackbox trace, which captures every single batch completely and can get huge quickly, the default trace in SQL Server 2008 captures only a small set of events that were deemed likely to cause stability problems or performance degradation of SQL Server. The events include database file size change operations, error and warning conditions, full-text crawl operations, object *CREATE, ALTER,* and *DROP* operations, changes to permissions or object ownership, and memory change events.

Not only can you not change anything about the files saved or their locations, you can't add or remove events, the data captured along with the events, or the filters that might be applied to the events. If you want something slightly different than the default trace, you can disable the predefined trace and create your own with whatever events, data, and filters you choose. Of course, you must then make sure the trace starts automatically. This is not impossible to do, but we suggest that you leave the default trace on, in addition to whatever other traces you need, so that you know that at least some information about the activities taking place on your SQL Server is being captured.

Final Words

In this chapter, I've looked at the general workings of the SQL Server engine, including the key components and functional areas that make up the engine. I've also looked at the interaction between SQL Server and the operating system. By necessity, I've made some simplifications throughout the chapter, but the information should provide some insight into the roles and responsibilities of the major components in SQL Server and the interrelationships among components.

This chapter also covered the primary tools for changing the behavior of SQL Server. The primary means of changing the behavior is by using configuration options, so we looked at the options that can have the biggest impact on SQL Server behavior, especially its performance. To really know when changing the behavior is a good idea, it's important that you understand how and why SQL Server works the way it does. My hope is that this chapter has laid the groundwork for you to make informed decisions about configuration changes.