ADDISON-WESLEY

T-SQL Enhancements in SQL Server 2005

Date: Jul 30, 2004 By Bob Beauchemin, Niels Berglund, Dan Sullivan. Sample Chapter is provided courtesy of Addison Wesley Professional.

SQL Server 2005 includes a plethora of new features and tools that can help developers more efficiently and effectively manage data. This article provides a preview of the new features and their use.

Sql server 2005 includes new Transact-SQL (T-SQL) functionality. The enhancements span the range from an alternative mechanism for transaction isolation to declarative support for hierarchical queries. And statement-level recompilation even improves existing T-SQL applications that were written before 2005.

Improvements to Transact-SQL

Microsoft has continually improved the Transact SQL language and the infrastructure of SQL Server itself. In brief, the improvements include the following:

SNAPSHOT isolation—Additional isolation level that does not use write locks

- Statement-level recompile—More efficient recompilation of stored procedures
- Event notifications—Integration of Data Definition Language (DDL) and DML operations with Service Broker
- Large data types—New data types that deprecate TEXT and IMAGE
- DDL triggers—Triggers that fire on DDL operations
- Common Table Expressions—Declarative syntax that makes a reusable expression part of a query
- Hierarchical queries—Declarative syntax for tree-based queries
- PIVOT—Declarative syntax aggregations across columns and converting columns to rows
- APPLY—New JOIN syntax made for use with user-defined functions and XML
- TOP—Row count based on an expression
- Transaction abort—TRY/CATCH syntax for handling errors

SNAPSHOT Isolation

SQL Server changes the state of a database by performing a transaction on it. Each transaction is a unit of work consisting of one or more steps. A "perfect" transaction is ACID, meaning it is atomic, consistent, isolated, and durable. In short, this means that the result of performing two transactions on a database, even if they are performed simultaneously by interleaving some of the steps that make them up, will not corrupt the database.

Atomic means that a transaction will perform all of its steps or fail and perform none of its steps. Consistent means that the transaction must not leave the results of a partial calculation in the database; for example, if a transaction is to move money from one account to another, it must not terminate after having subtracted money from one account but not having added it to another. Isolated means that none of the changes a transaction makes to a database become visible to other transactions until the transaction making the changes completes, and then they all appear simultaneously. Durable means that changes made to the database by a transaction that completes are permanent, typically by being written to a medium like a disk.

A transaction need not always be perfect. The isolation level of a transaction determines how close to perfect it is. Prior to SQL Server 2005, SQL Server provided four levels of isolation: READ UNCOMMITTED, REPEATABLE READ, READ COMMITTED, and SERIALIZABLE.

A SERIALIZABLE transaction is a perfect transaction. Functionally, a database could always use SERIALIZABLE—that is, perfect transactions, but doing so would typically adversely affect performance. Judicious use of isolation levels other than SERIALIZABLE, when analysis of an application shows that it does not require perfect transactions, will improve performance in these cases.

SQL Server uses the isolation level of a transaction to control concurrent access to data through a set of read and write locks. It applies these locks pessimistically; that is, they physically prevent any access to data that might compromise the required isolation level. In some cases, this will delay a transaction as it waits for a lock to be freed, or may even cause it to fail because of a timeout waiting for the lock.

SQL Server 2005 adds SNAPSHOT isolation that, in effect, provides alternate implementations of SERIALIZABLE and READ COMMITTED levels of isolation that use optimistic locking to control concurrent access rather than pessimistic locking. For some applications, SNAPSHOT isolation may provide better performance than pre–SQL Server 2005 implementations did. In addition, SNAPSHOT isolation makes it much easier to port database applications to SQL Server from database engines that make extensive use of SNAPSHOT isolation.

SQL Server 2005 has two kinds of SNAPSHOT isolation: transaction-level and statement level. Transaction-level SNAPSHOT isolation makes transactions perfect, the same as SERIALIZABLE does. Statement-level SNAPSHOT isolation makes transactions that have the same degree of isolation as READ COMMITTED does.

The transaction-level SNAPSHOT isolation optimistically assumes that if a transaction operates on an image of that database's committed data when the transaction started, the result will be the same as a transaction run at the SERIALIZABLE isolation level. Some time before the transaction completes, the optimistic assumption is tested, and if it proves not to be true, the transaction is rolled back.

Transaction-level SNAPSHOT isolation works by, in effect, making a version of the database by taking a snapshot of it when a transaction starts. Figure 7-1 shows this.

There are three transactions in Figure 7-1: transaction 1, transaction 2, and transaction 3. When transaction 1 starts, it is given a snapshot of the initial database. Transaction 2 starts before transaction 1 finishes, so it is also given a snapshot of the initial database. Transaction 3 starts after transaction 1 finishes but before transaction 2 does. Transaction 3 is given a snapshot of the initial database plus all the changes committed by transaction 1.

Figure 7-1: Snapshot Versioning

The result of using SERIALIZABLE or transaction-level SNAPSHOT isolation is the same; some transactions will fail and have to be retried, and may fail again, but the integrity of the database is always guaranteed.

Of course, SQL Server can't actually make a snapshot of the entire database, but it gets that effect by keeping track of each change to the database until all transactions that were started before the change was made are completed. This technique is called row versioning.

The row versioning model is built upon having multiple copies of the data. When reading data, the read happens against the copy, and no locks are held. When writing the data, the write happens against the "real" data, and it is protected with a write lock. For example, in a system implementing row versioning, user A starts a transaction and updates a column in a row. Before the transaction is committed, user B wants to read the same column in the same row. He is allowed to do the read but will read an older value. This is not the value that A is in the process of updating to, but the value A is updating from.

In statement-level SNAPSHOT isolation, the reader always reads the last committed value of a given row, just as READ COMMITTED does in a versioning database. Let's say we have a single-row table (called tab) with two columns: ID and name. Table 7-1 shows a versioning database at READ COMMITTED isolation.

Table 7-1: Versioning Database at READ COMMITTED Isolation

Step	User 1	User 2
1	BEGIN TRAN SELECT name FROM tab WHERE id = 1 **value is 'Name'	

2		BEGIN TRAN -UPDATE tab SET name = 'Newname' WHERE id = 1
3	SELECT name FROM tab WHERE id = 1 **value is 'Name'	
4		COMMIT
5	SELECT name FROM tab WHERE id = 1 **value is 'NewName'	
6	COMMIT	
7	SELECT name FROM tab WHERE id = 1 **value is 'NewName'	

The other transaction isolation level in a versioning database, SERIALIZABLE, is always implemented by the behavior that the reader always reads the row as of the beginning of the transaction, regardless of whether other users' changes are committed during the duration of the transaction or not. This was shown qualitatively in Figure 7-1. Table 7-2 shows a specific example of how two transactions interoperate when the SERIALIZABLE level of SNAPSHOT isolation is used.

The difference between this table and Table 7-1 occurs at step 5. Even though user 2 has updated a row and committed the update, user 1, using the SERIALIZABLE transaction isolation level, does not "see" the next value until user 1 commits his transaction. He sees the new value only in step 7. In SQL Server this is called "transaction-level SNAPSHOT isolation."

Table 7-2: Versioning Database at SERIALIZABLE Isolation

Step	**User 1**	**User 2**
1	BEGIN TRAN SELECT name FROM tab WHERE id = 1 **value is 'Name'	
2		BEGIN TRAN -UPDATE tab SET name = 'Newname' WHERE id = 1
3	SELECT name FROM tab WHERE id = 1 **value is 'Name'	
4		COMMIT
5	SELECT name FROM tab WHERE id = 1 **value is 'Name'	
6	COMMIT	
7	SELECT name FROM tab WHERE id = 1 **value is 'NewName'	

Both statement- and transaction-level SNAPSHOT isolation require that SNAPSHOT be enabled by using the SNAPSHOT isolation option of the ALTER DATABASE command. The following SQL batch does this for the pubs database.

```
ALTER DATABASE pubs
SET ALLOW_SNAPSHOT_ISOLATION ON
```

SNAPSHOT isolation can be turned on or off as needed.

Once SNAPSHOT isolation has been enabled, transaction-level isolation is used by specifically setting the transaction isolation level to SNAPSHOT. The following SQL batch does this.

```
ALTER DATABASE pubs
SET ALLOW_SNAPSHOT_ISOLATION ON
GO
USE pubs
GO
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRANS
— SQL Expressions
COMMIT TRANS
```

The SQL expression in the preceding batch will be executed, in effect, against a snapshot of the database that was taken when BEGIN TRANS was executed.

Statement-level SNAPSHOT isolation requires the use of an additional database option, READ_COMMITTED_SNAPSHOT. If this database option and

ALLOW_ SNAPSHOT_ISOLATION are ON, all transactions done at the READ UNCOMMITTED or READ COMMITTED levels will be executed as READ COMMITTED–level transactions using versioning instead of locking. Both transactions shown in the SQL batch that follows will be executed as READ COMMITTED using versioning.

```
– alter the database
ALTER DATABASE pubs
SET ALLOW_SNAPSHOT_ISOLATION ON
SET READ_COMMITTED_SNAPSHOT ON
GO
USE pubs
GO
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
BEGIN TRAN
– SQL expression will be executed as READ COMMITTED using versioning
END TRAN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
– SQL expression will be executed as READ COMMITTED using versioning
END TRAN
```

Whether ALLOW_SNAPSHOT_ISOLATION is ON or not can be checked for a particular database by the DATABASEPROPERTYEX command. This command returns the current database option or setting for a particular database. The setting to check is the SnapshotIsolationFramework setting, as in following code for the pubs database:

```
SELECT DATABASEPROPERTYEX ('pubs', 'SnapshotIsolationFramework')
```

Table 7-3: Example of SNAPSHOT Isolation

Step	User 1	User 2
1	SET TRANSACTION ISOLATION LEVEL SNAPSHOT BEGIN TRAN UPDATE snapTest SET col1 = 'NewNiels' WHERE id = 1	
2		-SET TRANSACTION ISOLATION LEVEL SNAPSHOT BEGIN TRAN SELECT col1 FROM snapTest WHERE id = 1 ** receives value 'Niels'
3	COMMIT TRAN	
4		SELECT col1 FROM snapTest WHERE id = 1 ** receives value 'Niels'
5		COMMIT TRAN
6		SELECT col1 FROM snapTest WHERE id = 1 -** receives value 'NewNiels'

As stated earlier, SQL Server does not actually make a copy of a database when a SNAPSHOT transaction is started. Whenever a record is updated, SQL Server stores in TEMPDB a copy (version) of the previously committed value and maintains these changes. All the versions of a record are marked with a timestamp of the transactions that made the change, and the versions are chained in TEMPDB using a linked list. The newest record value is stored in a database page and linked to the version store in TEMPDB. For read access in a SNAPSHOT isolation transaction, SQL Server first accesses from the data page the last committed record. It then retrieves the record value from the version store by traversing the chain of pointers to the specific record version of the data.

