Microsoft SQL Server Query Processor Internals and Architecture

SQL Server 7.0 7 out of 10 rated this helpful

Hal Berenson and Kalen Delaney

January 2000

Summary: This article examines how Microsoft SQL Server queries are processed on the client, how the various clients interact with SQL Server, and what SQL Server does to process clients' requests. (26 printed pages)

Introduction

Microsoft® SQL Server™ internals and architecture make up a huge topic, so we'll limit this article to areas of interest to developers, focusing on topics that are not thoroughly discussed in other sources. In discussing the architecture of SQL Server, we'll look primarily at how things are processed on the client, how the various clients interact with SQL Server, and what SQL Server does to process clients' requests. There are other sources of information available that discuss other aspects of SQL Server. In particular, *Inside SQL Server 7.0*, by Ron Soukup and Kalen Delaney, published by Microsoft Press, goes into great detail on the storage engine's mechanisms and behavior, but its coverage of the query processor is not quite so in-depth. This article fills in some of that gap.

Hopefully, the information presented here will teach you some things that will allow you to write better applications. You'll be able to look at performance problems in a new light, with a new understanding.

SQL Server Is a Client/Server System

It's been said for many years that SQL Server is a client/server system. In fact, the Sybase DataServer, from which the original SQL Server was developed, was the first commercial relational database system to be initially developed as a client/server system. But what does that mean? It means more than just that SQL Server is two-tier system. Traditionally, a two-tier system means that the client application runs on one machine and sends requests to a server located on another machine. To SQL Server, client/server means that a *piece* of SQL Server, the client API portion, sits somewhere remotely in the process structure, separate from the server component itself.

In the classic two-tier model, the client portion sits on the desktop machine, with large amounts of client application logic and business logic, and generates requests directly to the database system. The client then gets data back from the server in response to its requests.

The same model applies in the three-tier world. SQL Server has been used for many years with transaction processing monitors such as BEA's Tuxedo and Compaq's ACMSxp, which represent the classic three-tier model dating back 20 or 30 years. The three-tier model also dominates today's Web-based applications, as represented by application servers such as Microsoft's MTS and the new COM+ 1.0. From the SQL Server perspective, the client in a three-tier solution is a piece of logic sitting in the *middle* tier. That middle tier interacts directly with the database system. The actual desktop, or thin client, uses some other mechanism and typically interacts directly with the middle tier, not directly with the database system. This structure is illustrated in Figure 1.

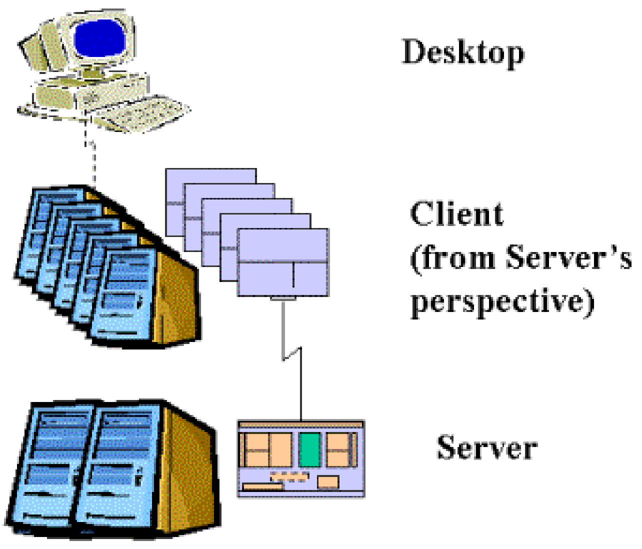

Figure 1. Three-tier system model

Client Architecture

From an architectural standpoint, the SQL Server Relational Server component doesn't really concern itself with where a client is running. In fact, if you are running your application on the same box where SQL Server is running, it's the same client/server model, as far SQL Server is concerned. The server runs a separate multithreaded process, and it services requests coming from clients, no matter where those clients are located. The client code itself is a separate DLL that runs inside the client application, and the real interface to SQL Server is the Tabular Data Stream (TDS) protocol that is spoken between the client and the server.

"What is the native interface of SQL Server?" is a frequently asked question. For a long time, many developers were resistant to interfaces like ODBC because they thought that DB-Library, the client API developed by Sybase, was the native interface to SQL Server. In fact, the SQL Server Relational Server itself doesn't have a native API; its interface is TDS, the protocol for the communication stream between the client and server. TDS encapsulates the SQL statements sent from the client to the server, as well as the result sets sent from the server back to the client. Any API that directly processes TDS is a *native* SQL Server interface.

Let's look at the components of the client, as shown in Figure 2. There are pieces of client architecture we won't discuss because they are outside the boundaries of SQL Server itself. But if you are writing an application, you are quite aware of those pieces. The ones that people know most about are the various object models; if you're writing an ASP or Microsoft Visual Basic® application, you interact with the database system through ADO; you don't call a lower-level API like ODBC or OLE-DB directly. ADO maps to OLE-DB and RDO maps to ODBC. So this object model, which is the most commonly used part of the programming model, is not strictly part of SQL Server's client architecture. In addition, there are additional components that can plug in at this level above the SQL Server infrastructure. One example of such components is OLE-DB's Session Pooling Service Provider.

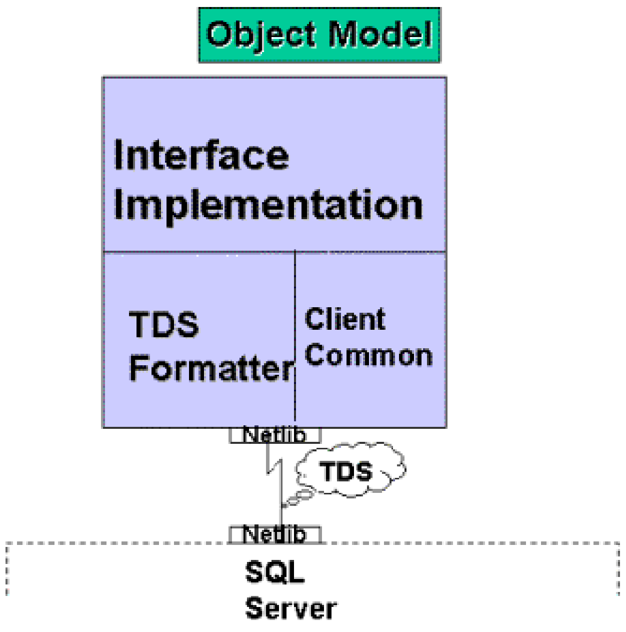

Figure 2. Client architecture

Client Interfaces

There are two interfaces for SQL Server that can be considered native for SQL Server 7.0, OLE-DB and ODBC. The DB-Library interface is also native, in that it speaks TDS, but DB-Library uses an older version of TDS, which requires some conversion at the server. While existing DB-Library applications continue to work against SQL Server 7.0, many new features and performance enhancements are only available through ODBC and OLE DB. Updating DB-Library to support the new capabilities of SQL Server 7.0 would have resulted in many incompatibilities for existing applications, and thus would require application changes. ODBC replaced DB-Library five years ago as the preferred API for new SQL Server applications, so introducing a new and incompatible version of DB-Library did not make sense.

There are three components to any of these client APIs, as you can see in Figure 2, above. The top piece implements the API specifics, such as what a rowset looks like, what cursors look like, and so forth. The TDS formatter takes an actual request, such as a SQL statement, and packages it up as a TDS message, ships it off to the SQL Server, gets back results, and then feeds the results into the interface implementation.

There is also some common library code used by all the providers. For example, the BCP facility is a library that both ODBC and OLE-DB call. DTC is another example. A third case is the ODBC canonical SQL syntax, the CALL syntax with parameter markings, which is common between all the providers.

The TDS protocol is the same for all the APIs, except for the previously mentioned limitation that DB-Library is still using the SQL Server 6.5 version. ODBC and OLE-DB use the SQL Server 7.0 version when speaking to SQL Server 7.0, but they have the ability to talk down to 6.5 or 6.0 servers. There are also the Net-Libraries, the abstraction layer where both client and server can speak a network abstraction interface and not have to worry about IPX or TCP/IP. We won't go into details on the workings of the Net-Libraries; suffice it to say that their job is basically to hide the low-level details of the network communications from the rest of software.

