What Is a Transaction?

A transaction is one or more SQL statements that must be completed as a whole, or in other words, as a single Logical Unit of Work (LUW). Transactions provide a way of collecting and associating multiple actions into a single all-or-nothing multiple operation action. All operations within the transaction must be fully completed or not performed at all.

Consider a bank transaction in which you move $1,000 from your checking to your savings account. This transaction is, in fact, two operations: a decrement of your checking account and an increment of your savings account. Consider the impact on your finances if the bank's server went down after completing the first stage and never got to the second! By collecting the two operations together, as a transaction, they either both succeed or both fail as a single, complete unit of work.

A transaction is a logical unit of work that has four special characteristics, known as the ACID properties:

- Atomicity—Associated modifications are an all-or-nothing proposition; either all are done or none are done.
- Consistency—After a transaction finishes, all data is in the state it should be, all internal structures are correct, and everything accurately reflects the transaction that has occurred.
- Isolation—One transaction cannot interfere with the processes of another transaction.
- Durability—After the transaction has finished, all changes made are permanent.

The responsibility for enforcing the ACID properties of a transaction is split between T-SQL developers and SQL Server. The developer is responsible for ensuring that the modifications are correctly collected together and that the data is going to be left in a consistent state that corresponds with the actions being taken. SQL Server ensures that the transaction is isolated and durable, undertakes the atomicity requested, and ensures the consistency of the final data structures. The transaction log of each database provides the durability for the transaction. As you will see in this chapter, you have some control over how SQL Server handles some of these properties. For example, you can modify a transaction's isolation by enlisting bound connections.

How SQL Server Manages Transactions

SQL Server uses the database's transaction log to record the modifications occurring within the database. Each log record is labeled with a unique log sequence number (LSN), and all log entries that are part of the same transaction are linked together so they can be easily located if the transaction needs to be undone or redone. The primary responsibility of logging is to ensure transaction durability—either ensuring that the completed changes make it to the physical database files, or ensuring that any unfinished transactions are rolled back should there be an error or a server failure.

What is logged? Obviously, the start and end of a transaction are logged, but also the actual data modification, page allocations and deallocations, and changes to indexes. SQL Server keeps track of a number of pieces of information, all with the aim of ensuring the ACID properties of the transaction.

After a transaction has been committed, it cannot be rolled back. The only way to undo a committed transaction is to write another transaction to reverse the changes made. Before a transaction is committed, it can be rolled back.

SQL Server provides transaction management for all users using the following components:

- Transaction-control statements to define the logical units of work
- A write-ahead transaction log
- An automatic recovery process
- Data-locking mechanisms to ensure consistency and transaction isolation

Defining Transactions

You can carry out transaction processing with Microsoft SQL Server in three ways:

- AutoCommit—Every Transact-SQL statement is its own transaction and automatically commits when it finishes. This is the default mode in which SQL Server operates.
- Explicit—This approach provides programmatic control of the transaction using the `BEGIN TRAN` and `COMMIT`/`ROLLBACK TRAN`/`WORK` commands.
- Implicit—SQL Server is placed into a mode of operation in which issuing certain SQL commands automatically starts a transaction. The developer must finish the transaction by explicitly issuing the `COMMIT`/`ROLLBACK TRAN`/`WORK` commands.

Each of these methods is discussed in the following sections.

NOTE

The terms for explicit and implicit transactions can be somewhat confusing as to which is which. The way to keep them straight is to think of how a multistatement transaction is initiated, not how it is completed. AutoCommit transactions are in a separate category because they are both implicitly started and committed. Implicit and explicit transactions have to be explicitly ended, but explicit transactions must also be explicitly started with the `BEGIN TRAN` statement, whereas no `BEGIN TRAN` is necessary to start a multistatement transaction when in implicit transaction mode.

AutoCommit Transactions

AutoCommit transactions are the default transaction mode for SQL Server. Each individual Transact-SQL command automatically commits or rolls back its work at the end of its execution. Each SQL statement is considered to be its own transaction, with begin and end control points implied.

```
[implied begin transaction]
UPDATE account
SET balance = balance + 1000
WHERE account_no = "123456789"
[implied commit or rollback transaction]
```

If an error is present within the execution of the statement, the action is undone (rolled back); if no errors occurred, the action is completed and the changes are saved.

Now consider the banking transaction again and write the T-SQL statements to move money from the savings account to the checking account. Assume it was written as follows:

```
declare @checking_account char(10),
        @savings_account char(10)
select @checking_account = '0003456321',
       @savings_account = '0003456322'
update account
   set balance = balance - $1000
   where account_number = @checking_account
update savings_account
   set balance = balance + $1000
   where account_number = @savings_account
```

What would happen if an error occurred updating the savings account? With AutoCommit, each statement is implicitly committed after it completes successfully, so the update for the checking account has already been committed. You would have no way of rolling it back except to write another separate update to add the $1,000 back to the account. If the system crashed during the updates, how would you know which if any completed, and whether you would need to undo any of the changes because the subsequent commands were not executed? You would need some way to group the two commands together as a single logical unit of work so they complete or fail as a whole. SQL Server provides transaction control statements that allow you to create multi statement user-defined transactions.