The code in Table 7-3 shows an example of how SNAPSHOT isolation works. The example uses a table, snapTest, looking like this.

```
–it is necessary to run
–SET ALLOW_SNAPSHOT_ISOLATION ON
–if that's not done already
CREATE TABLE snapTest ([id] INT IDENTITY,
          col1 VARCHAR(15))

–insert some data
INSERT INTO snapTest VALUES(1,'Niels')
```

The steps in Table 7-3 do the following:

1. We start a transaction under SNAPSHOT isolation and update one column in one row. This causes SQL Server to store a copy of the original value in TEMPDB. Notice that we do not commit or roll back at this stage, so locks are held. If we were to run sp_lock, we would see an exclusive lock on the primary key.
2. We start a new transaction under a new session and try to read from the same row that is being updated at the moment. This is the row with an exclusive lock. If this had been previous versions of SQL Server (running under at least READ COMMITTED), we would be locked out. However, running in SNAPSHOT mode, SQL Server looks in the version store in TEMPDB to retrieve the latest committed value and returns "Niels".
3. We commit the transaction, so the value is updated in the database and another version is put into the version store.
4. User 2 does a new SELECT (from within his original transaction) and will now receive the original value, "Niels".
5. User 2 finally commits the transaction.
6. User 2 does a new SELECT (after his transaction commits) and will now receive the new value, "NewNiels".

SNAPSHOT isolation is useful for converting an application written for a versioning database to SQL Server. When an application is developed for a versioning database, the developer does not need to be concerned with locking. Converting such an application to SQL Server may result in diminished performance because more locking is done than is required. Prior to SQL Server 2005, this sort of conversion may have required rewriting the application. In version 2005, in many cases the only thing that will have to be done is to enable SNAPSHOT isolation and READ_COMMITTED_SNAPSHOT.

SNAPSHOT isolation is also beneficial for applications that mostly read and do few updates. It is also interesting to note that when SQL Server 2005 is installed, versioning is enabled in the MASTER and MSDB databases by default.

Drawbacks of Versioning

Versioning has the capability to increase concurrency but does come with a few drawbacks of its own. Before you write new applications to use versioning, you should be aware of these drawbacks. You can then assess the value of locking against the convenience of versioning.

It can be costly because record versions need to be maintained even if no read operations are executing. This has the capability of filling up TEMPDB. If a database is set up for versioning, versions are kept in TEMPDB whether or not anyone is running a SNAPSHOT isolation–level transaction. Although a "garbage collector" algorithm will analyze the older versioning transaction and clean up TEMPDB eventually, you have no control over how often that cleanup in done. Plan the size of TEMPDB accordingly; it is used to keep versions for all databases with SNAPSHOT enabled. If you run out of space in TEMPDB, long-running transactions may fail.

In addition, reading data will sometimes cost more because of the need to traverse the version list. If you are doing versioning at the READ COMMITTED isolation level, the database may have to start at the beginning of the version list and read through it to attempt to read the last committed version.

There is also the possibility of update concurrency problems. Let's suppose that in Table 7-1 user 1 decides to update the row also. Table 7-4 shows how this would look.

Table 7-4: Versioning Database at SERIALIZABLE Isolation—Concurrent Updates

Step	User 1	User 2
1	BEGIN TRAN SELECT name FROM tab WHERE id = 1 **value is 'Name'	
2		BEGIN TRAN -UPDATE tab SET name = 'Newname' WHERE id = 1
3		COMMIT
4	UPDATE tab SET name = 'Another name' WHERE id = 1 ** produces concurrency violation	
5	ROLLBACK (and try update again?)	

In this scenario, user 1 reads the value "Name" and may base his update on that value. If user 2 commits his transaction before user 1 commits his, and user 1 tries to update, he bases his update on possibly bad data (the old value he read in step 1). Rather than allowing this to happen, versioning databases produce an error. The error message in this case is as follows:

```
Msg 3960, Level 16, State 1, Line 1. Cannot use snapshot isolation to access table 'tab' in database 'pubs'. Snapshot transaction aborted due to update conflict. Retry
```

Obviously, retrying transactions often enough will slow down the overall throughput of the application. In addition, the window of time for a concurrency violation to occur increases the longer a transaction reads old values. Because, at the SERIALIZABLE isolation level, the user always reads the old value until he commits the transaction, the window is much bigger—that is, concurrency violations are statistically much more likely to occur. In fact, vendors of versioning databases recommend against using SERIALIZABLE isolation (SQL Server ISOLATION LEVEL SNAPSHOT) in most cases. READ COMMITTED is a better choice with versioning.

Finally, as we said before, in versioning databases reads don't lock writes, which might be what we want. Is this possible with a versioning database? Locking-database programmers, when using versioning, tend to lock too little, introducing subtle concurrency problems. In a versioning database, there must be a way to do insist on a lock on read. Ordinarily this is done by doing a SQL SELECT FOR UPDATE. But SQL Server does not support SELECT FOR UPDATE with the appropriate semantic. There is, however, a solution. Even when READ_COMMITTED_SNAPSHOT is on, you can ensure a read lock by using SQL Server's REPEATABLE READ isolation level, which never does versioning. The SQL Server equivalent of ANSI's SELECT FOR UPDATE is SELECT with (REPEATABLEREAD). Note that this is different from the SQL Server UPDLOCK (update lock), which is a special lock that has similar semantics but only works if all participants in all transactions are using UPDLOCK. This is one place where programs written for versioning databases may have to change their code in porting to SQL Server 2005.

Monitoring Versioning

Allowing versioning to achieve concurrency is a major change. We've already seen how it can affect monitoring and capacity planning for TEMPDB. Therefore, all the tools and techniques that we've used in the past must be updated to account for this new concurrency style. Here are some of the enhancements that make this possible.

There are the following new T-SQL properties and metadata views:

- DATABASEPROPERTYEX—Tells us if SNAPSHOT is on
- sys.fn_top_version_generators()—Tables with most versions
- sys.fn_transaction_snapshot()—Transaction active when a SNAPSHOT transaction starts
- sys.fn_transactions()—Includes information about SNAPSHOT transaction (or not), if SNAPSHOT includes information about version chains and SNAPSHOT timestamps

There are new performance monitor counters for the following:

- Average version store data-generation rate (kilobytes per minute)
- Size of current version store (kilobytes)
- Free space in TEMPDB (kilobytes)
- Space used in the version store for each database (kilobytes)
- Longest running time in any SNAPSHOT transaction (seconds)

SNAPSHOT isolation information is also available during event tracing. Because a SNAPSHOT transaction has to be aware of any updates committed by other users, other users' updates appear in SQL Profiler while tracing a SNAPSHOT isolation transaction. Beware, since this can significantly increase the amount of data collected by Profiler.

Statement-Level Recompilation

The next thing we'll look at is a performance enhancement that is part of the infrastructural improvements in T-SQL: statement recompilation. In SQL Server 2000, the query plan architecture differs from previous versions, and it is divided into two structures: a compiled plan and an executable plan.

- Compiled plan (a.k.a. query plan)—A read-only data structure used by any number of users. The plan is reentrant, which implies that all users share the plan and no user context information (such as data variable values) is stored in the compiled plan. There are never more than one or two copies of the query plan in memory—one copy for all serial executions and another for all parallel executions.
- Executable plan—A data structure for each user that concurrently executes the query. This data structure, which is called the executable plan or execution context, holds the data specific to each user's execution, such as parameter values.

This architecture, paired with the fact that the execution context is reused, has very much improved the execution of not only stored procedures but

functions, batches, dynamic queries, and so on. However, there is a common problem with executing stored procedures, and that is recompilation. Examples of things that cause recompilation to occur are as follows:

- Schema changes
- Threshold changes in rows
- Certain SET options

Figure 7-2: Trace Properties Dialog for SQL Profiler

A recompilation can incur a huge cost especially if the procedure, function, or batch is large, because SQL Server 2000 does module-level recompilation. In other words, the whole procedure is recompiled even if the cause of the recompilation affects only a small portion of the procedure. In addition, if the recompilation happens because a SET option changes, the executable plan will be invalidated and not cached. The code in Listing 7-1 is extremely simple, but it can be used to illustrate the problem.

Figure 3 **Figure 7-3: Trace Output**

Listing 7-1 is a stored procedure which in the middle of the procedure changes the CONCAT_NULL_YIELDS_NULL option. When this runs against SQL Server 2000, a recompilation happens for each execution of the procedure.

Listing 7-1: Procedure That Causes Recompilation

```
CREATE PROCEDURE test2
AS

SELECT 'before set option'

—//change a set option
SET CONCAT_NULL_YIELDS_NULL OFF

SELECT 'after set option'
```

To verify that recompilation happens on SQL Server 2000, do the following:

1. Catalog the procedure in Listing 7-1.
2. Open the SQL Server Profiler and from the File menu, select New | Trace.
3. When the Trace Properties dialog comes up, choose the Events tab.
4. In the Stored Procedures event group, choose the SP:Recompile event, click the Add button, as shown in Figure 7-2, and then click Run.
5. Execute the procedure a couple of times from Query Analyzer and view the trace output.
6. The output from the trace will show a couple of entries in the Event Class column with the value of SP:Recompile, as in Figure 7-3. This indicates that the procedure has been recompiled.

