F

Transactions

Transactions play a very important role in how you access your data. A transaction is a sequence of operations performed as a single logical unit of work. There are four classic properties that transactions are required to support, commonly known as the ACID test. This is an acronym for the following:

- **Atomicity:** Every action that participates in a transaction must be performed as a logical unit of work, such that every action either does or doesn't happen.
- **Consistency:** Once the transaction is finished, the system must be left in a consistent condition, including physical resources like data, indexes, and the links between data pages.
- **Isolation:** All modifications within a transaction must be isolated from other transactions. When executing commands within a transaction, no other transaction may see data in an intermediate state.
- **Durability:** Once the transaction has completed, all actions that have taken place in the transaction must be permanently persisted.

The fact that a transaction meets these four properties is essential to the stability of your database systems. You'll see, however, that although each of these properties can be met, some control is left in the hands of the programmer.

Coding Transactions

A transaction starts with a `BEGIN TRANSACTION` statement. Executing `BEGIN TRANSACTION` puts your connection to the database in a state where nothing that you do will be permanently saved to physical storage until you explicitly tell it to. You'll have a chance to examine this in the next set of examples.

In the first example, you'll start a transaction, insert a row, and then roll the transaction back (in other words, the effects of the transaction are completely cancelled and the database returns to its original state). You'll work with the `artist` table from the last appendix, and start by getting the base level list of artists in your table.

```
SELECT artistId, name
FROM artist
```

You see that you have three artists saved (keep in mind that the `artistId` column values will vary, depending on how many values have been entered, plus how many errors have occurred).

```
artistId        name
--------------------------------------
27              jethro tull
1               the beatles
2               the who
```

Next, you execute the following batch of commands:

```
BEGIN TRANSACTION --start a transaction

   -- insert two records
   INSERT INTO artist (name)
   VALUES ('mariah carey')
   INSERT INTO artist (name)
   VALUES ('britney spears')

   -- make sure that they are in the database
   SELECT artistId, name FROM artist
```

This returns the following result:

```
artistId        name
----------------------------------------
37              britney spears
27              jethro tull
36              mariah carey
1               the beatles
2               the who
```

It's too much to assume the database server has good taste in music; obviously you're too smart to think that, so you roll back the transaction, followed by yet another check of the rows in the table.

```
ROLLBACK TRANSACTION
SELECT artistId, name FROM artist
```

Now the rows are gone, as if they never existed.

```
artistId        name
---------------------------------------
27              jethro tull
1               the beatles
2               the who
```

Technically speaking, the rows did physically exist, but SQL Server had a record of what it had done to the server, so you can either commit them as a group or, in this case here, not commit them.

Transactions can be nested, such that, for every `BEGIN TRANSACTION` statement that you execute against the database, you have to execute one `COMMIT TRANSACTION`. On the other side of the coin, the `ROLLBACK TRANSACTION` statement undoes an unlimited number of transactions.

You can tell how many transactions have been started by examining the `@@trancount` global variable. In the following example, you'll see the effects of executing `BEGIN TRANSACTION` and `COMMIT TRANSACTION` statements, by looking at the `@@trancount` value before and after transaction commands:

```
SELECT @@trancount AS zero
BEGIN TRANSACTION
   SELECT @@trancount AS one
   BEGIN TRANSACTION
      SELECT @@trancount AS two
      BEGIN TRANSACTION
      SELECT @@trancount AS three
      COMMIT TRANSACTION
   SELECT @@trancount AS two
   COMMIT TRANSACTION
SELECT @@trancount AS one
COMMIT TRANSACTION
SELECT @@trancount AS zero
```

This returns the following output. Note that it increments from zero to three and then decrements back down to zero.

```
zero
----------------
0

one
----------------
1

two
----------------
2
```

```
three
----------------
3

two
----------------
2

one
----------------
1

zero
----------------
0
```

You must commit every transaction or the resources will be locked from your other transactions, based on the isolation part of the ACID test. However, as stated before, `ROLLBACK` rolls back all of the `BEGIN TRANSACTION` statements at once.

```
SELECT @@trancount AS one
BEGIN TRANSACTION
   SELECT @@trancount AS two
   BEGIN TRANSACTION
      SELECT @@trancount AS three
      BEGIN TRANSACTION
         SELECT @@trancount AS four
         ROLLBACK TRANSACTION
         SELECT @@trancount AS zero
```