A Client's View of the Server

As already mentioned, a client's primary method of communication with SQL Server is through the use of TDS messages. TDS is a simple protocol. When SQL Server receives a message, it can be thought of as an event occurring. First, a client sends in a login message (or event) on a connection and gets back a success or failure response. When you want to send in a SQL statement, there is a SQL language message that a client can package up and send to SQL Server. Also, when you want to invoke a stored procedure, a system procedure, or a pseudo system stored procedure (which we'll discuss below), the client can send an RPC message, which corresponds to an RPC event on the SQL Server. In these last two cases, the server then sends back results in a token stream of data. Microsoft does not document the actual TDS messages, as this is considered a private contract between SQL Server components.

Catalog stored procedures are another key part of the client/server interaction. These appeared first in SQL Server 6.0 in ODBC, and included such procedures as sp_tables and sp_columns. The ODBC and OLE-DB APIs define standard ways of describing the metadata about database objects. These standards need to be usable against all kinds of RDBMS servers and are not tuned to SQL Server's own system tables. Instead of the client sending multiple queries against the system tables to the server and constructing this standard view of the metadata on the client, a set of system stored procedures was created that sits down on the server and returns the information in the right format for that API. This allows many important metadata requests to be completed in a single round-trip.

The procedures written for ODBC were documented, and they are generally useful for people who want information from the system tables that is not provided by other mechanisms. This allows Transact-SQL procedures and DB-Library applications to access metadata without writing complex queries against the SQL Server system tables, and isolates the application from changes that Microsoft makes to the system tables in the future.

OLE DB defines a set of schema rowsets that are analogous to, but different from, ODBC's metadata. A new set of catalog stored procedures was created to efficiently populate these schema rowsets. However, this new set of stored procedures was not documented because the stored procedures duplicate the earlier functionality. With several existing ways to obtain metadata, the SQL Server team decided to avoid exposing one that added no value to the programming model.

There is a third aspect to how a client interacts with a server. It first appeared in SQL Server 6.0 but was kept relatively quiet. This is the concept of pseudo system stored procedures; they play a very prominent role in SQL Server 7.0. When server-side cursors were first being developed for SQL Server 6.0, the developers had a choice about how they would manage client/server interactions. Cursors did not fit neatly into the available set of TDS messages, because they allowed row after row of data to be returned without the client specifying additional SQL statements. The developers could have added additional messages to the TDS protocol, but then too many other components would need to be changed. The version of TDS in SQL Server 6.0 also needed to stay close to the Sybase version in order to guarantee interoperability, and so the developers came up with alternative mechanism. They made the new (server-side cursor) features appear to be system stored procedures when in reality the stored procedures are just entry points into the SQL Server code base. They are invoked from a client application using standard RPC TDS messages. They are called pseudo system stored procedures, because they are invoked at the client like a stored procedure, but they are not made up of simple SQL statements like other stored procedures. Most of these pseudo system stored procedures are private and undocumented. In the case of the cursor procedures, all the APIs expose their *own* set of cursor API models and their own cursor manipulation functions, so there is no reason to document the stored procedures themselves. Even in the Transact-SQL language there is syntax that exposes the cursors, using DECLARE, OPEN, FETCH, and so forth, so there was no need at all to document the pseudo system stored procedures, such as sp_cursor, that are only used internally.

ODBC and OLE DB expose the concept of parameterized queries and the Prepare/Execute model. Prior to SQL Server 7.0, these concepts were implemented by code in the client API. In SQL Server 7.0, Microsoft added support to the Relational Server for these concepts and exposed that support through new pseudo system stored procedures. These features, and how the server supports them, are explored later in this article. Support for parameterized queries, through the sp_executesql procedure, was deemed particularly useful for direct Transact-SQL and DB-Library use and this procedure was documented. The procedures for prepare/execute are used exclusively by the ODBC driver and OLE DB provider.

All clients that can communicate with SQL Server, then, are built on these three sets of functionality: the TDS protocol, the catalog stored procedures, and the pseudo system stored procedures.

Server Architecture

SQL Server, or more specifically the SQL Server Relational Server, is frequently described as being composed on two main halves, the relational engine and the storage engine. As mentioned earlier, there are many storage engine details documented elsewhere, so this article concentrates on describing the functionality of the relational engine. Figure 3 illustrates the main components of the relational engine portion of SQL Server. The illustrated components can be organized into three groupings of subsystems. On the left are the components that address compiling queries, including the query optimizer. The optimizer is one of the most mysterious parts of any relational database engine but is also one of the most critical from a performance standpoint. It is the responsibility of the query optimizer to take the nonprocedural request expressed in SQL and translate it into a set of disk I/Os, filtering, and other procedural logic that efficiently satisfies the request. On the right-hand side is the execution infrastructure. This is really a much smaller set of facilities. Once the compilation components have finished their job, they have created something that can be directly executed with just a few services thrown in.

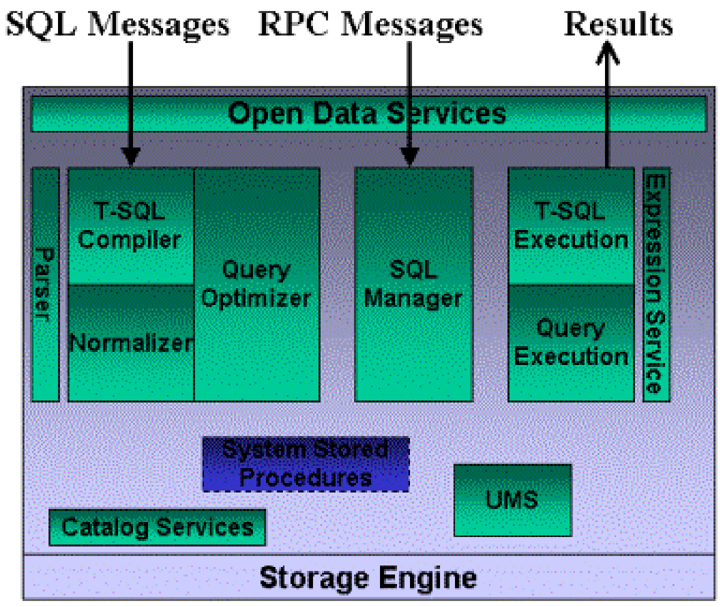

Figure 3. Server architecture

In the middle of the figure is something called the SQL Manager. It controls the flow of everything inside the SQL Server. RPC messages are handled by the SQL Manager, and in SQL Server 7.0 that is the majority of invocations of functionality coming from clients. The pseudo system stored procedures, which we discussed in the previous section, are also logically a part of the SQL Manager. SQL statements typically coming in as TDS SQL Language messages are processed directly from the compilation side; this is less common in SQL Server 7.0 than in previous versions, but is still common. Results are sent back out by components in the execution engine calling ODS to format the TDS results messages.

The majority of that output comes from the execution side of the figure and the results really come out of the expression service. The Expression Services library is a component that does data conversion, predicate evaluation (filtering), and arithmetic calculations. It also works with the ODS layer to format output results into TDS messages.

There are also a couple of components that we'll only mention briefly, which provide additional services within the relational engine. One of these components is the catalog services component, which handles data such definition statements as CREATE TABLE, CREATE VIEW, and so forth. Catalog services also handles system tables, materializing those that are really pseudo tables. The catalog services component is located primarily in the relational engine, but actually about one-third of it operates within the sphere of the storage engine, so it is treated as a shared component.

Another component of the relational engine is the User Mode Scheduler (UMS), SQL Server's own internal scheduler for fibers and threads. There is a very sophisticated internal mechanism for scheduling how work is assigned to either fibers or threads, depending on how you've configured the server, and allows SQL Server to do the proper load balancing across processors on an SMP system. The UMS also keeps SQL Server from thrashing by running too many threads concurrently. Finally, there are the system procedures that people are familiar with; logically they are part of the relational engine. They clearly aren't server code, because you can easily use sp_helptext to examine the Transact-SQL code that defines these procedures. However, the system procedures are treated as part of the server because their purpose is to expose primitive server capabilities, like the system tables, at a higher and more appropriate level for application use. If application developers have used the higher-level system procedures—which are also easier to use—as an interface, their applications will continue to work even if the primitive-level system tables are changed from release to release.