Figure 7-4: Trace Properties Dialog in SQL Server 2005

As mentioned before, the cost of recompilation can be very high for large procedures, and in the SQL Server 2005 release, Microsoft has changed the model to statement-level re-compilation. At this stage you may worry that performance will suffer if each statement in a procedure is individually recompiled. Rest assured that the initial compilation is still on the module level, so only if a recompile is needed is it done per statement.

Another performance benefit in SQL Server 2005 is the fact that when statement recompilation is done, the execution context will not be invalidated. The procedure in Listing 7-1 can be used in SQL Server 2005 to compare the differences between SQL Server 2000 and 2005. In SQL Server 2005, follow the steps listed earlier and notice in the trace how a recompile happens only the first time; for each subsequent execution, there is no recompile. This is due to the fact that an execution plan will be created after the initial recompile. Run the following code after you have executed the procedure a couple of times, and notice that the result you get consists of both a compiled plan and an executable plan.

```
SELECT * FROM syscacheobjects
WHERE dbid = db_id('pubs')
AND objid = object_id('test2')
```

To be certain that you get the correct result, you can clean out the cache before you execute the procedure by executing dbcc freeproccache.

When setting up the trace, you will see how the SQL Profiler allows you to trace more events than in SQL Server 2000. Figure 7-4 shows the Events Selection tab from the Trace Properties dialog.

As mentioned in the beginning of this chapter, the statement-level recompilation can be seen as a purely infrastructural enhancement. As a developer or DBA, you will not explicitly use it even though you implicitly benefit from it, and it may change the way you develop stored procedures. No longer do recompiles have as much of a negative impact on performance.

DDL Triggers

A trigger is a block of SQL statements that are executed based on the fact that there has been an alteration (INSERT, UPDATE, or DELETE) to a table or on a view. In previous versions of SQL Server, the statements had to be written in T-SQL, but in version 2005, as we saw in Chapter 3, they can also be written using .NET languages. As we mentioned, the triggers are fired based on action statements (DML) in the database.

What about changes based on Data Definition Language statements, changes to the schema of a database or database server? It has not been possible to use triggers for that purpose—that is, until SQL Server 2005. In SQL Server 2005 you can create triggers for DDL statements as well as DML.

The syntax for creating a trigger for a DDL statement is shown in Listing 7-2, and as with a DML trigger, DDL triggers can be written using .NET languages as well.

Listing 7-2: Syntax for a DDL Trigger

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[ WITH ENCRYPTION ]
{ FOR | AFTER } { event_type [ ,...n ] | DDL_DATABASE_LEVEL_EVENTS }
  [ WITH APPEND ]
  [ NOT FOR REPLICATION ]
{ AS
  { sql_statement [ ...n ] | EXTERNAL NAME < method specifier > }
}
< method_specifier > ::=
  assembly_name:class_name[::method_name]
```

The syntax for a DML trigger is almost identical to that for a DDL trigger. There are, however, some differences.

- The ON clause in a DDL trigger refers to either the scope of the whole database server (ALL SERVER) or the current database (DATABASE).
- A DDL trigger cannot be an INSTEAD OF trigger.
- The event for which the trigger fires is defined in the event_type argument, which for several events is a comma-delimited list. Alternatively, you can use the blanket argument DDL_DATABASE_LEVEL_EVENTS.

The SQL Server Books Online has the full list of DDL statements, which can be used in the event_type argument and also by default are included in the DDL_DATABASE_LEVEL_EVENTS. A typical use of DDL triggers is for auditing and logging. The following code shows a simple example where we create a trigger that writes to a log table.

```
—first create a table to log to
CREATE TABLE ddlLog (id INT PRIMARY KEY IDENTITY,
  logTxt VARCHAR(MAX))
GO

—create our test table
CREATE TABLE triTest (id INT PRIMARY KEY)
GO

— create the trigger
CREATE TRIGGER ddlTri
ON DATABASE
```

```
AFTER DROP_TABLE
AS
INSERT INTO ddlLog VALUES('table dropped')
```

You may wonder what the VARCHAR(MAX) is all about in creating the first table—we'll cover that later in this chapter. The trigger is created with a scope of the local database (ON DATABASE), and it fires as soon as a table is dropped in that database (ON DROP_TABLE). Run following code to see the trigger in action.

```
DROP TABLE triTest
SELECT * FROM ddlLog
```

The DROP TABLE command fires the trigger and inserts one record in the ddlLog table, which is retrieved by the SELECT command.

As mentioned previously, DDL triggers can be very useful for logging and auditing. However, we do not get very much information from the trigger we just created. In DML triggers, we have the inserted and deleted tables, which allow us to get information about the data affected by the trigger. So, clearly, we need a way to get more information about events when a DDL trigger fires. The way to do that is through the eventdata function.

Eventdata

The eventdata() function returns information about what event fired a specific DDL trigger. The return value of the function is XML, and the XML is typed to a particular schema (XSD). Depending on the event type, the XSD includes different information. The following four items, however, are included for any event type:

- The time of the event
- The SPID of the connection that caused the trigger to fire
- The login name and user name of the user who executed the statement
- The type of the event

The additional information included in the result from eventdata is covered in SQL Server Books Online, so we will not go through each item here. However, for our trigger, which fires on the DROP TABLE command, the additional information items are as follows:

- Database
- Schema
- Object
- ObjectType
- TSQLCommand

In Listing 7-3 we change the trigger to insert the information from the eventdata function into the ddlLog table. Additionally, we change the trigger to fire on all DDL events.

Listing 7-3: Alter Trigger to Use eventdata

```
– alter the trigger
ALTER TRIGGER ddlTri
ON DATABASE
AFTER DDL_DATABASE_LEVEL_EVENTS
AS
INSERT INTO ddlLog VALUES CONVERT(VARCHAR(max)eventdata()
```

From the following code, we get the output in Listing 7-4.

```
–delete all entries in ddlLog
DELETE ddlLog

–create a new table
CREATE TABLE evtTest (id INT PRIMARY KEY)

–select the logTxt column with the XML
SELECT logTxt
FROM ddlLog
```

Listing 7-4: Output from eventdata

```
<EVENT_INSTANCE>
 <PostTime>2004-01-30T11:58:47.217</PostTime>
 <SPID>57</SPID>
 <EventType>CREATE_TABLE</EventType>
 <ServerName>ZMV44</ServerName>
 <LoginName>ZMV44\Administrator</LoginName>
 <UserName>ZMV44\Administrator</UserName>
 <DatabaseName>pubs</DatabaseName>
 <SchemaName>dbo</SchemaName>
 <ObjectName>foo</ObjectName>
 <ObjectType>TABLE</ObjectType>
 <TSQLCommand>
  <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON"
   ANSI_PADDING="ON" QUOTED_IDENTIFIER="ON"
   ENCRYPTED="FALSE" />
  <CommandText>
   CREATE TABLE evtTest (id int primary key)
  </CommandText>
 </TSQLCommand>
</EVENT_INSTANCE>
```

Because the data returned from the function is XML, we can use XQuery queries to retrieve specific item information. This can be done both in the trigger and from the table where we store the data. The following code illustrates how to retrieve information about the EventType, Object, and CommandText items in the eventdata information stored in the table ddlLog. Notice that we first store it into an XML data type variable, before we execute the XQuery statement against it.

```
DECLARE @data XML
SELECT @data = logTxt FROM ddlLog
WHERE id = 11

SELECT
CONVERT(NVARCHAR(100),
@data.query('data(//EventType)')) EventType,
CONVERT(NVARCHAR(100),
@data.query('data(//Object)')) Object,
CONVERT(NVARCHAR(100),
@data.query('data(//TSQLCommand/CommandText)')) Command
```

If the syntax in the previous code snippet seems strange, that's because it is XML and XQuery; read Chapters 8 and 9, where the XML data type and XQuery are covered in detail.

The programming model for both DML and DDL triggers is a synchronous model, which serves well when the processing that the trigger does is relatively short-running. This is necessary because DDL and DML triggers can be used to enforce rules and can roll back transactions if these rules are violated. If the trigger needs to do longer-running processing tasks, the scalability inevitably suffers. Bearing this in mind, we can see that for certain tasks, it would be beneficial to have an asynchronous event model. Therefore, in SQL Server 2005 Microsoft has included a new event notification model that works asynchronously: event notifications.

Event Notifications

Event notifications differ from triggers by the fact that the actual notification does not execute any code. Instead, information about the event is posted to a SQL Server Service Broker (SSB) service and is placed on a message queue from where it can be read by some other process.[1] Another difference between triggers and event notifications is that the event notifications execute in response to not only DDL and DML statements but also some trace events.

The syntax for creating an event notification is as follows.