To have complete control of a transaction and define logical units of work that consist of multiple data modifications, you need to write explicit user-defined transactions. Any SQL Server user can make use of the transaction control statements; no special privileges are required.

To start a multistatement transaction, use the `BEGIN TRAN` command, which optionally takes a transaction name:

```
BEGIN TRAN[SACTION] [transaction_name [WITH MARK ['description']]]
```

This name is essentially meaningless as far as transaction management is concerned, and if transactions are nested (which will be discussed later in this chapter), the name is only useful for the outermost `BEGIN TRAN` statement. Rolling back to any other name, besides a savepoint name, will generate an error and not roll back the transaction. The statements can only be rolled back when the outermost transaction is rolled back.

Naming transactions is really only useful when using the `WITH MARK` option. If the `WITH MARK` option is specified, a transaction name must be specified. `WITH MARK` allows for restoring a transaction log backup to a named mark in the transaction log.

This option allows you to restore a database to a known state, or to recover a set of related databases to a consistent state. However, be aware that `BEGIN TRAN` records are only written to the log if an actual data modification occurs within the transaction.

A transaction is completed successfully by issuing either a `COMMIT TRAN` or `COMMIT [WORK]` statement, or can be undone using either `ROLLBACK TRAN` or `ROLLBACK [WORK]`. The syntax of these commands is as follows:

```
COMMIT [TRAN[SACTION] [transaction_name]] | [WORK]

ROLLBACK [TRAN[SACTION] [transaction_name | savepointname]] | [WORK]
```

The `COMMIT` statement marks the successful conclusion of the transaction. This statement can be coded as `COMMIT`, `COMMIT WORK`, or `COMMIT TRAN`. It makes no difference—other than that the first two versions are SQL-92 ANSI compliant.

The `ROLLBACK` statement unconditionally undoes all work done within the transaction. This statement can also be coded as `ROLLBACK`, `ROLLBACK WORK`, or `ROLLBACK TRAN`. The first two commands are ANSI-92 SQL compliant and do not accept user-defined transaction names. `ROLLBACK TRAN` is required if you want to roll back to a savepoint within a transaction.

There are certain commands that cannot be specified within a user-defined transaction, primarily because they cannot be effectively rolled back in the event of a failure. In most cases, because of their long-running nature, you would not want them to be specified within a transaction anyway. Here are the commands you cannot specify in a user-defined transaction:

`ALTER DATABASE`	`CREATE DATABASE`	`DROP DATABASE`
`BACKUP DATABASE`	`RESTORE DATABASE`	`RECONFIGURE`
`BACKUP LOG`	`RESTORE LOG`	`UPDATE STATISTICS`

Savepoints

Savepoints allow you to set a marker in a transaction that you can roll back to undo a portion of the transaction, but commit the remainder of the transaction. The syntax is as follows:

```
SAVE TRAN[SACTION] savepointname
```

Example:-

```
BEGIN TRAN mywork
   UPDATE table1...
   SAVE TRAN savepoint1
      INSERT INTO table2...
      DELETE table3...
      IF @@error = -1
         ROLLBACK TRAN savepoint1
COMMIT TRAN
```

Nested Transactions

By rule, you can't have more than one active transaction per user session within SQL Server. However, consider that you have a SQL batch that issues a `BEGIN TRAN` statement, and then subsequently invokes a stored procedure, which also issues a `BEGIN TRAN` statement. Because you can only have one transaction active, what does the `BEGIN TRAN` inside the stored procedure accomplish? In SQL Server, this leads to an interesting anomaly referred to as nested transactions.

To determine whether transactions are open and how deep they are nested within a connection, you can use the global function called `@@trancount`.

If no transaction is active, the transaction nesting level is `0`. As a transaction is initiated, the transaction nesting level is incremented; as a transaction completes, the transaction nesting is decremented. The overall transaction remains open and can be entirely rolled back until the transaction nesting level returns to `0`.

You can use the `@@trancount` to monitor the current status of a transaction. For example, what would SQL Server do when encountering the following transaction (which produces an error because of the reference constraint on the `titles` table)?

```
BEGIN TRAN
   DELETE FROM publishers
   WHERE pub_id = '0736'
```

Is the transaction still active? You can find out using the `@@trancount` function:

```
select @@trancount
go

-----------
          1
```

In this case, `@@trancount` returns a value of `1`, which indicates that the transaction is still open and in progress. This means that you can still issue commands within the transaction and commit the changes, or roll back the transaction. Also, if you were to log out of the user session from SQL Server before the transaction nesting level reached `0`, SQL Server would automatically roll back the transaction.