Client/Server Interactions When Processing SQL

We'll now take a look at what happens on the client side when a client application is interacting with the SQL Server. Here is an example of an ODBC call:

```
SQLExecDirect(hstmt, "SELECT * FROM parts where partid = 7", SQL_NTS)
```

(There is an almost direct equivalent of this call for OLE-DB, which we won't look at because the processing is practically identical to the ODBC call.) This ODBC call takes a SQL statement and sends it to the SQL Server for execution.

In this particular query, we are selecting all the columns from the parts table, for the row(s) that have a particular part ID. This is a classic example of ad hoc SQL. In releases prior to SQL Server 7.0, one of the distinguishing characteristics of ad hoc SQL as compared to a stored procedure was that the plans generated by the query optimizer were never cached. The queries would come in, be compiled, be executed, and then the plan would be thrown away. In SQL Server 7.0, as we'll be discussing a bit later, there is actually a mechanism whereby the plans for ad hoc queries can be cached.

Before this statement can be sent to the SQL Server, there are questions that must be asked. Clients all provide some notion of cursors, so one of the questions that must be asked by the client internally is what kind of result set or what kind of cursor is the programmer asking for. The fastest type is what the documentation calls a *default result set*. This type of cursor has also been historically called a firehouse cursor and sometimes it isn't even thought of as a cursor at all. After the SQL request is sent to the server, the server starts sending results back to the client and won't stop sending results until the client has consumed the entire set. This is like a giant firehouse pumping data out at the client.

Once the client has determined that it's a default result set, the next step is to determine if there are any parameter markers. One of the options when using this SQLExecDirect call in ODBC (and its equivalent in OLE-DB) is that, instead of supplying a specific value like 7 in the WHERE clause, you can pass in a parameter marker by replacing the constant with a question mark, as shown here:

```
SQLExecDirect(hstmt, "SELECT * FROM parts where partid = ?", SQL_NTS)
```

Note that you must separately provide the actual value of the parameter.

The client needs to know if there are any parameter markers present in this SQL statement, or is it true ad hoc, non-parameterized SQL. That will affect what the client does with this statement internally and determines what is actually sent as messages to the SQL Server. In the case where there is no question mark, it is clear that the client simply wants to send this request as SQL Language TDS messages, and then the client will sit at the end of the firehose and take the results back. The client can then return the result to the application based on the application's parameters. The client's internal processing choices can be a little obscure in terms of what you request through the ODBC or OLE DB APIs. For example, an application program doesn't directly request a default result set. Instead, in ODBC, if you ask for a cursor that is read-only and forward-only and gives you one row at a time, that defines it to be a firehose cursor (a default result set) as far as the client internals goes.

There is one main problem with a firehose cursor. The client can't send any other SQL statements down to the server until it has consumed all the rows. Because the result set may have a very large number of rows, some applications won't work well with firehose cursors. Fast forward-only cursors, described later, are a new SQL Server 7.0 feature intended specifically to address this case.

Prior to SQL Server version 7.0, the SQLExecDirect call would be processed in much the same way whether or not parameter markers were substituted for the constant. If you specified a parameter marker, the client would actually take the value that you supplied through a different call (the value "7" in the example at the beginning of this section) and plug it in where the question mark was. The new statement with the substituted value was then sent down as an ad hoc SQL statement. There was no benefit of using parameterized SQL at the server.

In SQL Server 7.0, however, if parameter markers are used with SQLExecDirect, the TDS message sent down to SQL Server isn't a SQL language message. Instead, it's sent down to server using the sp_executesql procedure, so it's an RPC as far as the TDS protocol is concerned. At the client, the result is basically the same. The client will get the firehose of data back.

If you don't want this firehose of data back, you can always use a block or a scrollable cursor. In this case, the flow becomes very different. A call is made to the sp_cursoropen entry point (one of these pseudo-stored procedures) passing in the SQL text. The sp_cursoropen manipulates the SQL to add additional logic to enable it to scroll, it potentially redirects some results into a temp table, and then it gives a response with a handle to the cursor indicating that the cursor is now open. Still outside of the programmer's control, the client calls sp_cursorfetch, brings down one or more rows to the client that are then returned to the user application. The client can also use sp_cursor to reposition the cursor, or change certain statistics. When you're done processing the cursor, the client will call sp_cursorclose.

Let's examine a simple case where we're just returning one row to the client. In the default result set case, you have one round trip of messages from the client to the server, and back again. There is the SQL message (or sp_executesql) going down to the server, and then the results coming back. In the case of a (nonfirehose) cursor for the same one row, you see what you've traditionally seen with SQL Server. There is a round-trip to do the open, a round-trip to do the fetch, and a round-trip to do the close. The process has used three times as many messages as the default result set would have used. In SQL Server 7.0, there is something called a fast forward-only cursor, which uses the same cursor infrastructure. It doesn't behave like a firehose, because it doesn't require that you process all the result rows prior to sending any additional SQL messages. So if you bring back five rows and there is still more data, you can still send an update down to the server.

A fast forward-only cursor is faster on the server than a regular cursor, and it lets you specify two additional options. One is called autofetch and one is called autoclose. Autofetch will return the first set of rows as part of the response message to the open. Autoclose automatically closes the cursor after the last row is read. Because it is forward-only and read-only, you can't scroll back. SQL Server simply passes a message back with that last set of data saying that the cursor is closed. If you're using fast forward-only cursors, you can get the communication down to the same one round-trip in messages for small numbers of rows. If you have large numbers of rows, you're at least only paying the additional cost for each block of rows. Cursor processing has gotten a lot closer to that default result set if you use fast forward-only cursors.

The flow of the SQLExecDirect model is illustrated in Figure 4.

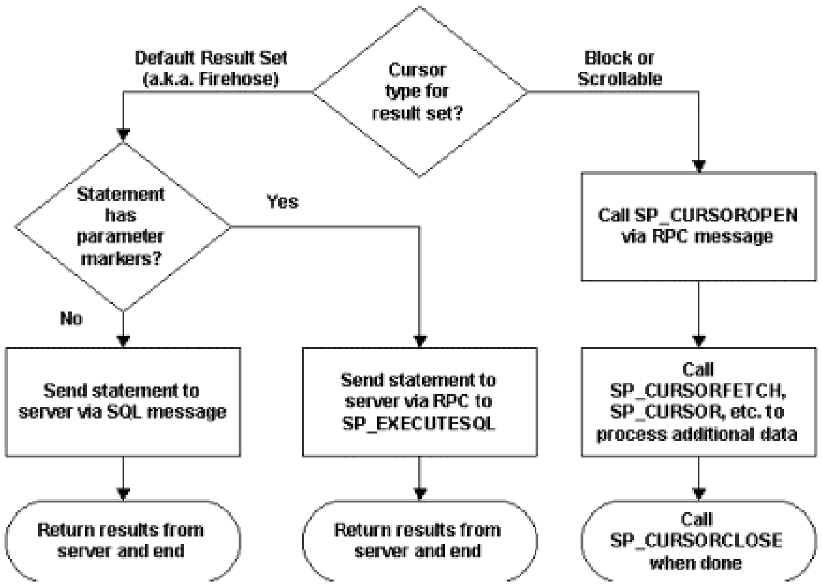
Default Result Set (a.k.a. Firehose)
Cursor type for result set?
Block or Scrollable
Statement has parameter markers?
Yes
No
Call SP_CURSOROPEN via RPC message
Send statement to server via SQL message
Send statement to server via RPC to SP_EXECUTESQL
Call SP_CURSORFETCH, SP_CURSOR, etc. to process additional data
Return results from server and end
Return results from server and end
Call SP_CURSORCLOSE when done

Figure 4. Client/server interactions

The Prepare/Execute Model

In addition to the execute direct model (invoked in ODBC with SQLExecDirect), there is another execution model exposed in ODBC and OLE-DB, called the prepare/execute model. Defining the SQL to be executed is done as a separate step from actually executing the SQL. Here's an example in ODBC:

```
SQLPrepare(hstmt, "SELECT * FROM parts where partid = ?", SQL_NTS)
SQLExecute(hstmt)
```

Prepare/execute was never a native model for SQL Server prior to SQL Server 7.0. Now, in 7.0, there are two pseudo system stored procedures that provide a native interface. For the prepare call, we again take a look at what kind of cursor it is, and then we either call sp_prepare or sp_cursorprepare. That does the compilation part of processing the SQL or the store procedure, but doesn't actually execute the plan. Instead, the pseudo system stored procedure returns a handle to the plan. Now your application can repeatedly re-execute the SQL, passing in different parameter values, for example, without needing to recompile.