```
CREATE EVENT NOTIFICATION event_notification_name
ON { SERVER | DATABASE |
[ ENABLED | DISABLED ]
{ FOR { event_type |
   DDL_DATABASE_LEVEL_EVENTS } [ ,...n ]
TO broker_service
```

The syntax looks a little like the syntax for creating a DDL trigger, and the arguments are as follows.

- event_notification_name—This is the name of the event notification.
- SERVER—The scope of the event notification is the current server.
- DATABASE—The scope of the event notification is the current database.

- ENABLED—This specifies that the event notification is active when the CREATE statement has executed.
- DISABLED—This specifies that the event notification is inactive until the notification is activated by executing an ALTER EVENT NOTIFICATION statement.
- event_type—This is the name of an event that, after execution, causes the event notification to execute. SQL Server Books Online has the full list of events included in event_type.
- DDL_DATABASE_LEVEL_EVENTS—The event notification fires after any of the CREATE, ALTER, or DROP statements that can be indicated in event_type execute.
- broker_service—This is the SSB service to which SQL Server posts the data about an event.

The event notification contains the same information received from the eventdata function mentioned previously. When the event notification fires, the notification mechanism executes the eventdata function and posts the information to the Service Broker. For an event notification to be created, an existing SQL Server Service Broker instance needs to be located either locally or remotely. The steps to create the SQL Server Service Broker are shown in Listing 7-5. Chapter 15 covers SSB in detail and also covers how to create queues, services, and so on.

Listing 7-5: Steps to Create a Service Broker Instance

```
–first we need a queue
CREATE QUEUE queue evtDdlNotif
WITH STATUS = ON

–then we can create the service
CREATE SERVICE evtDdlService
ON QUEUE evtDdlNotif
–this is a MS supplied contract
–which uses an existing message type
–{http://schemas.microsoft.com/SQL/Notifications}EventNotification
(http://schemas.microsoft.com/SQL/Notifications/PostEventNotification)
```

First, the message queue that will hold the eventdata information is created. Typically, another process listens for incoming messages on this queue, or another process will kick off when a message arrives. A service is then built on the queue. When a SQL Server Service Broker service is created, there needs to be a contract to indicate what types of messages this service understands. In a SQL Server Service Broker application, the developer usually defines message types and contracts based on the application's requirements. For event notifications, however, Microsoft has a predefined message type, {http://schemas.microsoft.com/SQL/Notifications}EventNotification, and a contract, http://schemas.microsoft.com/SQL/Notifications/PostEventNotification.

The following code shows how to create an event notification for DDL events scoped to the local database, sending the notifications to the evt DdlService.

```
CREATE EVENT NOTIFICATION ddlEvents
ON DATABASE
FOR DDL_DATABASE_LEVEL_EVENTS
TO SERVICE evtDdlService
```

With both the event notification and the service in place, a new process can now be started in SQL Server Management Studio, using the WAITFOR and RECEIVE statements (more about this in Chapter 15) as in the following code.

```
WAITFOR(
RECEIVE * FROM evtDdlNotif
)
```

You can now execute a DDL statement, and then switch to the process with the WAITFOR statement and view the result. Running CREATE TABLE evtNotifTbl (id INT) shows in the WAITFOR process a two-row resultset, where one of the rows has a message_type_id of 20. This is the {http://schemas.microsoft.com/SQL/Notifications}EventNotification message type. The eventdata information is stored as a binary value in the message_body column. To see the actual data, we need to change the WAITFOR statement a little bit.

```
DECLARE @msgtypeid INT
DECLARE @msg VARBINARY(MAX)

WAITFOR(
RECEIVE TOP(1)
@msgtypeid = message_type_id,
@msg = message_body
FROM evtDdlNotif
)
–check if this is the correct message type
IF @msgtypeid = 20
BEGIN
–do something useful WITH the message
–here we just select it as a result
SELECT CONVERT(NVARCHAR(MAX), @msg)
END
```

Running this code against the CREATE TABLE statement shown earlier produces the same output as in Listing 7-4. An additional benefit with event notifications is that they can be used for both system level and trace events in addition to DDL events. The following code shows how to create an event notification for SQL Server logins.

```
CREATE EVENT NOTIFICATION loginEvents ON SERVER
FOR audit_login TO SERVICE evtLoginService
```

For system-level event notifications, the ON SERVER keyword needs to be explicitly specified; it cannot be used at the database level. Listing 7-6 shows the eventdata information received after executing a login.

Listing 7-6: eventdata Output from Login

```
<EVENT_INSTANCE>
 <PostTime>2003-06-29T09:46:23.623</PostTime>
 <SPID>51</SPID>
 <EventType>AUDIT_LOGIN</EventType>
 <ServerName>ZMV44</ServerName>
 <LoginName>ZMV44\Administrator</LoginName>
 <UserName>ZMV44\Administrator</UserName>
 <Database>eventstest</Database>
 <!– additional information elided –>
</EVENT_INSTANCE>
```

You may wonder what happens if the transaction that caused the notification is rolled back. In that case, the posting of the notification is rolled back as well. If for some reason the delivery of a notification fails, the original transaction is not affected.

Some of the previous code examples have used VARCHAR(MAX) as the data type for a column. Let's look at what that is all about.

Large Value Data Types

In SQL Server 2000 (and 7) the maximum size for VARCHAR and VARBINARY was 8,000 and for NVARCHAR 4,000. If you had data that potentially exceeded that size, you needed to use the TEXT, NTEXT, or IMAGE data types (known as Large Object data types, or LOBs). This was always a hassle because they were hard to work with, in both retrieval and action statements.

This situation changes in SQL Server 2005 with the introduction of the MAX specifier. This specifier allows storage of up to 2^{31} bytes of data, and for Unicode it is 2^{30}
bytes. When you use the VARCHAR(MAX) or NVARCHAR(MAX) data type, the data is stored as character strings, whereas for VARBINARY(MAX) it is stored as bytes. These three data types are commonly known as Large Value data types. The following code shows the use of these data types in action.

```
CREATE TABLE largeValues (
 lVarchar VARCHAR(MAX),
 lnVarchar NVARCHAR(MAX),
 lVarbinary VARBINARY(MAX)
)
```

We mentioned earlier that LOBs are hard to work with. Additionally, they cannot, for example, be used as variables in a procedure or a function. The Large Value data types do not have these restrictions, as we can see in the following code snippet, which shows a Large Value data type being a parameter in a function. It also shows how the data type can be concatenated.

```
CREATE FUNCTION dovmax(@in VARCHAR(MAX))
RETURNS VARCHAR(MAX)
AS
BEGIN
– supports concatenation
RETURN @in + '12345'
END
```

SQL Server's string handling functions can be used on VARCHAR(MAX) and NVARCHAR(MAX) columns. So instead of having to read in the whole amount of data, SUBSTRING can be used. By storing the data as character strings (or bytes), the Large Value data types are similar in behavior to their smaller counterparts VARCHAR, NVARCHAR, and VARBINARY, and offer a consistent programming model. Using the Large Value data types instead of LOBs is recommended; in fact, the LOBs are being deprecated.

When we first came across the enhanced size of the VARCHAR data type in SQL Server 7 (from 256 to 8,000), we thought, "Great, we can now have a

table with several VARCHAR columns with the size of 8,000 instead of a text column." You probably know that this doesn't work, because in SQL Server 7 and 2000, you cannot have a row exceeding 8,060 bytes, the size of a page. In SQL Server 2005 this has changed as well, and a row can now span several pages.

T-SQL Language Enhancements

Even though this book is much about the CLR and outside access to SQL Server, let's not forget that Microsoft has enhanced the T-SQL language a lot in SQL Server 2005. In this section, we will look at some of the improvements.

TOP

TOP was introduced in SQL Server 7. Until SQL Server 2005, the TOP clause allowed the user to specify the number or percent of rows to be returned in a SELECT statement. In SQL Server 2005, the TOP clause can be used also for INSERT, UPDATE, and DELETE (in addition to SELECT), and the syntax is as follows: TOP (expression) [PERCENT]. Notice the parentheses around the expression; this is required when TOP is used for UPDATE, INSERT, and DELETE.

The following code shows some examples of using TOP.

```
–create a table and insert some data
CREATE TABLE toptest (col1 VARCHAR(150))
INSERT INTO toptest VALUES('Niels1')
INSERT INTO toptest VALUES('Niels2')
INSERT INTO toptest VALUES('Niels3')
INSERT INTO toptest VALUES('Niels4')
INSERT INTO toptest VALUES('Niels5')

–this returns 'Niels1' and 'Niels2'
SELECT TOP(2) * FROM toptest

–this sets 'Niels1' and 'Niels2' to 'hi'
UPDATE TOP(2) toptest SET col1 = 'hi'
SELECT * FROM toptest

–the two rows with 'hi' are deleted
DELETE TOP(2) toptest
SELECT * FROM toptest

–create a new table and insert some data
CREATE TABLE toptest2 (col1 VARCHAR(150))
INSERT INTO toptest2 VALUES('Niels1')
INSERT INTO toptest2 VALUES('Niels2')
INSERT INTO toptest2 VALUES('Niels3')
INSERT INTO toptest2 VALUES('Niels4')
INSERT INTO toptest2 VALUES('Niels5')

–'Niels1' and 'Niels2' are inserted
INSERT top(2) toptest
SELECT * FROM toptest2

SELECT * FROM toptest
```

An additional difference between the TOP clause in previous versions of SQL Server and in SQL Server 2005 is that we now can use expressions for number definition. The following code shows a couple of examples of that (it uses the tables from the preceding example).

```
–declare 3 variables
DECLARE @a INT
DECLARE @b INT
DECLARE @c INT

–set values
SET @a = 10
SET @b = 5
SELECT @c = @a/@b

–use the calculated expression
SELECT TOP(@c)* FROM toptest

–insert some more data in toptest
INSERT INTO toptest VALUES('Niels6')
INSERT INTO toptest VALUES('Niels7')
INSERT INTO toptest VALUES('Niels8')


–use a SELECT statement as expression
–this should return 5 rows
SELECT TOP(SELECT COUNT(*) FROM toptest2) *
FROM toptest
```

The next T-SQL enhancement we'll look at is something completely new in SQL Server: the OUTPUT clause.

OUTPUT

The execution of a DML statement such as INSERT, UPDATE, or DELETE does not produce any results that indicate what was changed. Prior to SQL Server 2005, an extra round trip to the database was required to determine the changes. In SQL Server 2005 the INSERT, UPDATE, and DELETE statements have been enhanced to support an OUTPUT clause so that a single round trip is all that is required to modify the database and determine what changed. You use the OUTPUT clause together with the inserted and deleted virtual tables, much as in a trigger. The OUTPUT clause must be used with an INTO expression to fill a table. Typically, this will be a table variable. The following example creates a table, inserts some data, and finally deletes some records.