This will return

```
one
----------------
0

two
----------------
1

three
----------------
2

four
----------------
3

zero
----------------
0
```

This is a great thing and also a problem. For starters, you'll likely have several `COMMIT` statements after the rollback, because you'll usually use logic that states the following: If it works, commit it; if it doesn't, roll it back. After the rollback, you must then tell every following statement that you don't want to execute it. Hence SQL Server gives you an additional resource for transactions that allows you to selectively roll back certain parts of a transaction.

Selective Rollback

There are two commands involved in setting up a selective rollback and one of them is new: `SAVE TRANSACTION`. A save point can be named (based on the usual criteria for identifiers, though only the first 32 characters are significant) as follows:

```
SAVE TRANSACTION <savepoint name>
```

Then, if it's decided that part of the transaction has failed, you can roll back part of the command but not the rest by executing a command like this:

```
ROLLBACK TRANSACTION <savepoint name>
```

Note that this is different from `BEGIN TRANSACTION <transaction name>`.

So, in this final example on transactions, you'll use the `SAVE TRANSACTION` command to selectively roll back one of the commands that you execute and not the other:

```
BEGIN TRANSACTION --start a transaction

--insert two records
INSERT INTO artist (name)
VALUES ('moody blues')

SAVE TRANSACTION britney
INSERT INTO artist (name)
VALUES ('britney spears')

--make sure that they are in the database
SELECT artistId, name FROM artist
```

which gives this output:

```
artistId      name
---------------------------------------------
45            britney spears
27            jethro tull
44            moody blues
1             the beatles
2             the who
```

Then you roll back to the britney save point you created, commit the major transaction, and select your rows to see the state of your artist table.

```
ROLLBACK TRANSACTION britney

COMMIT TRANSACTION

SELECT artistId, name FROM artist
```

You see that it *has* rolled back the britney spears row but not the moody blues row.

```
artistId        name
----------------------------------------
27              jethro tull
44              moody blues
1               the beatles
2               the who
```

You'll make use of the SAVE TRANSACTION mechanism quite often when building transactions. It allows you to roll back only parts of a transaction and to decide what action to take. It's especially important when you get to stored procedures, as a stored procedure isn't allowed to change the value of @@trancount as a result of its operation.

The Scope of Transactions

A **batch** is a logical group of T-SQL statements that are parsed and then executed together. You use the GO statement to split code up into batches; GO is not a T-SQL command, but just a keyword recognized by SQL Server command utilities like Query Analyzer.

A transaction is scoped to an entire connection to SQL Server, and may theoretically span many batches of statements sent between the client and the server. Note that a connection isn't the same thing as a user. A user may have many connections to the server and, in fact, a user may be forced to wait for one of his or her own connections to complete a transaction before continuing with work. One transaction, or a set of nested transactions, has to be completed before the next one can start.

How does SQL Server make sure that, if a connection is cut before a transaction has finished, everything is rolled back to its original state? How does SQL Server make sure that transactions don't overlap with each other, thereby violating the isolation requirement of the ACID test? In the next section, you'll see the locking mechanisms that SQL Server employs to keep track of transactional activity on the database.

Locking

Locking in SQL Server is a bit like the occupied sign on a bathroom door. It tells the outside world that something is going on inside and everyone else will just have to wait. (Of course, in this case you certainly don't have a problem waiting!)

Locking plays a vital role in ensuring transactions work correctly and obey the ACID rules. Rolling back one transaction would be dangerous if another transaction had already begun working with the same set of data. This situation–many users trying to access resources at the same time–is generally known as **concurrency**. In this section, I'll present a basic overview of how locking is internally implemented (as opposed to the treatment of an optimistic lock and how this prevents data corruption if multiple processes try to access the data simultaneously).