In SQL Server 6.5, because there was no native interface, the two phases of prepare and execute had to be emulated. This could happen in one of two ways. In one method, the prepare phase would never really happen. There was just a partial execution to return the metadata (there is a set option to do that) so SQL Server can send the description of the format of the results back to the application. In the other method, SQL Server actually created a temporary stored procedure, which was private to an individual user, so there was no sharing of the plan. This second method could fill up your tempdb database, so most application developers turned off the option to use this second method through a checkbox in the ODBC configuration dialog box.

In SQL Server 7.0, the prepare/execute method is a native feature of SQL Server. After the SQL statement has been prepared, it is executed. In the case of default result sets, that is just by the application programmer calling sp_execute with the handle supplied from the prepare operation, and then the statement runs. In the case of cursors, it looks exactly like the other cursor processing and, in fact, it has the same characteristics, including allowing autofetch and autoclose if the cursor is fast forward-only.

The flow of the prepare/execute operation is illustrated in Figure 5.

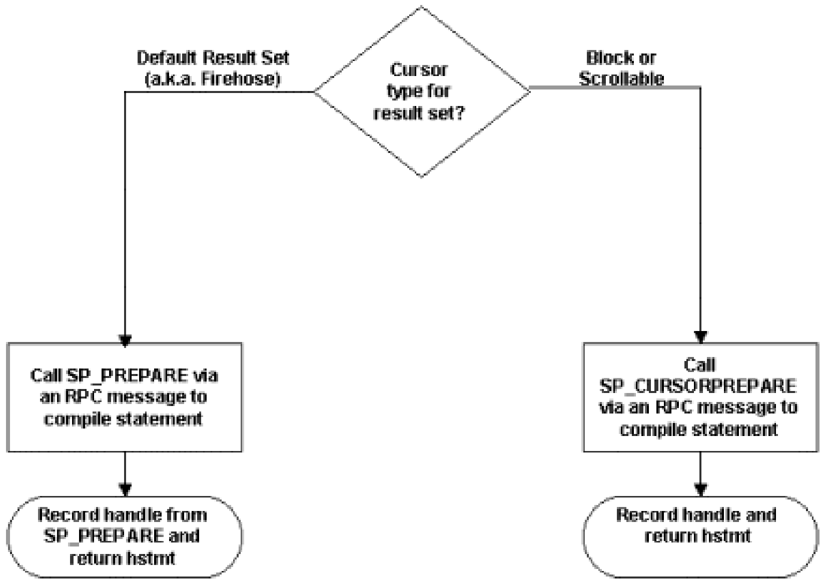
Default Result Set
(a.k.a. Firehose)
Cursor type for result set?
Block or Scrollable
Call SP_PREPARE via an RPC message to compile statement
Record handle from SP_PREPARE and return hstmt
Call SP_CURSORPREPARE via an RPC message to compile statement
Record handle and return hstmt

Figure 5. Prepare/execute model

Calling Stored Procedures

Stored procedures are generally invoked from ODBC and OLE-DB by sending a SQL statement down to the SQL Server that uses the ODBC canonical CALL syntax to call a procedure. It might look something like this:

```
SQLExecDirect(hstm, "{call addorder(?)}", SQL_NTS)
```

In the case of a default result set, it's a simple flow because this is what RPC messages were originally intended for. The client sends an RPC message to the server and gets back the results coming from the procedure. If it's a cursor, it's a little more complicated. The client calls sp_cursoropen, like with any other cursor. Sp_cursoropen has some logic built into it to detect whether the stored procedure contains only a single SELECT statement. If so, a cursor is opened on that SELECT. If it's not a single SELECT statement in the procedure, then the client gets back a message with an indicator that says "we opened this for you, but we're going to stream the results back to you as a firehose and you can present that to the user."

The flow of the execution for stored procedure processing is illustrated in Figure 6.

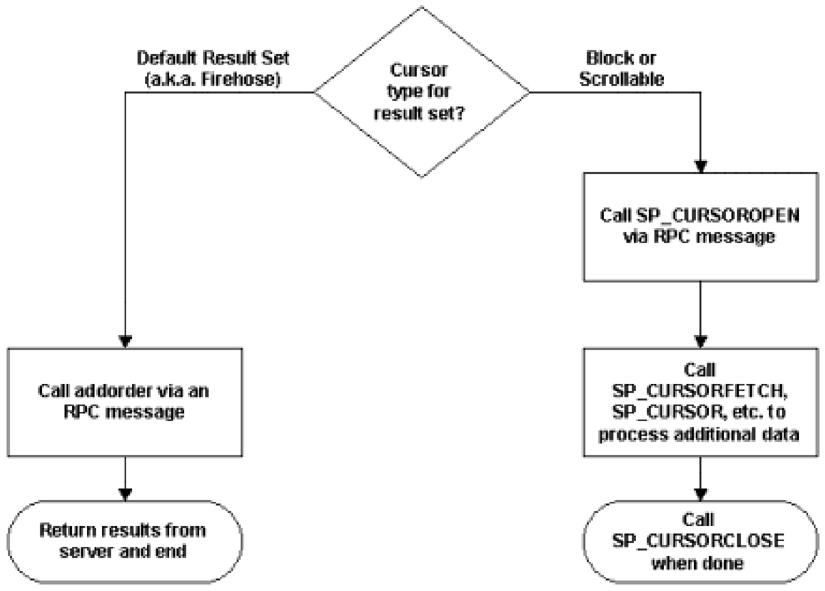
Default Result Set
(a.k.a. Firehose)
Cursor type for result set?
Block or Scrollable
Call SP_CURSOROPEN via RPC message
Call addorder via an RPC message
Call SP_CURSORFETCH, SP_CURSOR, etc. to process additional data
Return results from server and end
Call SP_CURSORCLOSE when done

Figure 6. Calling stored procedures

The SQL Manager

The SQL Manager, which we looked at previously, is the driving force in a lot of the server processing. It's really at the heart of the server. The SQL Manager deals with all of the requests to run stored procedures. It manages the procedure cache, it has the pseudo system stored procedures, and it's involved in auto parameterization of ad hoc queries, which we'll discuss shortly. If you found an article like this one for SQL 6.5 or earlier, you wouldn't really see any discussion of the SQL manager. Instead, you'd see a few different components that do parts of what the SQL manager does. But in SQL Server 7.0, those components have been unified into this one component that really drives the query processing through the system.

Typically, the SQL manager is invoked with an RPC message, as you ask SQL Server to do some work for you. However, when a SQL language statement comes in through a SQL message and goes into the compilation side of the engine, the SQL manager is also involved. It can be involved when a procedure or a batch has an EXEC statement in it, because the EXEC is actually calling the SQL manger. If the SQL statement passes the autoparameterization template, which we'll discuss below, then the SQL manager is called to parameterize the queries. It's also called when ad hoc queries need to be placed in the cache.

Compilation and Execution

Now we'll take a look at how compilation and execution generally flow inside SQL Server. It's important to be aware of the fact that compilation and execution are two distinct phases inside SQL Server. The gap between when SQL Server compiles a query and when it is executed can be very small, a few microseconds, or it can be seconds, minutes, hours or even days. During compilation (which includes optimization), we must distinguish what kind of knowledge we can use as part of the compilation. Not everything that is true at compilation time will be true at execution time. You must think of compilation and execution as two separate activities, even in cases where you're sending in an ad hoc SQL statement and immediately executing it.

When a query is ready to be processed by SQL Server, the SQL Manager looks it up in cache; and if it's not there, it must be compiled. The compilation process encompasses a few things. First, there's parsing and normalization. Parsing is the actual dissecting of your SQL statement, turning it into data structures that can be processed more readily by a computer. Parsing also includes validating that you have legal syntax. Parsing does not include things like checking for valid table and column names. Those things are dealt with during normalization. Normalization is basically intended to resolve those things that you reference inside your SQL into what their actual characteristics are in the database and check whether the semantics you're asking for make sense. For example, it is semantically illogical to try to execute a table.