```
–create table and insert data
CREATE TABLE outputtbl
(id INT IDENTITY, col1 VARCHAR(15))
go

INSERT INTO outputtbl VALUES('row1')
INSERT INTO outputtbl VALUES ('row2')
INSERT INTO outputtbl VALUES ('row5')
INSERT INTO outputtbl VALUES ('row6')
INSERT INTO outputtbl VALUES ('row7')
INSERT INTO outputtbl VALUES ('row8')
INSERT INTO outputtbl VALUES ('row9')
INSERT INTO outputtbl VALUES ('row10')

– make a table variable to hold the results of the OUTPUT clause
DECLARE @del AS TABLE (deletedId INT, deletedValue VARCHAR(15))
–delete two rows and return through
–the output clause
DELETE outputtbl
OUTPUT DELETED.id, DELETED.col1 INTO @del
WHERE id < 3
SELECT * FROM @del
GO
deletedId  deletedValue
———— ————————-
1      row1
2      row2

(2 row(s) affected)
```

The previous example inserted the id and col1 values of the rows that were deleted into the table variable @del.

When used with an UPDATE command, OUTPUT produces both a DELETED and an INSERTED table. The DELETED table contains the values before the UPDATE command, and the DELETED table has the values after the UPDATE command. An example follows that shows OUTPUT being used to capture the result of an UPDATE.

```
–update records, this populates
–both the inserted and deleted tables
DECLARE @changes TABLE
(id INT, oldValue VARCHAR(15), newValue VARCHAR(15))
UPDATE outputtbl
SET col1 = 'updated'
OUTPUT inserted.id, deleted.col1, inserted.col1
INTO @changes
WHERE id < 5
SELECT * FROM @changes
GO
id     oldValue    newValue
———— ————————- ————————-
3      row5     updated
4      row6     updated

(2 row(s) affected)
```

Common Table Expressions and Recursive Queries

A Common Table Expression, or CTE, is an expression that produces a table that is referred to by name within the context of a single query. The general syntax for a CTE follows.

```
[WITH <common_table_expression> [,...n] ]
<common_table_expression>::=
```

```
  expression_name
  [(column_name [,...n])]
  AS
  (<CTE_query_expression>)
```

The following SQL batch shows a trivial usage of a CTE just to give you a feeling for its syntax.

```
WITH MathConst(PI, Avogadro)
AS
(SELECT 3.14159, 6.022e23)
SELECT * FROM MathConst
GO
PI              Avogadro
--------------- ------------
3.14159           6.022E+23
(1 row(s) affected)
```

The WITH clause, in effect, defines a table and its columns. This example says that a table named MathConst has two columns named PI and Avogadro. This is followed by a SELECT statement enclosed in parentheses after an AS keyword. And finally, all this is followed by a SELECT statement that references the MathConst table. Note that the syntax of the WITH clause is very similar to that of a VIEW. One way to think of a CTE is as a VIEW that lasts only for the life of the query expression at the end of the CTE. In the example, MathConst acts like a VIEW that is referenced in the query expression at the end of the CTE.

It is possible to define multiple tables in a CTE. A SQL batch follows that shows another trivial usage of a CTE that defines two tables, again shown just to make the syntax clear.

```
WITH MathConst(PI, Avogadro)
AS
(SELECT 3.14159, 6.022e23),
— second table
Package(Length, Width)
AS (SELECT 2, 5)
SELECT * FROM MathConst, Package
PI              Avogadro     Length   Width
--------------- ------------ ------ --

3.14159           6.022E+23       2      5

(1 row(s) affected)
```

In this example, the CTE produced two tables, and the query expression merely joined them.

Both of the previous examples could have been done without using CTEs and, in fact, would have been easier to do without them. So what good are they?

In once sense, a CTE is just an alternate syntax for creating a VIEW that exists for one SQL expression, or it can be thought of as a more convenient way to use a derived table—that is, a subquery. However, CTEs are part of the SQL-92 standard, so adding them to SQL Server increases its standards compliance. In addition, CTEs are implemented in other databases, so ports from those databases may be easier with the addition of CTEs.

In some cases, CTEs can save a significant amount of typing and may provide extra information that can be used when the query plan is optimized. Let's look at an example where this is the case.

For this example, we will use three tables from the AdventureWorks database, a sample database that is distributed with SQL Server. We will use the SalesPerson, SalesHeader, and SalesDetail tables. The Sales Person table lists each salesperson that works for AdventureWorks. For each sale made at AdventureWorks, a SalesHeader is entered along with a SalesDetail for each item that that was sold in that sale. Each Sales Header lists the ID of the salesperson who made the sale. Each Sales Detail entry lists a part number, its unit price, and the quantity of the part sold.

The stock room has just called the Big Boss and told him that they are out of part number 90. The Big Boss calls you and wants you to make a report that lists the ID of each salesperson. Along with the ID, the Big Boss wants the text "MakeCall" listed if a salesperson made a sale that depends on part number 90 to be complete. Otherwise, he wants the text "Relax" printed. Just to ensure that the report lights a fire under the salespeople, the Big Boss also wants each line to list the value of the sale and the salesperson's sales quota.

Before we actually make use of the CTE, let's first write a query that finds all the IDs of salespeople who have sales that depend on part number 90.

```
SELECT DISTINCT SH.SalesPersonId FROM SalesOrderHeader SH JOIN
SalesOrderDetail SD ON SH.SalesOrderId = SD.SalesOrderId
AND SD.ProductID = 90
SalesPersonId
GO
SalesPersonId
-------------
14
21
22
more rows
(14 row(s) affected)
```

But the Big Boss has asked for a report with lines that look like this.

```
Action  SalesPersonID SalesQuota           Value
------ ------------- ----------- ----------
MakeCall 22      250000.0000          2332.7784
... more lines
Relax  35      250000.0000          0
```

Each line number has the ID of a salesperson. If that salesperson has an order that depends on part number 90, the first column says "MakeCall" and the last column has the value involved in the order. Otherwise, the first column says "Relax" and the last column has 0 in it.

Figure 7-5: A Chart of Accounts

Without CTEs, we could use a subquery to find the salespeople with orders that depend on the missing part to make the report the Big Boss wants, as in the SQL batch that follows.

```
SELECT 'MakeCall' AS Action, S.SalesPersonID, S.SalesQuota,
(SELECT SUM(SD.UnitPrice * SD.OrderQty) FROM SalesOrderHeader SH
JOIN SalesOrderDetail SD ON
SH.SalesOrderId = SD.SalesOrderId
AND SD.ProductID=90 AND SH.SalesPersonID=S.SalesPersonID
)
FROM SalesPerson S
WHERE EXISTS
(
SELECT * FROM SalesOrderHeader SH JOIN SalesOrderDetail SD ON
SH.SalesOrderID = SD.SalesOrderID AND SD.ProductID = 90
AND SH.SalesPersonID = S.SalesPersonID
)
UNION
SELECT 'Relax' AS Action, S.SalesPersonID, S.SalesQuota, 0
FROM SalesPerson S
WHERE NOT EXISTS
(
SELECT * FROM SalesOrderHeader SH JOIN SalesOrderDetail SD ON
SH.SalesOrderID = SD.SalesOrderID AND SD.ProductID = 90
AND SH.SalesPersonID = S.SalesPersonID
)
```

Notice that the subquery is reused in a number of places—once in the calculation of the value of the sales involved in the missing part and then again, twice more, in finding the salespeople involved in sales with and without the missing part.

Now let's produce the same report using a CTE.

```
WITH Missing(SP, AMT)
AS(
SELECT SH.SalesPersonID, SUM(SD.UnitPrice * SD.OrderQty) FROM SalesOrderHeader SH
JOIN SalesOrderDetail SD ON SH.SalesOrderId = SD.SalesOrderId
AND SD.ProductID=90 GROUP BY SH.SalesPersonID
)
SELECT 'MakeCall' AS Action, S.SalesPersonID, S.SalesQuota, Missing.AMT
FROM Missing JOIN SalesPerson S ON Missing.SP = S.SalesPersonID
UNION
SELECT 'Relax' AS Action, S.SalesPersonID, S.SalesQuota, 0
FROM SalesPerson S WHERE S.SalesPersonID NOT IN (SELECT SP FROM  Missing)
```

The Missing CTE is a table that has a row for each salesperson who has an order that depends on the missing part, and the value of what is missing. Notice that the Missing table is used in one part of the query to find the value of the missing parts and in another to determine whether a sales person should "MakeCall" or "Relax".

Although your opinion may differ, the CTE syntax is a bit clear and more encapsulated; that is, there is only one place that defines what orders are missing part number 90. Also, in theory, the CTE is giving the optimizer a bit more information in that it is telling the optimizer it plans on using Missing more than once.

The CTE is also part of another feature of SQL Server 2005 that is also part of the SQL:1999 standard. It is called a recursive query. This is especially useful for a chart of accounts in an accounting system or a parts explosion in a bill of materials. Both of these involve tree-structured data. In general, a

recursive query is useful anytime tree-structured data is involved. We will look at an example of a chart of accounts to see how recursive queries work.

Figure 7-5
shows a simple chart of accounts containing two kinds of accounts: detail accounts and rollup accounts. Detail accounts have an actual balance associated with them; when a posting is made to an accounting system, it is posted to detail accounts. In Figure 7-5, account 4001 is a detail account that has a balance of $12.

Rollup accounts are used to summarize the totals of other accounts, which may be detail accounts or other rollup accounts. Every account, except for the root account, has a parent. The total of a rollup account is the sum of the accounts that are its children. In Figure 7-5 account 3002 is a rollup account, and it represents the sum of its two children, accounts 4001 and 4002.

In practice, one of the ways to represent a chart of accounts is to have two tables: one for detail accounts and the other for rollup accounts. A detail account has an account number, a parent account number, and a balance for columns. A rollup account has an account number and a parent but no balance associated with it. The SQL batch that follows builds and populates these two tables for the accounts shown in Figure 7-5.