Lock Types

SQL Server internally implements a locking scheme that allows a transaction or operation to associate a token with a given resource or collection of resources. The token indicates to other processes that the resource is locked. SQL Server prevents the explicit setting or clearing of locks, so a resource with such a token is protected. Depending upon the level of usage, the following resources may be locked (the **lock types** are listed in order of decreasing granularity):

- **Row:** An individual row in a table
- **Key** or **Key-Range:** A key or range of keys in an index
- **Page:** The 8K page structure that has an index or data
- **Table:** An entire table, that is, all rows and index keys
- **Database:** An entire database

Note that there is also an **extent** *lock, which is normally used when allocating space in a database and which has a range of eight 8K pages.*

The initial type of lock employed is determined by SQL Server's optimizer, which also decides when to change from one lock type to another in order to save memory and increase performance. Generally, the optimizer will choose the smallest lock–the row type–when possible but, if there is a likelihood of needing to lock all or a range of rows, the table type may be chosen. By using optimizer hints on your queries, you can adjust to some extent the types of locks used. This will be discussed shortly.

Lock Modes

To architect systems well, it's useful to understand how SQL Server selects the level of granularity to employ. First I'll describe the different **lock modes** that exist:

- **Shared locks:** These are for when the connection is simply looking at the data. Multiple connections may have shared locks on the same resource.
- **Update locks:** These allow you to state that you are planning to update the data but you aren't ready to do so–for example, if you're still processing a `WHERE` clause to determine which rows you need to update. You may still have shared locks on the table, and other connections may be issued during the period of time that you're still preparing to update.

- **Exclusive locks:** These give exclusive access to the resource. No other connection may have access at all. These are frequently employed after a physical resource has been modified within a transaction.
- **Intent locks:** These convey information to other processes that have or are about to create locks of some type. There are three types of intent locks: **intent shared**, **intent exclusive**, and **shared with intent exclusive** (reading with intent to make changes). For anyone who has ever been around divers, intent locks are analogous to the buoys that divers place on the surface of the water to mark their position. Because you see the buoy, you know that there is activity going on underwater. SQL Server sees these markers on objects and knows that, if, for example, an intent shared lock is located on a table, then it isn't possible to get an exclusive lock on that table as other connections are using the resources already.

As with the previous list of resources that may be locked, there are a few additional modes that I'll mention in passing that are outside the scope of the appendix. **Schema** *locks are used when the database structure is changed, a stored procedure is recompiled, or anything else changes to the physical structure of the database.* **Bulk update** *locks are used when performing bulk updates. See SQL Server 2000 Books Online for more information.*

So now you know that there are locks at various levels of the process, and anytime you do anything in SQL Server, tokens are set that tell other processes what you're doing to the resource. This is to force them to wait until you've finished using the resource if they need to do a task that is incompatible with what you're currently doing–for instance, trying to look at a record that you're modifying or trying to delete a record that you're currently fetching to read.

Isolation Levels

Locks are held for the duration of atomic operations, depending on the operation you've asked SQL Server to perform. Consider, for example, that you start a transaction on a table called `tableX` as shown:

```
SELECT * FROM dbo.tableX
```

Locks (in this case shared locks) are held on this resource for a period of time, based on what is known as the **isolation level**. SQL Server provides the following four isolation levels, ordered here from least to most restrictive:

- **Read uncommitted:** Ignores all locks and doesn't issue locks. In the read uncommitted isolation level, you're allowed to see data from atomic operations that haven't yet finished, and that might be rolled back. Note that no modification statement actually operates in this isolation level; if a row is modified within a transaction, locks are still left that prevent others from corrupting changes.

This is a very dangerous thing to do in general. Because you're allowed to see even uncommitted transactions, it's never clear if what you've read is of value. It can, however, be used during reporting if the reporting can accept the margin of error this may allow–the data is read in a way that may not meet all constraints laid down and it may vanish after reading.

- **Read committed:** This is by far the most commonly used isolation level, as it provides a reasonable balance between safety and concurrency. You can't see uncommitted data, though you do allow unrepeatable reads (a condition where, if you read in a set of rows, you expect the rows to exist for the length of the transaction) and **phantom rows**, which are new records that are created by other users while you are in your transaction.

  Ordinarily, this is the best choice for the isolation level and is, in fact, the SQL Server default.