The next step is the compilation of the Transact-SQL code. Transact-SQL and SQL itself get people a little confused, and the developers at Microsoft use them as interchangeably as anyone else does. However, there is actually an important difference. SQL is all of the DML statements: INSERT, UPDATE, DELETE, and SELECT. SQL Server also has a language that wraps these DML statements, called Transact-SQL, or TSQL. TSQL provides procedural constructs: IF statements, WHILE statements, local variable declarations, and so forth. These are treated very differently inside the server. The procedural logic of TSQL is compiled by an engine that knows how to do procedural tasks.

The SQL statements themselves are compiled by the classic query optimizer. The optimizer has to translate the nonprocedural request of a set-based SQL statement into a procedure that can execute efficiently and return the desired results. Unless stated otherwise, when we talk about compilation, from now on we will mean both the compilation of TSQL and the optimization of the SQL statements.

We mentioned earlier that compilation and execution are two distinct phases of query processing, so one of the things the optimizer does is to optimize based on a fairly stable state. You may realize that SQL Server may recompile a statement depending on certain criteria being met, so it's not a state known to be permanently stable, but it has to be something that is not in a constant state of flux. If the optimizer used information that changes very dramatically and very frequently—the number of concurrent processes or the number of locks held—then queries would constantly must be recompiled, and compilation tends to be slow. For example, you might have a SQL statement that runs in 1/100th of a second, but takes half a second to compile. It would be preferable if SQL Server could compile the statement once, and let people execute it thousands or millions of times, without repaying the compilation cost on every statement.

The end product of the compilation phase is a query plan, which is put into the procedure cache. Cheap ad hoc SQL plans don't really go in the cache, but that's a minor detail for now. We don't want to flood the cache with things that are unlikely to be reused, and ad hoc SQL, on the grand scale of things, is the least likely type of query to have its plan reused. If the statement is already so cheap to compile (less than 1/100th of a second), there is no point in storing the plan in the cache and littering the cache with plans that are unlikely to be reused.

After the plan has been put in the cache, the SQL Manager goes back through the logic in terms of executing it, checking to see if anything has changed, and if the plan needs to be recompiled again. Even though there are only microseconds separating compilation from execution, somebody might have executed a Data Definition Language (DDL) statement that added an index to a crucial table. This is admittedly unlikely, but it is possible, so SQL Server must account for it. There are a few things that will cause SQL Server to recompile a stored plan. Metadata changes, such as adding or dropping indexes, are the most likely reason. The server needs to make sure the plan used reflects the current state of the indexes.

Another cause of recompilation is statistics changes. SQL Server keeps quite a bit of histogram information on the data that it processes. If the data distribution changes a lot, it is likely that a different query plan is needed to get the most efficient execution. SQL Server keeps track of how often data is inserted into or deleted from a table, and if the number of changes exceeds a certain threshold, which changes based on the size of the table, then it will go back and recompile plans based on new distribution information.

The flow of compilation and execution can be seen in Figure 7.

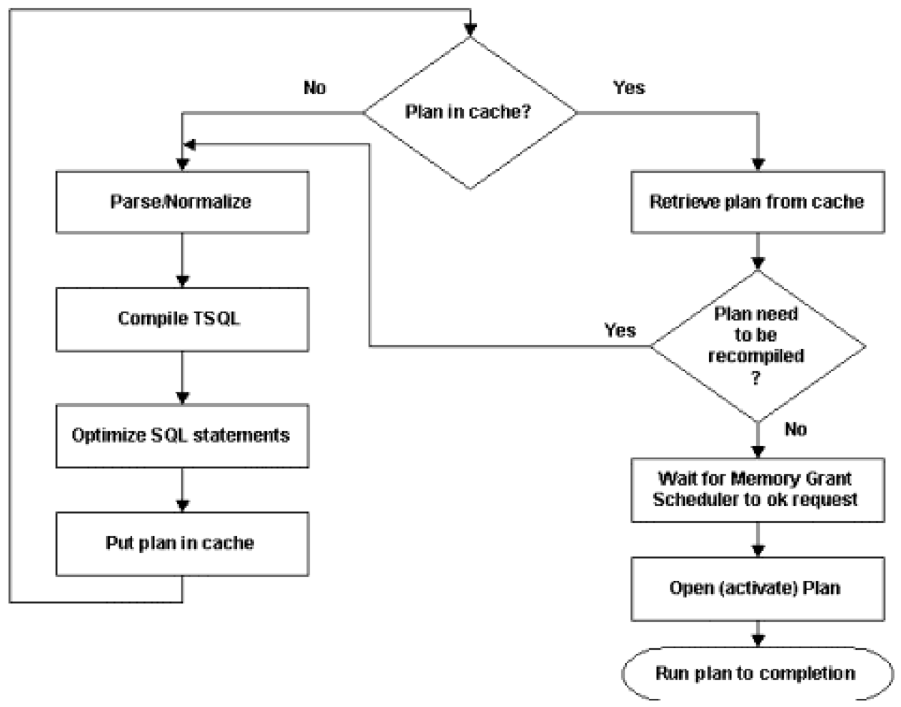

Figure 7. Compilation and execution

Keep in mind that a change of actual parameter values will *not* cause a plan to be recompiled, nor will environmental changes like the amount of memory available or the amount of needed data that is already in cache.

Execution is straightforward, and if all you had were very simple queries like "insert one row," or a singleton select from a table with a unique index, the processing would be very simple. But many queries require large amounts of memory to run efficiently, or at least could benefit from that memory. In SQL Server 6.5, we controlled that by limiting how much memory a query could use to half a megabyte (MB) or 1 MB. There was a configuration parameter to control the amount of memory used by a query, called *sort pages*. The name implies that it was basically only sort operations that could benefit from larger amounts of memory. No matter how large a sort we had to do, in SQL Server 6.5 we were limited to never using more than 1 MB for a particular operation. This was true even if you were working on a machine that had 1 or 2 gigabytes (GB) of memory, and you had a huge sort of millions of rows. Clearly, that was not effective for these complex queries, so the SQL Server developers added the ability in SQL Server 7.0 to use large amounts of memory for individual queries.

Another problem came up at this point. Once you start allowing queries to use large amounts of memory, you have to decide how to arbitrate this memory use among many queries that might need that memory. SQL Server handles this as follows. When a query plan is optimized, the optimizer determines two pieces of information about memory use for that query. First, it picks a minimum amount of memory that the query needs in order to run effectively, and that value is stored with the query plan. The optimizer also determines a maximum amount of memory that the query could benefit from. For example, it doesn't help to give a sort 2 GB of memory if the entire table you're sorting fits into 100 MB. You really just want the 100 MB, and so that would be the value for the maximum amount of useful memory that is stored with the query plan.

When SQL Server goes to execute a plan, the plan is passed to a routine called the memory grant scheduler. The grant scheduler does a few interesting things. First, if the query the grant scheduler is looking at doesn't have a sort or a hash operation as part of the plan, SQL Server knows it doesn't use large amounts of memory. In this case, there is no waiting in the memory grant scheduler at all. The plan can be run immediately, so a typical transaction processing request will bypass this mechanism completely. The memory grant scheduler also has several queues available to handle requests of different sizes. The memory scheduler gives some priority to requests that are smaller. For example, if you have a query that is "select top 10," and you're only going to be sorting 20 rows, it does have to go through the memory grant scheduler, but the query should be released and scheduled pretty quickly. The server wants to run many such queries in parallel or concurrently.

If there are very large queries, you really only want to run a few of those at a time, and let them get more of the memory they need. SQL Server determines a value that is calculated to be 4 X (the number of CPUs on the system). If possible, SQL Server will run that number of queries concurrently, giving them their minimum effective memory size. If there is still memory left over, some of them will be allowed to have their maximum effective memory. SQL Server tries to both maximize the amount of memory available to queries and also to keep many queries running on the system.

The ability to use the maximum effective memory is of great value in operations such as nightly batch cycles. You may be generating very large reports or performing rebuilds of indexes. These kinds of queries can use a lot of memory, and this mechanism will dynamically adjust to the need. So if there's not much waiting in the queues, the memory grant scheduler will give the queries the maximum requested memory quite often. If you're doing heavy processing during the day, you will run a lot more queries concurrently. These queries will get their minimum allocations so they'll run effectively, but the memory is more of a shared resource.