```
CREATE TABLE DetailAccount(id INT PRIMARY KEY,
parent INT, balance FLOAT)
CREATE TABLE RollupAccount(id INT PRIMARY KEY,
parent INT)
INSERT INTO DetailAccount VALUES (3001, 2001, 10)
INSERT INTO DetailAccount VALUES(4001, 3002, 12)
INSERT INTO DetailAccount VALUES(4002, 3002, 14)
INSERT INTO DetailAccount VALUES(3004, 2002, 17)
INSERT INTO DetailAccount VALUES(3005, 2002, 10)
INSERT INTO DetailAccount VALUES(3006, 2002, 25)
INSERT INTO DetailAccount VALUES(3007, 2003, 7)
INSERT INTO DetailAccount VALUES(3008, 2003, 9)

INSERT INTO RollupAccount VALUES(3002, 2001)
INSERT INTO RollupAccount VALUES(2001, 1000)
INSERT INTO RollupAccount VALUES(2002, 1000)
INSERT INTO RollupAccount VALUES(2003, 1000)
INSERT INTO RollupAccount VALUES(1000, 0)
```

Figure 7-6: Recursive Query

Note that this example does not include any referential integrity constraints or other information to make it easier to follow.

A typical thing to do with a chart of accounts it to calculate the value of all the rollup accounts or, in some cases, the value of a particular rollup account. In Figure 7-5
(shown earlier) the value of the rollup accounts is shown in gray, next to the account itself. We would like to be able to write a SQL batch like the one that follows.

```
SELECT id, balance FROM Rollup — a handy view
id     balance
------ -----------
1000    104
2001    36
2002    52
2003    16
3001    10
3002    26
3004    17
3005    10
3006    25
3007    7
3008    9
4001    12
4002    14

(13 row(s) affected)

SELECT id, balance FROM Rollup WHERE id = 2001
id     balance
------ -----------
2001    36

(1 row(s) affected)
```

This query shows a view name, Rollup, that we can query to find the values of all the accounts in the chart of accounts or an individual account. Let's look at how we can do this.

To start with, we will make a recursive query that just lists all the account numbers, starting with the top rollup account, 1000. The query that follows does this.

```
WITH Rollup(id, parent)
AS
(
 — anchor
 SELECT id, parent FROM RollupAccount WHERE id = 1000
 UNION ALL
— recursive call
SELECT R1.id, R1.parent FROM
(
  SELECT id, parent FROM DetailAccount
  UNION ALL
  SELECT id, parent FROM RollupAccount
) R1
JOIN Rollup R2 ON R2.id = r1.parent
)
— selecting results
SELECT id, parent FROM Rollup
GO
id     parent
------ ------
1000    0
2001    1000
2002    1000
2003    1000
3007    2003
3008    2003
3004    2002
3005    2002
3006    2002
3001    2001
3002    2001
4001    3002
4002    3002

(13 row(s) affected)
```

The previous batch creates a CTE named Rollup. There are three parts to a CTE when it is used to do recursion. The anchor, which initializes the recursion, is first. It sets the initial values of Rollup. In this case, Rollup is initialized to a table that has a single row representing the rollup account with id = 1000. The anchor may not make reference to the CTE Rollup.

The recursive call follows a UNION ALL keyword. UNION ALL must be used in this case. It makes reference to the CTE Rollup. The recursive call will be executed repeatedly until it produces no results. Each time it is called, Rollup will be the results of the previous call. Figure 7-6 shows the results of the anchor and each recursive call.

First the anchor is run, and it produces a result set that includes only the account 1000. Next the recursive call is run and produces a resultset that consists of all the accounts that have as a parent account 1000. The recursive call runs repeatedly, each time joined with its own previous result to produce the children of the accounts selected in the previous recursion. Also note that the recursive call itself is a UNION ALL because the accounts are spread out between the DetailAccount table and the RollupAccount table.

After the body of the CTE, the SELECT statement just selects all the results in Rollup—that is, the UNION of all the results produced by calls in the CTE body.

Now that we can produce a list of all the accounts by walking through the hierarchy from top to bottom, we can use what we learned to calculate the value of each account.

To calculate the values of the accounts, we must work from the bottom up—that is from the detail accounts up to the rollup account 1000. This means that our anchor must select all the detail accounts, and the recursive calls must progressively walk up the hierarchy to account 1000. Note that there is no requirement that the anchor produce a single row; it is just a SELECT statement.

The query that follows produces the values of all the accounts, both detail and rollup.

```
WITH Rollup(id, parent, balance)
AS
(
— anchor
SELECT id, parent, balance FROM DetailAccount
UNION ALL
— recursive call
SELECT R1.id, R1.parent, R2.balance
FROM RollupAccount R1
JOIN Rollup R2 ON R1.id = R2.parent
```

```
)
SELECT id, SUM(balance) balance FROM Rollup GROUP BY id
GO
id     balance
------ -----------
1000    104
2001    36
2002    52
2003    16
3001    10
3002    26
3004    17
3005    10
3006    25
3007    7
3008    9
4001    12
4002    14

(13 row(s) affected)
```

This query starts by having the anchor select all the detail accounts. The recursive call selects all the accounts that are parents, along with any balance produced by the previous call. This results in a table in which accounts are listed more than once. In fact, the table has as many rows for an account as that account has descendant accounts that are detail accounts. For example, if you looked at the rows produced for account 2001, you would see the three rows shown in the following diagram.

```
id     balance
------ -----------
2001    14
2001    12
2001    10
```

The balances 14, 12, and 10 correspond to the balances in the detail accounts 3001, 4001, and 4002, which are all decedents of account 2001. The query that follows the body of the CTE then groups the rows that are produced by account ID and calculates the balance with the SUM function.

There are other ways to solve this problem without using CTEs. A batch that uses a stored procedure that calls itself or a cursor could produce the same result. However, the CTE is a query, and it can be used to define a view, something a stored procedure or a cursor-based batch cannot. The view definition that follows defines a view, which is the recursive query we used earlier, and then uses it to get the balance for a single account, account 2001.

```
CREATE VIEW Rollup
AS
WITH Rollup(id, parent, balance)
AS
(
SELECT id, parent, balance FROM DetailAccount
UNION ALL
SELECT R1.id, R1.parent, R2.balance
FROM RollupAccount R1
JOIN Rollup R2 ON R1.id = R2.parent
)
SELECT id, SUM(balance) balance FROM Rollup GROUP ID id
GO
— get the balance for account 2001
SELECT balance FROM rollup WHERE id = 2001
GO
balance
-----------
36

(1 row(s) affected)
```

One of the strengths of a recursive query is the fact that it is a query and can be used to define a view. In addition, a single query in SQL Server is always a transaction, which means that a recursive query based on a CTE is a transaction.

Recursive queries, like any recursive algorithm, can go on forever. By default, if a recursive query attempts to do more than 100 recursions, it will be aborted. You can control this with an OPTION(MAXRECURSION 10), for example, to limit recursion to a depth of 10. The example that follows shows its usage.

```
WITH Rollup(id, parent, balance)
AS
(
— body of CTE removed for clarity
)
SELECT id, SUM(balance) balance FROM Rollup GROUP BY id
OPTION (MAXRECURSION 10)
GO
```

APPLY Operators

T-SQL adds two specialized join operators: CROSS APPLY and OUTER APPLY. Both act like a JOIN operator in that they produce the Cartesian product of two tables except that no ON clause is allowed. The following SQL batch is an example of a CROSS APPLY between two tables.

```
CREATE TABLE T1
(
  ID int
)
CREATE TABLE T2
(
 ID it
)
GO
INSERT INTO T1 VALUES (1)
INSERT INTO T1 VALUES (2)
INSERT INTO T2 VALUES (3)
INSERT INTO T2 VALUES (4)
GO
SELECT COUNT(*) FROM T1 CROSS APPLY T2
--------
4
```

The APPLY operators have little utility with just tables or views; a CROSS JOIN could have been substituted in the preceding example and gotten the same results. It is intended that the APPLY operators be used with a table-valued function on their right, with the parameters for the table-valued function coming from the table on the left. The following SQL batch shows an example of this.

Table 7-5: Individual Sales, Including Quarter of Sale

Year	Quarter	Amount
2001	Q1	100
2001	Q2	120
2001	Q2	70
2001	Q3	55
2001	Q3	110
2001	Q4	90
2002	Q1	200
2002	Q2	150

2002	Q2	40
2002	Q2	60
2002	Q3	120
2002	Q3	110
2002	Q4	180

```
CREATE TABLE Belt
(
 model VARCHAR(20),
 length FLOAT
)
GO
— fill table with some data
DECLARE @index INT
SET @index = 5
WHILE(@index > 0)
BEGIN
INSERT INTO BELT VALUES ('B' + CONVERT(VARCHAR, @index), 10 * @index)
SET @index = @index – 1
END
GO
— make a table-valued function
CREATE FUNCTION Stretch (@length FLOAT)
RETURN @T TABLE
(
 MinLength FLOAT,
 MaxLength FLOAT
)
AS BEGIN
IF (@length > 20)
INSERT @T VALUES (@length – 4, @length + 5)
RETURN
END
GO
SELECT B.* S.MinLength, S.MaxLength FROM Belt AS B
CROSS APPLY Stretch(B.Length) AS S
GO
————————
B30, 26, 35
B40, 36, 45
B50, 46, 55
```

Table 7-6: Yearly Sales Broken Down by Quarter

Year	Q1	Q2	Q3	Q4
2001	100	190	165	90
2002	200	250	230	180

Figure 7-7: Tables for Hardware Store

The rows in the Belt table are cross-applied to the Stretch function. This function produces a table with a single row in it if the @length parameter passed into it is greater than 20; otherwise, it produces a table with no rows in it. The CROSS APPLY operator produces output when each table involved in the CROSS APPLY has at least one row in it. It is similar to a CROSS JOIN in this respect.

OUTER APPLY is similar to OUTER JOIN in that it produces output for all rows involved in the OUTER APPLY. The following SQL batch shows the results of an OUTER APPLY involving the same Belt table and Stretch function as in the previous example.