- **Repeatable read:** A repeatable read guarantees that, if a user needs to carry out the same `SELECT` statement twice, and provided the user hasn't changed the data between these two `SELECT` operations, the user will get back exactly the same data twice. An unrepeatable read situation can't guarantee this as another concurrent user might have changed the data.

  Repeatable read takes the protection of the read committed isolation level and prevents unrepeatable reads by placing locks on all data that is being used in a query, preventing other connections from changing data that already has been read. When executing a query the second time, it will return at least the same set of rows as it did the first time. However, it doesn't prevent phantom rows.

- **Serializable:** This isolation level places range locks on the indexes and data sets, thus preventing users from making any phantom rows.

To illustrate what I've been saying, let's look at an example of what can occur if you use the default read committed isolation level. First, you build a table that you'll use for the test, and you add some records.

```
CREATE TABLE testIsolationLevel
(
   testIsolationLevelId INT IDENTITY,
   value varchar(10)
)

INSERT INTO dbo.testIsolationLevel(value)
VALUES ('Value1')
INSERT INTO dbo.testIsolationLevel(value)
VALUES ('Value2')
```

Now you can execute the following commands in one connection to SQL Server:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED

--set for illustrative purposes only. If you do not set this on a connection
--it will be in read committed mode anyway because this is the default.

BEGIN TRANSACTION
SELECT * FROM dbo.testIsolationLevel
```

and you'll see the following results:

```
testIsolationLevelId          value
-----------------------------------------------
1                             Value1
2                             Value2
```

Using a different connection, you execute the following:

```
DELETE FROM dbo.testIsolationLevel
WHERE testIsolationLevelId = 1
```

Then, going back to your first connection–which is still open and within a transaction–you execute

```
SELECT * FROM dbo.testIsolationLevel

COMMIT TRANSACTION
```

and you'll have no problems deleting the row, as shown:

```
testIsolationLevelId          value
-----------------------------------------------
2                             Value2
```

This was an *unrepeatable* read. Next, you drop the table, reinitialize, and rerun your little test–changing the isolation level to

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

This time, when you try to execute the `DELETE` statement in the second connection, it won't complete until your first connection is completed. The second connection has been **blocked** (or locked out) by the first and can't execute its commands. Hence, your second `SELECT` statement will return at least the same values–in other words, it's a repeatable read.

But you still have to consider the problem of phantoms (that is, phantom rows). You've prevented anyone from removing rows, but what about adding rows? From the name "repeatable read," you might tend to believe that you're protected against this situation. However, retry your situation. You'll drop the table and re-create it, inserting the same two rows. Then you execute the original transaction over connection one as follows:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED

BEGIN TRANSACTION
SELECT * FROM dbo.testIsolationLevel
```

and again you'll see the following results:

```
testIsolationLevelId        value
----------------------------------------------
1                           Value1
2                           Value2
```

In a second connection, you execute the following command:

```
INSERT INTO dbo.testIsolationLevel(value)
VALUES ('Value3')
```

This isn't blocked as the `DELETE` was, but is allowed. Now, if you execute the following command in the first conection as before:

```
SELECT * FROM dbo.testIsolationLevel

COMMIT TRANSACTION
```

you get an additional row:

```
testIsolationLevelId        value
----------------------------------------------
1                           Value1
2                           Value2
3                           Value3
```

Now you have a *repeatable* read, but you have a phantom row too. Finally, if you go back and run this example one last time, but specify `SERIALIZABLE` as the isolation level, the `INSERT` in this example will have to wait for the connection to finish.

From these examples, the serializable isolation level seems like the best choice, because you might want to prevent other users from accessing your data at the same time as you're using it. If you're modifying data using a transaction, you'll most likely want to ensure that a given situation exists until all the operations have completed.

Whenever you modify data, SQL Server places exclusive locks on the resources that you're modifying. If another user is trying to access that particular resource, it must wait. However, all of the other resources that are being used as part of the transaction are locked as well, and a concurrent user can't access any of these resources while they are being used in the transaction. As the level of isolation increases, SQL Server must place and hold locks on resources for longer in the process.