Once the scheduler says it's OK to allocate the memory for a request, the plan is "opened," which starts the actual execution running. From there, the plan will run to completion. If your query is using the default result set model, the plan will run until it has produced all of its rows, and they've all been sent back to the client. If you're using a cursor model, the processing is a little different. Each client request is then for just one block of rows, not all of them. After each set of results is sent back to the client, SQL Server must wait for the client to request the next set. While it is waiting, the entire plan is made *dormant*. This means that some of the locks are released, some resources are given up, and some positioning information is squirreled away. This information allows SQL Server to get back to where it was when the next set of rows is requested, and execution can continue.

Procedure Cache

We've mentioned the notion of SQL Server's procedure cache several times. It's important to understand that the procedure cache in SQL Server 7.0 is quite different than it was in previous versions. In earlier releases there were two effective configuration values for controlling the size of the procedure cache: one value specified a fixed size for SQL Server's total usable memory, and the second was a percentage of that memory (after fixed needs were satisfied) that was to be used exclusively for storing query plans. It was also true in previous versions that query plans for ad hoc SQL statements were never stored in the cache, only the plans for stored procedures. In SQL Server 7.0, the total size of memory is by default dynamic, and the space used for query plans is also very fluid.

One of the first things SQL Server 7.0 asks when processing a query is: Was it both ad hoc *and* cheap to compile? If it was, SQL Server won't cache it at all. It is cheaper to recompile these plans in the future than to push more expensive plans or data pages out of memory. If the query is not ad hoc or is not cheap to compile, SQL Server will get some memory from the buffer cache and save the plan in it. The memory comes from the buffer cache because that is the sole source of memory for 99 percent of the server's needs in SQL Server 7.0. There are a few special cases where SQL Server will allocate some large chunks of memory directly from the operating system, but those cases are extremely rare. For everything else, the management has been centralized.

Plans are saved in cache along with a cost factor that reflects the cost to actually create the plan by compiling the query. If it's an ad hoc plan, SQL Server will set its cost to zero, which means that it is immediately available to be kicked out of the procedure cache. For ad hoc SQL, although we're taking a chance that it might be reused, it's a low probability. If there's memory pressure on the system, we'd like plans for ad hoc statements to be the first to go. Thus, plans for ad hoc queries start out as eligible to be booted out of the cache. If the query is not ad hoc, SQL Server sets the cost to what it actually cost to compile the query. These costs are in units of disk I/O. When a data page is read off the disk, it has a cost of one I/O. When this plan was built, information was read off the disk, including statistics and the text of the query itself. SQL did additional processing and that processing work is normalized to the cost of an I/O. Now the cost of building a procedure can be weighed against the cost of doing an I/O. The cost figures highly in the ability to manage how much of the cache is actually going to stored procedures or any kind of query plans, compared to how much is going to disk caching of data. Once the cost is computed, the plan is put into the cache.

Figure 8 shows the flow of costing a plan and placing it into cache.

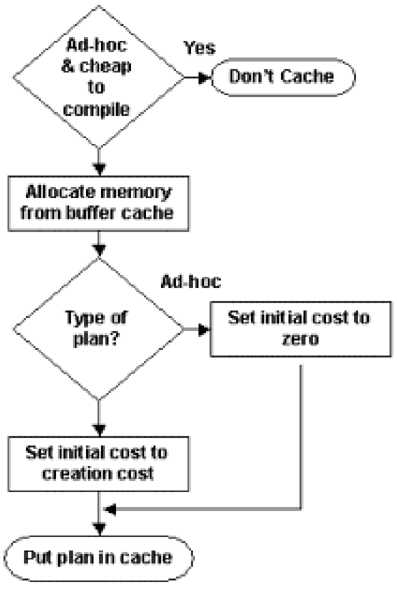
Ad-hoc & cheap to compile
Yes
Don't Cache
Allocate memory from buffer cache
Type of plan?
Ad-hoc
Set initial cost to zero
Set initial cost to creation cost
Put plan in cache

Figure 8. Inserting a plan into the cache

If another query comes along that can reuse that plan, SQL Server again determines what type of plan it is. If it's an ad hoc plan, SQL Server increments the cost by 1. So, if ad hoc plans are really being reused, they'll stay in the cache for a little while longer as their cost factor increases. If the plan is reused frequently, the cost will keep getting bumped up by one until it gets up to its actual creation cost. That is as high as the cost will ever be set. But it's reused a lot; if the same user or other users keep resubmitting the exact same SQL text, the plan will remain in cache.

If the query is not ad hoc, meaning that it's a stored procedure, a parameterized query, or an autoparameterized query, the cost is set back up to the creation cost every time the plan is reused. As long as the plan is reused, it will stay in cache. Even if it's not used for a while, depending on how expensive it was to create initially, it can remain in cache for a fair amount of time.

Figure 9 shows the flow of retrieving a plan from cache, and adjusting the cost.

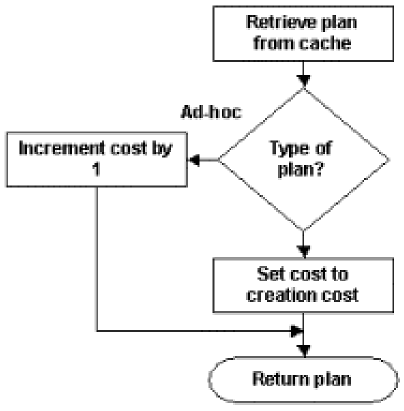

Figure 9. Retrieving a plan from the cache

The lazywriter is the mechanism that ages plans and is responsible for removing plans from cache when necessary. The lazywriter is actually part of the storage engine, but because it's so important to the query processing mechanism, we'll discuss it here. The lazywriter uses the same mechanism for managing memory used by query plans that it does for managing pages, because in SQL Server 7.0 plans are stored in the normal buffer cache. The lazywriter looks through all the buffer headers in the system. If there's very little memory pressure on the system, it looks very slowly; if memory pressure grows, the lazywriter starts running more frequently. As the lazywriter runs, it looks at a buffer header and looks at the current cost for the page in that buffer. If the cost is zero, that means it hasn't been touched since the lazywriter's last pass around, and the lazywriter will release that page to get some free memory in the system, to use for page I/O or for other plans. In addition, if the buffer contains a procedure plan, the lazywriter calls the SQL manager, which does some cleanup. Finally, the buffer is put on the freelist for reuse.

If the cost associated with a buffer is greater than zero, the lazywriter decrements the cost and continues inspecting other buffers. The cost is then actually indicating how many cycles of the lazywriter something will sit in the cache without reuse before it's thrown out. This algorithm, with the exception of the step calling SQL Manager if the object is a stored procedure, doesn't differentiate between plans in cache and data or index pages in cache. The lazywriter doesn't really know anything about an object being a procedure, and the algorithm balances nicely the use of a cache for disk I/O versus the use of cache for a procedure plan.

You will find that if you have something that is expensive to compile, even if it is not touched for a long time, it will sit in cache because its initial cost is so high. Anything that is reused frequently will sit in cache for a long time because its cost is also being reset whenever it's used, and the lazywriter will never see it go to zero.

Figure 10 shows the flow of the lazywriter's processing of the cache.

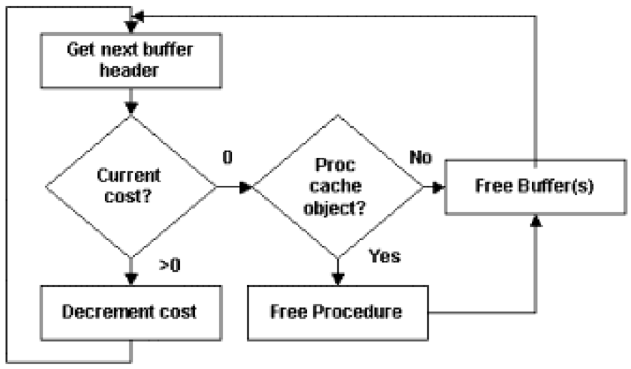

Figure 10. Lazywriter cache processing flow

Processing the Client's SQL

We'll now examine once more the processing that occurs when a SQL statement is submitted. First, we'll look at the case where an RPC event comes to the SQL Server from a client. Because SQL Server receives an RPC event, it knows that the event is some kind of parameterized SQL; it's either a prepare/execute model or EXECUTESQL. SQL Server will need to construct a cache key to identify this particular SQL Server text. If, instead, SQL Server is processing an actual stored procedure, it doesn't need to construct a key of its own; it can simply use the name of the procedure. For straight SQL text coming in through an RPC call, the cache key is constructed by hashing the SQL text. In addition, the key must reflect certain state information, such as some of the ANSI settings. A connection that has all the ANSI settings ON and another that has all the ANSI settings OFF, even if they pass in the same query, can't use the same plan. The processing will be different. For example, if one connection has set the value of the concat_null_yields_null to ON, it might produce completely different results from a connection that has set concat_null_yields_null to OFF, even if executing the exact same SQL text. So, SQL Server may need to maintain multiple copies of the plan in cache, one for each different combination of enabled ANSI settings. The enabled set of options is part of a key and that key is at the heart of looking things up using this caching mechanism, so SQL Server builds this key and uses it to do a lookup in the cache. If the plan isn't found in the cache, SQL Server will compile the plan, as described earlier, and put it in the cache with that key.