```
SELECT B.* S.MinLength, S.MaxLength FROM Belt AS B
CROSS APPLY Stretch(B.Length) AS S
GO
————————
B10, 6, 15
B20, 16, 25
B30, 26, 35
B40, 36, 45
B50, 46, 55
```

The preceding example could have been done using CROSS and OUTER JOIN. CROSS APPLY is required, however, when used in conjunction with XML data types in certain XML operations that will be discussed in Chapter 9.

PIVOT Command

SQL Server 2005 adds the PIVOT command to T-SQL, so named because it can create a new table by swapping the rows and columns of an existing table. PIVOT is part of the OLAP section of the SQL:1999 standard. There are two general uses for the PIVOT command. One it to create an analytical view of some data, and the other is to implement an open schema.

A typical analytical use of the PIVOT command is to covert temporal data into categorized data in order to make the data easier to analyze. Consider a table used to record each sale made as it occurs; each row represents a single sale and includes the quarter that indicates when it occurred. This sort of view makes sense for recording sales but is not easy to use if you want to compare sales made in the same quarter, year over year.

Table 7-5 lists temporally recorded sales. You want to analyze same-quarter sales year by year from the data in the table. Each row represents a single sale. Note that this table might be a view of a more general table of individual sales that includes a date rather than a quarterly enumeration.

The PIVOT command, which we will look at shortly, can convert this temporal view of individual sales into a view that has years categorized by sales in a quarter. Table 7-6 shows this.

Presenting the data this way makes it much easier to analyze same-quarter sales. This table aggregates year rows for each given year in the previous table into a single row. However, the aggregated amounts are broken out into quarters rather than being aggregated over the entire year.

The other use of the PIVOT command is to implement an open schema. An open schema allows arbitrary attributes to be associated with an entity. For example, consider a hardware store; its entities are the products that it sells. Each product has a number of attributes used to describe it. One common attribute of all products it the name of the product.

The hardware store sells "Swish" brand paint that has attributes of quantity, color, and type. It also sells "AttachIt" fastener screws, and these have attributes of pitch and diameter. Over time, it expects to add many other products to its inventory. With this categorization "Swish, 1 qt, green, latex" would be one product or entity, and "Swish, 1qt, blue, oil" would be another.

A classic solution to designing the database the hardware store will use to maintain its inventory is to design a table per product. For example, a table named Swish with columns for quantity, color, and type. This, of course, requires products and their attributes to be known and for those attributes to remain constant over time. What happens if the manufacturer of the Swish paint adds a new attribute, "Drying Time", but only to certain colors of paint?

An alternate solution is to have only two tables, regardless of the number of products involved or the attributes they have. In the case of the hardware store, there would be a Product table and a Properties table. The Product table would have an entry per product, and the Properties table would contain the arbitrary attributes of that product. The properties of a product are linked to it via a foreign key. This is called an open schema. Figure 7-7 shows the two ways of designing tables to represent the inventory of the hardware store.

The PIVOT operator can easily convert data that is stored using an open schema to a view that looks the same as the table-per-product solution. Next, we will look at the details of using PIVOT to analyze data and support open schemas, and then how to use PIVOT to work with open schemas. There is also an UNPIVOT operator, which can be used to produce the original open schema format from previously pivoted results.

Using PIVOT for Analysis

In this example, we are going to use PIVOT to analyze the sales data we showed in an earlier table. To do this, we build a SALES table and populate it with data, as is shown in the following SQL batch.

```
CREATE TABLE SALES
(
[Year] INT,
Quarter CHAR(2),
Amount FLOAT
)
GO
INSERT INTO SALES VALUES (2001, 'Q2', 70)
INSERT INTO SALES VALUES (2001, 'Q3', 55)
INSERT INTO SALES VALUES (2001, 'Q3', 110)
INSERT INTO SALES VALUES (2001, 'Q4', 90)
INSERT INTO SALES VALUES (2002, 'Q1', 200)
INSERT INTO SALES VALUES (2002, 'Q2', 150)
INSERT INTO SALES VALUES (2002, 'Q2', 40)
INSERT INTO SALES VALUES (2002, 'Q2', 60)
INSERT INTO SALES VALUES (2002, 'Q3', 120)
INSERT INTO SALES VALUES (2002, 'Q3', 110)
INSERT INTO SALES VALUES (2002, 'Q4', 180)
GO
```

To get a view that is useful for quarter-over-year comparisons, we want to pivot the table's Quarter column into a row heading and aggregate the sum of the values in each quarter for a year. The SQL batch that follows shows a PIVOT command that does this.

```
SELECT * FROM SALES
PIVOT
(SUM (Amount) – Aggregate the Amount column using SUM
FOR [Quarter] – Pivot the Quarter column into column headings
IN (Q1, Q2, Q3, Q4)) – use these quarters
AS P
GO
Year  Q1      Q2      Q3      Q4
----- ------- ------- ------- -------
2001  100     190     165     90
2002  200     250     230     180
```

The SELECT statement selects all the rows from SALES. A PIVOT clause is added to the SELECT statement. It starts with the PIVOT keyword followed by its body enclosed in parentheses. The body contains two parts separated by the FOR keyword. The first part of the body specifies the kind of aggregation to be performed. The argument of the aggregate function must be a column name; it cannot be an expression as it is when an aggregate function is used outside a PIVOT. The second part specifies the pivot column—that is, the column to pivot into a row—and the values from that column to be used as column headings. The value for a particular column in a row is the aggregation of the column specified in the first part, over the rows that match the column heading.

Note that it is not required to use all the possible values of the pivot column. You only need to specify the Q2 column if you wish to analyze just the year-over-year Q2 results. The SQL batch that follows shows this.

```
SELECT * FROM SALES
PIVOT
(SUM (Amount)
FOR [Quarter]
IN (Q2))
AS P
GO

Year    Q2
------- -----------
2001    190
2002    250
```

Figure 7-8: Rotating Properties

Figure 7-9: Basic PIVOT

Figure 7-10: Results of Open Schema Pivot

Note that the output produced by the PIVOT clause acts as though SELECT has a GROUP BY [Year] clause. A pivot, in effect, applies a GROUP BY to the SELECT that includes all the columns that are not either the aggregate or the pivot column. This can lead to undesired results, as shown in the SQL batch that follows. It uses essentially the same SALES table as the previous example, except that it has an additional column named Other.

```
CREATE TABLE SALES2
(
[Year] INT,
Quarter CHAR(2),
Amount FLOAT,
Other INT
)
INSERT INTO SALES2 VALUES (2001, 'Q2', 70, 1)
INSERT INTO SALES2 VALUES (2001, 'Q3', 55, 1)
INSERT INTO SALES2 VALUES (2001, 'Q3', 110, 2)
INSERT INTO SALES2 VALUES (2001, 'Q4', 90, 1)
INSERT INTO SALES2 VALUES (2002, 'Q1', 200, 1)
INSERT INTO SALES2 VALUES (2002, 'Q2', 150, 1)
INSERT INTO SALES2 VALUES (2002, 'Q2', 40, 1)
INSERT INTO SALES2 VALUES (2002, 'Q2', 60, 1)
INSERT INTO SALES2 VALUES (2002, 'Q3', 120, 1)
INSERT INTO SALES2 VALUES (2002, 'Q3', 110, 1)
INSERT INTO SALES2 VALUES (2002, 'Q4', 180, 1)

SELECT * FROM Sales2
PIVOT
(SUM (Amount)
FOR Quarter
IN (Q3))
AS P
GO
Year    Other    Q3
------- -------- -----------
2001    1        55
2002    1        115
2001    2        110
```

Note that the year 2001 appears twice, once for each value of Other. The SELECT that precedes the PIVOT keyword cannot specify which columns to use in the PIVOT clause. However, a subquery can be used to eliminate the columns not desired in the pivot, as shown in the SQL batch that follows.

```
SELECT * FROM
(Select Amount, Quarter, Year from Sales2
) AS A
PIVOT
(SUM (Amount)
FOR Quarter
IN (Q3))
AS P
GO
Year    Q3
------- -----------
2001    165
2002    230
```

A column named in the FOR part of the PIVOT clause may not correspond to any values in the pivot column of the table. The column will be output, but will have null values. The following SQL batch shows this.

```
SELECT * FROM SALES
PIVOT
(SUM (Amount)
FOR [Quarter]
IN (Q2, LastQ))
As P
GO
Year    Q2          LastQ
------- ----------- -----------
2001    190         NULL
2002    250         NULL
```

Note that the Quarter column of the SALES table has no value "LastQ", so the output of the PIVOT lists all the values in the LastQ column as NULL.

Using PIVOT for Open Schemas

Using PIVOT for an open schema is really no different from using PIVOT for analysis, except that we don't depend on PIVOT's ability to aggregate a result. The open schema has two tables, a Product table and a Properties table, as was shown in Figure 7-7. What we want to do is to take selected rows from the Properties table and pivot them—that is, rotate them—and then add them as columns to the Product table. This is shown in Figure 7-8.

Figure 7-9
shows the PIVOT we will use to select the line from the Product table for "Swish" products and joint them with the corresponding pivoted lines from the Properties table.

This query selects row from the Properties table that have a string equal to "color", "type", or "amount" in the value column. They are selected from the value column because value is the argument of the MAX function that follows the PIVOT keyword. The strings "color", "type", and "amount" are used because they are specified as an argument of the IN clause after the FOR keyword. Note that the arguments of the IN clause must be literal; there is no way to calculate them—for example, by using a subquery.

The results of the pivot query in Figure 7-9 are shown in Figure 7-10.

Note that the columns that were selected from the Properties table now appear as rows in the output.

Ranking and Windowing Functions

SQL Server 2005 adds support for a group of functions known as ranking functions. At its simplest, ranking adds an extra value column to the resultset that is based on a ranking algorithm being applied to a column of the result. Four ranking functions are supported.

ROW_NUMBER() produces a column that contains a number that corresponds to the row's order in the set. The set must be ordered by using an OVER clause with an ORDER BY clause as a variable. The following is an example.