- **Read uncommitted:** No locks are issued and none are honored, even exclusive locks.
- **Read committed:** Shared locks are issued when looking at a resource, but once the process has finished with them, they are released.
- **Repeatable read:** The same shared locks are issued as with read committed; however, they aren't released until the end of the transaction.
- **Serializable:** All the ranges of data (including whole tables) that might meet the criteria in any reads you make are locked, and no outside user can make any changes to this data.

One thing to note. When using any isolation level other than serializable, you leave yourself open to a very small hole in data integrity. Consider what happens in triggers and check constraints, which use functions to access other tables to see if conditions are being met prior to saving a record. Depending on your isolation level, you could check that a certain condition exists and, 10 milliseconds later, another user could conceivably change data in your system to the point where the data is within specifications. Another 10 milliseconds later, you could commit the now invalid data to disk.

This is further evidence to support my earlier claim that it would be best if serializable was the chosen isolation level for all transactions. However, I should state the reason why read committed is the default isolation level for SQL Server. It has everything to do with balancing the potentially conflicting issues of consistency and concurrency.

Using the serializable isolation level can prevent other users from getting to the data that they need and greatly reduce concurrency, especially when you have many users using the same resources. Using read committed leaves a hole in your defenses, usually for an extremely small period of time (usually in the order of milliseconds), with minimal risk.

It's recommended that serializable or repeatable read only be used when absolutely necessary. An example of such a case might be a banking or financial system where even the smallest possible error in a user's account could cause real problems.

The other recommendation is to keep all transactions as small as possible (but not shorter than necessary), so as to keep locks as short as possible. This entails trying to keep transactions in a single batch. Whatever resources are locked during a transaction will stay locked while SQL Server waits for the execution to finish. Performance obviously deteriorates the longer the transaction execution time and the greater the isolation level. SQL Server will always try to complete every batch it begins executing. If you're disconnected in the middle of an open-ended transaction, the connection won't die immediately, further compromising performance.

Optimizer Hints

At this point, I should make a brief mention of table level **optimizer hints**. When you don't want an entire connection or operation to use a particular isolation level, but you simply want a table in a statement to behave in a certain manner, you can specify the isolation level via an optimizer hint.

```
      SELECT <fieldName1>, <fieldName2>, … , <fieldNameN>
      from <tableName1> WITH (optimizerHint1, optimizerHint2,...,optimizerHintX)
      WHERE <where conditions>
```

Imagine you have a `customer` table, for example, and an `invoice` table that has a foreign key to the customer table.

```
      SELECT invoice.date AS invoiceDate
      FROM dbo.customer WITH (READUNCOMMITTED)
      JOIN dbo.invoice
         ON customer.customerId = invoice.customerId
```

In this instance, there would be no shared locks placed on the `customer` table, but there would be on the `invoice` table as resources are accessed. There are optimizer hints for each of the isolation level values, as well as a few others to force certain kinds of locks:

- `READUNCOMMITTED` (also known as `NOLOCK`): No locks are created or honored.
- `READCOMMITTED`: Honors all locks, but doesn't hold onto the locks once it's finished with them (the default).
- `REPEATABLEREAD`: Never releases locks once they've been acquired.
- `SERIALIZABLE` (also known as `HOLDLOCK`): Never releases locks until the transaction has ended, and locks all other statements from modifying rows that might ever match any `WHERE` clause that limits the selection of data in the hinted table.
- `READPAST`: Only valid for `SELECT`s; this tells the query processor to skip over locked rows and move to unlocked ones. Only ignores row locks.
- `ROWLOCK`: Forces the initial locking mechanism chosen for an operation to be a row-level lock, even if the optimizer would have chosen a less granular lock.
- `TABLOCK`: Forces a table lock to be used even when a more granular lock would normally have been chosen.
- `TABLOCKX`: Same as the `TABLOCK`, but uses exclusive locks instead of whatever lock type would have been chosen.
- `UPDLOCK`: Forces the use of an update lock instead of a shared lock.

Although each of these optimizer hints has its uses and can be helpful, they should be used with extreme care and, in most cases, it will be best to use isolation levels rather than trying to manipulate lock types and lock modes directly.