SQL Server also needs to determine if the command is a prepare operation, which means the plan should just be compiled and not executed. If it is a prepare operation, SQL Server returns a handle back to the client that the client will later use to retrieve that plan and execute it. If it's not a prepare operation, SQL Server will take that plan and execute it, just as if it was found in the original lookup in the cache.

The prepare/execute model adds a complicating factor to management of the cache. The prepare gives out a handle that can later be used to execute the plan. The application may keep this handle active for hours or days, periodically executing the plan. We can't invalidate the handle if we need to make room in the cache for a more active plan or data pages. What SQL Server actually does is put the plan in the cache and, in addition, it also saves away the SQL from the prepare operation in a more compact space. Now if there is memory pressure, the space used by the plan can be freed up in the manner described earlier, but there is still a copy of the SQL that was prepared. If the client comes through to execute the prepared SQL and there is no plan found in cache, SQL Server can retrieve the text, recompile it and put it back in the cache. In this way, the 16 kilobytes (KB) or more of pages in cache used to hold the plan can be reused, and the space held long term is perhaps 100 or 200 bytes of SQL code stored in another location.

The other case when processing a statement from a client is when the query comes in as a SQL language event. The flow is not that different, with one exception. In this case, SQL Server tries to use a technique called autoparameterization. The SQL text is matched up against an autoparameterization template. Autoparameterization is a difficult issue, so other database management products, which have been able to take advantage of shared SQL in the past, have generally not provided this option. The problem with it is if SQL Server were to automatically autoparameterize every query, some (or even most) of those queries would get very bad plans for some of the specific values that are subsequently submitted. In the case where the programmer puts a parameter marker in the code, the assumption is that the programmer knows the range of values to expect, and is willing to accept the plan that SQL Server comes up with. But when the programmer actually supplies a specific value, and SQL Server decides to treat that value as a changeable parameter, there is the possibility that any plan generated that works for one value may not work for subsequent values. With stored procedures, the programmer can force new plans to be generated by putting the WITH RECOMPILE option in the procedure. With autoparameterization, there is no way for the programmer to indicate that a new plan must be developed for each new value.

SQL Server is then very conservative when it comes to autoparameterization. There is a template of queries that are safe to be autoparameterized, and only queries that match the template will have autoparameterization applied. For example, suppose we have a query containing a WHERE clause with an equality operator and no joins, with a unique index on the column in the WHERE clause. SQL Server knows there will be never be more than one row returned and the plan should always use that unique index. SQL Server will never consider scanning, and the actual value will never change the plan in any way. This kind of query is safe for autoparameterization.

If the query matches the autoparameterization template, SQL Server actually replaces the literals with parameter markers, (for example, @p1, @p2) and that's what we send into the server, just as if it were an sp_executesql call. If the query was something SQL Server decided was not safe to autoparameterize, the client will send SQL Server the literal SQL text as ad hoc SQL.

Figure 11 shows the flow of processing when a client sends a request to SQL Server.

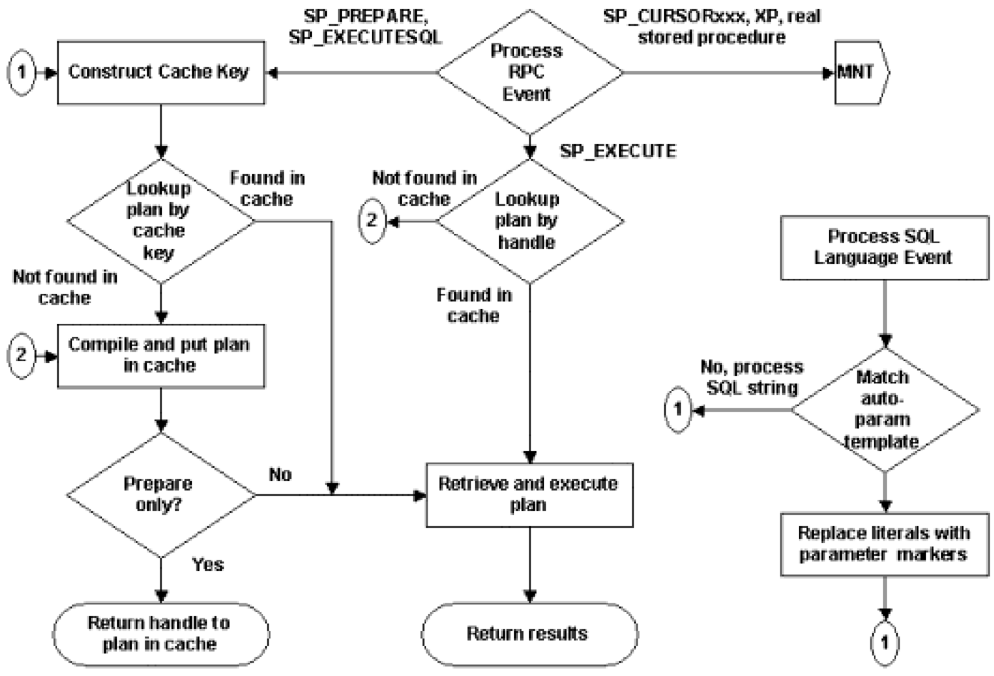
SP_PREPARE, SP_EXECUTESQL
SP_CURSORxxx, XP, real stored procedure
Process RPC Event
MNT
1
Construct Cache Key
SP_EXECUTE
Lookup plan by cache key
Found in cache
Not found in cache
2
Lookup plan by handle
Not found in cache
Found in cache
Process SQL Language Event
2
Compile and put plan in cache
No, process SQL string
1
Match auto-param template
Prepare only?
No
Yes
Retrieve and execute plan
Replace literals with parameter markers
Return handle to plan in cache
Return results
1

Figure 11. Processing the client's SQL

Compilation

We'll now look at compilation and optimization in a bit more detail. During the compilation process, SQL Server parses the statement and creates something called a sequence tree that is the internal representation of the statement. This is one of the few data structures that actually remain in SQL Server 7.0 from SQL Server 6.5. The sequence tree is then normalized. The main function of the Normalizer is to perform binding. Binding includes verifying that the tables and columns exist, and loading the metadata about the tables and columns. Information about required (implicit) conversions is also added to the sequence tree, for example, if the query is trying to add integer 10 to a numeric value, SQL Server will insert an implicit convert into the tree. Normalization also replaces references to a view with the definition of that view. Finally, normalization performs a few syntax-based optimizations. If the statement is a classic SQL statement, SQL Server extracts information from the sequence tree about that query and creates a special structure called a query graph, which is set up to let the optimizer work on it very effectively. The query graph is then optimized, and a plan is produced.

Figure 12 shows the flow of the compilation process.