```
SELECT orderid, customerid,
    ROW_NUMBER() OVER(ORDER BY orderid) AS num
FROM orders
WHERE orderid < 10400
AND  customerid <= 'BN'

produces

orderid   customerid num
_____- ____ ___________
10248    VINET   1
10249    TOMSP   2
10250    HANAR   3
10251    VICTE   4
10252    SUPRD   5
10253    HANAR   6
10254    CHOPS   7
10255    RICSU   8
... more rows
```

Note that if you apply the ROW_NUMBER function to a nonunique column, such as customerid in the preceding example, the order of customers with the same customerid (ties) is not defined. In any case, ROW_NUMBER produces a monotonically increasing number; that is, no rows will ever share a ROW_NUMBER.

```
SELECT orderid, customerid,
    ROW_NUMBER() OVER(ORDER BY customerid) AS num
FROM orders
WHERE orderid < 10400
AND  customerid <= 'BN'

produces

orderid   customerid num
_____- _____ ___________
10308    ANATR   1
10365    ANTON   2
10355    AROUT   3
10383    AROUT   4
10384    BERGS   5
10278    BERGS   6
10280    BERGS   7
10265    BLONP   8
10297    BLONP   9
10360    BLONP   10
```

RANK() applies a monotonically increasing number for each value in the set. The value of ties, however, is the same. If the columns in the OVER(ORDER BY) clause have unique values, the result produced by RANK() is identical to the result produced by ROW_NUMBER(). RANK() and ROW_NUMBER() differ only if there are ties. Here's the second earlier example using RANK().

```
SELECT orderid, customerid,
    RANK() OVER(ORDER BY customerid) AS [rank]
FROM orders
WHERE orderid < 10400
AND  customerid <= 'BN'

produces

orderid   customerid rank
_____- _____ ___________
10308    ANATR   1
10365    ANTON   2
10355    AROUT   3
10383    AROUT   3
10384    BERGS   5
10278    BERGS   5
10280    BERGS   5
10265    BLONP   8
10297    BLONP   8
10360    BLONP   8
... more rows
```

Note that multiple rows have the same rank if their customerid is the same. There are holes, however, in the rank column value to reflect the ties. Using the DENSE_RANK() function works the same way as RANK() but gets rid of the holes in the numbering. NTILE(n) divides the resultset into "n" approximately even pieces and assigns each piece the same number. NTILE(100) would be the well-known (to students) percentile. The following query shows the difference between ROW_NUMBER(), RANK(), DENSE_RANK(), and TILE(n).

```
SELECT orderid, customerid,
    ROW_NUMBER() OVER(ORDER BY customerid) AS num,
    RANK()    OVER(ORDER BY customerid) AS [rank],
    DENSE_RANK() OVER(ORDER BY customerid) AS [denserank],
    NTILE(5)   OVER(ORDER BY customerid) AS ntile5
FROM orders
WHERE orderid < 10400
AND  customerid <= 'BN'

produces

orderid   customerid num  rank  denserank ntile5
_____-   _____       ___  ___    ____-    ___
10308    ANATR        1    1       1       1
10365    ANTON        2    2       2       1
10355    AROUT        3    3       3       2
10383    AROUT        4    3       3       2
10278    BERGS        5    5       4       3
10280    BERGS        6    5       4       3
10384    BERGS        7    5       4       4
10265    BLONP        8    8       5       4
10297    BLONP        9    8       5       5
10360    BLONP       10    8       5       5
```

The ranking functions have additional functionality when combined with windowing functions. Windowing functions divide a resultset into partitions, based on the value of a PARTITION BY clause inside the OVER clause. The ranking functions are applied separately to each partition. Here's an example.

```
SELECT *,
 RANK() OVER(PARTITION BY COUNTRY ORDER BY age) AS [rank]
FROM
(
 SELECT lastname, country,
  DATEDIFF(yy,birthdate,getdate())AS age
 FROM employees
) AS a

produces

lastname   country    age   rank
_________ _________- _____-  ___
Dodsworth    UK       37     1
Suyama       UK       40     2
King         UK       43     3
Buchanan     UK       48     4
Leverling    USA      40     1
Callahan     USA      45     2
Fuller       USA      51     3
Davolio      USA      55     4
Peacock      USA      66     5
```

There are separate rankings for each partition. An interesting thing to note about this example is that the subselect is required because any column used in a PARTITION BY or ORDER BY clause must be available from the columns in the FROM portion of the statement. In our case, the seemingly simpler statement that follows:

```
SELECT lastname, country,
 DATEDIFF(yy,birthdate,getdate())AS age,
 RANK() OVER(PARTITION BY COUNTRY ORDER BY age) AS [rank]
FROM employees
```

wouldn't work; instead, you'd get the error "Invalid column name 'age'". In addition, you can't use the ranking column in a WHERE clause, because it is evaluated after all the rows are selected, as shown next.

```
— 10 rows to a page, we want page 40

— this won't work
SELECT
 ROW_NUMBER() OVER (ORDER BY customerid, requireddate) AS num,
 customerid, requireddate, orderid
FROM orders
WHERE num BETWEEN 400 AND 410

— this will
SELECT * FROM
(
SELECT
 ROW_NUMBER() OVER (ORDER BY customerid, requireddate) AS num,
 customerid, requireddate, orderid
FROM orders
) AS a
WHERE num BETWEEN 400 AND 410
```

Although the preceding case looks similar to selecting the entire resultset into a temporary table, with num as a derived identity column, and doing a SELECT of the temporary table, in some cases the engine will be able to accomplish this without the complete set of rows. Besides being usable in a SELECT clause, the ranking and windowing functions are also usable in the ORDER BY clause. This gets employees partitioned by country and ranked by age, and then sorted by rank.

```
SELECT *,
 RANK() OVER(PARTITION BY COUNTRY ORDER BY age)) AS [rank]
FROM
(
 SELECT lastname, country,
  DATEDIFF(yy,birthdate,getdate())AS age
 FROM employees
) AS a
ORDER BY RANK() OVER(PARTITION BY COUNTRY ORDER BY age), COUNTRY

produces

lastname   country     age     rank
_________ ________- _____- _________
Dodsworth      UK      37     1
Leverling      USA     40     1
Suyama         UK      40     2
Callahan       USA     45     2
King           UK      43     3
Fuller         USA     51     3
Buchanan       UK      48     4
Davolio        USA     55     4
Peacock        USA     66     5
```

You can also use other aggregate functions (either system-defined aggregates or user-defined aggregates that you saw in Chapter 5) with the OVER clause. When it is used in concert with the partitioning functions, however, you get the same value for each partition. This is shown next.

```
— there is one oldest employee age for each country
SELECT *,
 RANK() OVER(PARTITION BY COUNTRY ORDER BY age) AS [rank],
 MAX(age) OVER(PARTITION BY COUNTRY) AS [oldest age in country]
FROM
(
 SELECT lastname, country,
  DATEDIFF(yy,birthdate,getdate())AS age
 FROM employees
) AS a

produces

lastname   country   age    rank   oldest age
                             in country
_________ ________- _____  ____    _____-
Dodsworth      UK      37     1     48
Suyama         UK      40     2     48
King           UK      43     3     48
Buchanan       UK      48     4     48
Leverling      USA     40     1     66
Callahan       USA     45     2     66
Fuller         USA     51     3     66
Davolio        USA     55     4     66
Peacock        USA     66     5     66
```

Transaction Abort Handling

Error handling in previous versions of SQL Server has always been seen as somewhat arcane, compared with other procedural languages. You had to have error handling code after each statement, and to have centralized handling of errors, you need GOTO statements and labels. SQL Server 2005 introduces a modern error handling mechanism with TRY/CATCH blocks. The syntax follows.

```
BEGIN TRY
  { sql_statement | statement_block }
END TRY
BEGIN CATCH TRAN_ABORT
  { sql_statement | statement_block }
END CATCH
```

The code you want to execute is placed within a TRY block. The TRY block must be immediately followed by a CATCH block in which you place the error handling code. The CATCH block can only handle transaction abort errors, so the XACT_ABORT setting needs to be on in order for any errors with a severity level less than 21 to be handled as transaction abort errors. Errors with a severity level of 21 or higher are considered fatal and cause SQL Server to stop executing the code and sever the connection.

When a transaction abort error occurs within the scope of a TRY block, the execution of the code in the TRY block terminates and an exception is thrown. The control is shifted to the associated CATCH block. When the code in the CATCH block has executed, the control goes to the statement after the CATCH block. TRY/CATCH constructs can be nested, so to handle exceptions within a CATCH block, write a TRY/CATCH block inside the CATCH. The following code shows a simple example of the TRY/CATCH block.

```
—make sure we catch all errors
SET XACT_ABORT ON
BEGIN TRY
 —start the tran
 BEGIN TRAN
 —do something here
 COMMIT TRAN
END TRY
BEGIN CATCH TRAN_ABORT
 ROLLBACK
 —cleanup code
END CATCH
```

Notice how the first statement in the CATCH block is the ROLLBACK. It is necessary to do the ROLLBACK before any other statements that require a transaction. This is because SQL Server 2005 has a new transactional state: "failed" or "doomed." The doomed transaction acts like a read-only transaction. Reads may be done, but any statement that would result in a write to the transaction log will fail with error 3930:

```
Transaction is doomed and cannot make forward progress. Rollback Transaction.
```

However, work is not reversed and locks are not released until the transaction is rolled back.

We mentioned previously that errors had to be transactional abort errors in order to be caught. This raises the question: What about errors created through the RAISERROR syntax—in other words, errors that you raise yourself? In SQL Server 2005, RAISERROR has a new option called TRAN_ABORT, which tags the raised error as a transactional abort error, which therefore will be handled in the CATCH block.

Where Are We?

With the inclusion of the CLR in SQL Server 2005 and the ability to use .NET languages natively from within SQL Server, there has been speculation on the future of T-SQL. T-SQL continues to be advanced and remains the best (and in some cases the only) way to accomplish many things. We firmly believe that the enhancements to T-SQL in this release of SQL Server show the importance of T-SQL and its power and future.