Deadlocks

A deadlock is a situation that occurs when two (or more) connections each have resources locked that the other connection needs. To illustrate, suppose you have two connections–A and B–that issue commands to update two tables–`tableA` and `tableB`. The problem occurs because connection A updates `tableA` first and then `tableB`, whereas connection B updates the same tables in the reverse order.

Connection A	**Connection B**
Connect to table A	Connect to table B
Table A locked	Table B locked
Update carried out	Update carried out
Attempt to connect to table B	Attempt to connect to table A
Locked out of table B	Locked out of table A

Because there is no `WHERE` clause, the entire table will have to be locked to perform the update, so a table lock is applied to `tableA` while connection A is using it. The same happens with connection B's command to update all of the rows in `tableB`. When the two connections try to execute their second `UPDATE` statements, they both get blocked–creating the deadlock. SQL Server then nominates one of the connections to become the deadlock victim and releases the other connection, rolling back the locks that were held on a resource.

To determine which connection ends the deadlock, SQL Server chooses the connection that is the least expensive to roll back. However, there are settings you can use to raise a connection's deadlock priority, to help SQL Server choose a less important connection to fail. The following `SET` command will set the connection deadlock priority to either `LOW` or `NORMAL`. `NORMAL` is the default; and a `LOW` setting for a connection means that it's most likely to be the victim of the deadlock and fail.

```
SET DEADLOCK_PRIORITY [LOW | NORMAL]
```

Deadlocks are annoying but fairly rare, and if you anticipate their occurrence and produce code to avoid them, they can be easily handled. Applications can simply be coded to look for a deadlock error message (`1205`) and have it resubmit the command; it's a highly unlikely occurrence to get a deadlock the second time around.

Note that this isn't to be confused with the blocking I discussed earlier. Deadlocks cause one of the connections to fail and raise an error; blocked connections wait until the connection times out–based on the `LOCK_TIMEOUT` *value–and then roll back the connection. Set this* `LOCK_TIMEOUT` *property by executing the following:*

```
SET LOCK_TIMEOUT <timeoutTimeInMilliseconds>
```

The key to minimizing deadlocks is, once again, to keep transactions short. The shorter the transaction, the less chance it has of being blocked or deadlocked by other transactions. One suggestion that is continually mentioned is to always access objects in the same order in transactions, perhaps alphabetically. Of course, such suggestions will only work if everybody abides by the rule, and this can be more of a headache than dealing with deadlocks.

Summary

In this appendix, you've seen how transactions play a very important role in how you access your data. I first discussed that there are four classic properties that transactions are required to support, commonly known as the ACID test. Now you need to look back and consider how well they are implemented in SQL Server transactions.

- ❑ **Atomicity:** The one thing that is kind of different here is that you have mechanisms (save points) that allow you to have subordinate transactions that can be rolled back within a transaction. In this manner, you have more control over how atomic a transaction is.
- ❑ **Consistency:** SQL Server transactions are very consistent, in that the system is left in a usable state, but only if you take proper precautions with the isolation level. At `READ COMMITTED` level (the default and best performing), you have a tiny hole in consistency in that once a transaction has read in data and released it, this data may change. However, if that data changes to make the operation you're doing illegal, and the transaction doesn't finish until after that takes place, the data may be left in an inconsistent state. SQL Server DBAs have generally been willing to take this risk because the only safe way to protect the data perfectly would be to use `SERIALIZABLE`, and this is very restrictive.
- ❑ **Isolation:** This is related to consistency, in that how well you deal with isolation levels is how well you're isolated. The big problem in isolation levels is `READ UNCOMMITTED`. This will read any data, in any state, even if it will be later rolled back. This is really a bad idea, but you'll use this isolation level all of the time because not only does it not honor locks, it doesn't leave them.
- ❑ **Durability:** This is one case where you have no worries. When the final `COMMIT TRANSACTION` statement is executed, the data will be left just as it was saved, and a record of this transaction will be written to the transaction log.

Transactions are a major part of coding a system, and shouldn't be shied away from because of any perceived performance issues. Use them as often as necessary to protect the data. I discuss transactions more in Chapter 13.