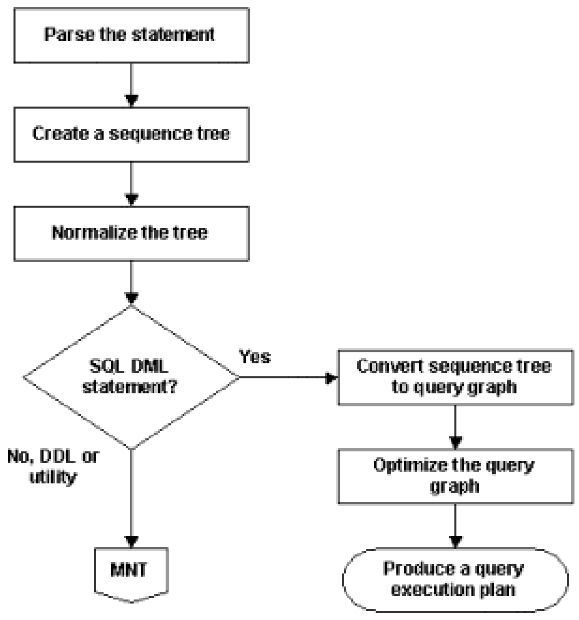

Figure 12. Compilation

Optimization

The SQL Server optimizer is really made up of the separate pieces. The first piece is a non-cost-based optimizer called trivial plan optimization. The whole idea of trivial plan optimization is that cost-based optimization is expensive to do when there really is only one viable plan for the SQL statement. A prime example is a query that consists of an INSERT statement with a VALUES clause. There is only one possible plan. Another example is a SELECT statement where all the columns are in a unique covering index and there is no other index that has that set of columns in it. These two examples are cases where SQL Server should simply generate the plan and not try to examine multiple plans to see if there is something better. The trivial plan optimizer finds the really obvious plans, which are typically very inexpensive. So the simplest queries tend to be weeded out early in the process and the optimizer doesn't spend a lot of time searching for a good plan. This is a good thing, because the number of potential plans in SQL Server 7.0 went up astronomically as SQL Server added hash joins, merge joins, and index intersections to its list of processing techniques.

If the trivial plan optimizer does not find a plan, SQL Server enters the next portion of optimization, known as simplification. Simplifications are syntactic transformations of the query itself, looking for commutative properties and operations that can be rearranged. SQL Server can do constant folding and other operations that don't require looking at the cost or analyzing what indexes are but can result in a more efficient query. SQL Server then loads up the statistics information on indexes and columns, and enters the final major part of optimization, which is the cost based optimizer.

Cost-based optimization has three phases. This first of these cost-based phases, called the transaction processing phase, finds plans for simple requests that are typical of transaction processing systems. These requests are generally more complex than those handled by the trivial plan optimizer, and require that plans be compared to find the one with the lowest cost. When the transaction processing phase completes, SQL Server compares the cost of the cheapest plan found to an internal threshold. The threshold is used to determine if further optimization is warranted. If the cost of the plan is less than the threshold, performing additional optimization would be more costly than just executing the plan that has already been found. So SQL Server will halt further optimization and use the plan found by the transaction processing phase.

If the plan found by the transaction processing phase is still expensive compared to that phases' threshold, SQL Server moved on to a second phase. This phase is sometimes called the QuickPlan phase. The QuickPlan phase expands the search for a good plan to cover choices that tend to be good for modestly complex queries. QuickPlan examines a range of possible plans, and when it is done compares the cost of the best plan to a second threshold. As with the transaction processing phase, if a plan has been found with a cost lower than the threshold, optimization is halted and that plan is used. Typically, if the plan that a query would have had in SQL Server 6.5 is also the optimal plan in SQL Server 7.0, the plan will tend to be found either by the trivial plan optimizer or by one of these first two phases of cost-based optimization. The rules were intentionally organized to try to make that true. The plan will probably consist of using a single index and using a nested loops join.

The last phase of optimization, called full optimization, is targeted at producing a good plan for complex to very complex queries. For complex queries, the plan produced by QuickPlan will often be considered much more expensive than the cost of continuing to search for a better plan, and full optimization will be performed. There are actually two separate options available in full optimization. If the best cost that the QuickPlan phase could come up with is greater than the configuration value for "cost threshold for parallelism," and if the server is a multiprocessor machine, the optimizer's final phase will involve looking for a plan that can run in parallel over multiple processors. If the best plan's cost from the QuickPlan phase is less than the configured "cost threshold for parallelism," the optimizer will only consider a serial plan. The full optimization phase can go through an exhaustive set of possibilities, and be quite time-consuming because a plan must be found during this last phase. The optimizer still might not check every conceivable plan, as it compares the cost of any potential plan against the cost to get to this point in the optimization, and it estimates how much it might cost to continue to try different optimizations. At some point, the optimizer might decide that using a current plan is cheaper than continuing to search for something better, and paying the additional compilation cost of continued optimization will not be cost effective. The kinds of queries processed during this final phase are typically queries whose plans will be used only once, so there is little chance that paying extra cost for compilation and optimization once will pay off when this plan is reused in subsequent executions. Those subsequent executions probably will not happen.

When a plan is found, it becomes the optimizer's output, and then SQL Server goes through all the caching mechanisms that we've already discussed, prior to executing the plan. You should be aware that if the full optimization phase came up with a parallel plan for this query, it doesn't necessarily mean the plan will be executed on multiple processors. If the machine is very busy, and cannot support running a single query on multiple CPUs, the plan will utilize a single processor.

Figure 13 shows the flow of processing through the optimizer.

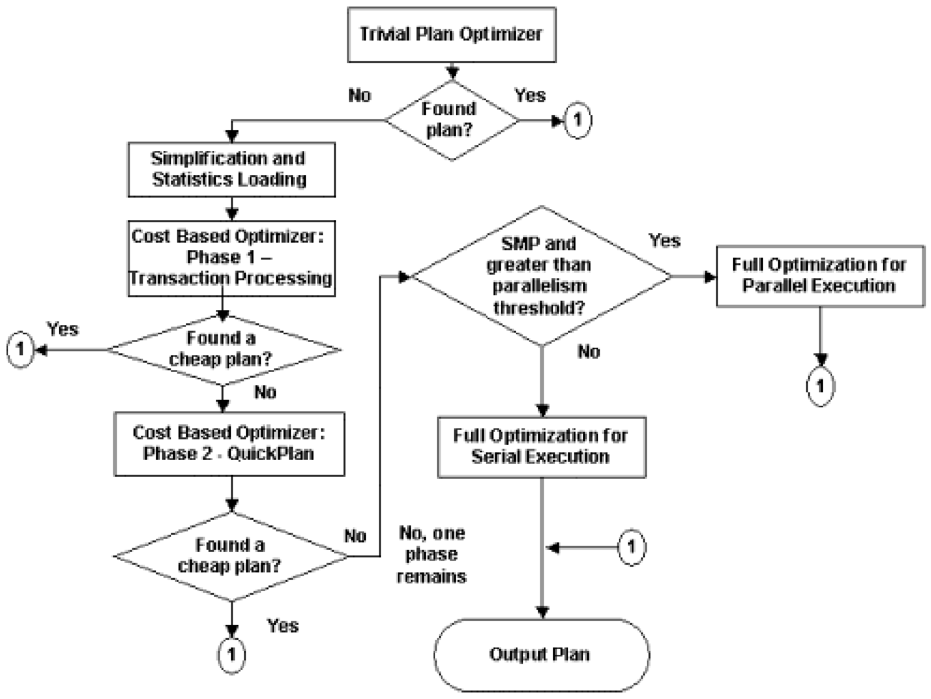
Trivial Plan Optimizer
No
Found plan?
Yes
1
Simplification and Statistics Loading
Cost Based Optimizer: Phase 1 – Transaction Processing
Yes
1
Found a cheap plan?
No
Cost Based Optimizer: Phase 2 - QuickPlan
Found a cheap plan?
No
Yes
1
SMP and greater than parallelism threshold?
Yes
Full Optimization for Parallel Execution
1
No
Full Optimization for Serial Execution
No, one phase remains
1
Output Plan

Figure 13. Optimization

Execution

The final step in query processing is execution. We're not going to talk about execution in any more detail other than this one short paragraph. The execution engine takes the plan that the optimizer has produced, and executes it. Along with the actual execution, the execution engine schedules the threads for the processes to run on, and provides interthread communication.

Summary

As mentioned, SQL Server internals and architecture make up a huge topic, and there is much more to it that we were able to present in this article. Our focus was directed toward a description of how SQL Server interacts with the client and how the SQL Server relational engine then processes those requests from the client. Hopefully, knowing how SQL Server processes your queries, and how and when it compiles them or recompiles them, will allow you to write better applications that take advantage of the power and sophistication of SQL Server 7.0.

© Microsoft Corporation. All rights reserved.

© 2014 Microsoft. All rights reserved.