Although nothing can prevent you from coding a `BEGIN TRAN` within another `BEGIN TRAN`, doing so has no real benefit even though such cases might occur. However, if you nest transactions in this manner, you must execute a `COMMIT` statement for each `BEGIN TRAN` statement issued. This is because SQL Server modifies the `@@trancount` with each transaction statement and considers the transaction finished only when the transaction nesting level returns to `0`.

Transaction Statements' Effects on `@@trancount`

Statement	**Effect on `@@trancount`**
`BEGIN TRAN`	+1
`COMMIT`	-1
`ROLLBACK`	Sets to 0
`SAVE TRAN savepoint`	No effect
`ROLLBACK TRAN savepoint`	No effect

Following is a summary of how transactional control relates to the values reported by `@@trancount`:

- When you log in to SQL Server, the value of `@@trancount` for your session is initially `0`.

- Each time you execute `begin transaction`, SQL Server increments `@@trancount`.
- Each time you execute `commit transaction`, SQL Server decrements `@@trancount`.
- Actual work is committed only when `@@trancount` reaches `0` again.
- When you execute `rollback transaction`, the transaction is canceled and `@@trancount` returns to `0`. Notice that `rollback transaction` cuts straight through any number of nested transactions, canceling the overall main transaction. This means that you need to be careful how you write code that contains a `rollback` statement. Be sure to return up through all levels so you don't continue executing data modifications that were meant to be part of the larger overall transaction.
- Setting savepoints and rolling back to a savepoint do not affect `@@trancount` or transaction nesting in any way.
- If a user connection is lost for any reason when `@@trancount` is greater than `0`, any pending work for that connection is automatically rolled back. SQL Server requires that multistatement transactions be explicitly committed.
- Because the `BEGIN TRAN` statement increments `@@trancount`, each `BEGIN TRAN` statement must be paired with a `COMMIT` for the transaction to successfully complete.

Take a look at some sample code to show the values of `@@trancount` as the transaction progresses. This first example is a simple explicit transaction with a nested `BEGIN TRAN`:

SQL Statement	**`@@trancount` Value**
`SELECT "Starting....."`	`0`
`BEGIN TRAN`	`1`
`   DELETE FROM table1`	`1`
`   BEGIN TRAN`	`2`
`      INSERT INTO table2`	`2`
`   COMMIT`	`1`
`   UPDATE table3`	`1`
`COMMIT`	`0`

Nested transactions are syntactic only. The only `commit tran` statement that has an impact on real data is the last one, the statement returning `@@trancount` to `0`. That statement fully commits the work done by the initial transaction and the nested transactions. Until that final `COMMIT TRAN` is encountered, all of the work can be rolled back with a `ROLLBACK` statement.

As a general rule, if a transaction is already active, you shouldn't issue another `BEGIN TRAN` statement. Check the value of `@@trancount` to determine if a transaction is already active. If you want to be able to roll back the work performed within a nested transaction without rolling back the entire transaction, set a savepoint instead of issuing a `BEGIN TRAN` statement. Later in this chapter, you will see an example of how to check `@@trancount` within a stored procedure to determine if the stored procedure is being invoked within a transaction and issuing a `BEGIN TRAN` or `SAVE TRAN` as appropriate.

Implicit Transactions

AutoCommit transactions and explicit user-defined transactions in SQL Server are not ANSI-92 SQL compliant. The ANSI-92 SQL standard states that any data retrieval or modification statement issued should implicitly begin a multistatement transaction that remains in effect until an explicit `ROLLBACK` or `COMMIT` statement is issued.

To enable implicit transactions for a connection, you need to turn on the `IMPLICIT_TRANSACTIONS` session setting. The syntax is as follows:

```
SET IMPLICIT_TRANSACTIONS {ON | OFF}
```

After this option is turned on, transactions will be implicitly started, if not already in progress, whenever any of the following commands are executed:

ALTER TABLE	CREATE	DELETE
DROP	FETCH	GRANT
INSERT	OPEN	REVOKE
SELECT	TRUNCATE TABLE	UPDATE

Note that neither the `ALTER VIEW` nor `ALTER PROCEDURE` statement starts an implicit transaction.

Implicit transactions must be explicitly completed by issuing a `COMMIT` or `ROLLBACK`, and a new transaction is started again on the execution of any of the preceding commands. If you plan to use implicit transactions, the main thing to be aware of is that locks are held until you explicitly commit the transaction. This can cause problems with concurrency and the ability of the system to back up the transaction log.

Even when using implicit transactions, you can still issue the `BEGIN TRAN` statement and create transaction nesting. In this next example, `IMPLICIT TRANSACTIONS ON` has been turned on to see the effect this has on the value of `@@trancount`.

SQL Statements	**`@@trancount` Value**
SET IMPLICIT_TRANSACTIONS ON	0
go	0
INSERT INTO table1	1

SQL Statements	`@@trancount` Value
`UPDATE table2`	1
`COMMIT`	0
`go`	
`BEGIN TRAN`	2
`DELETE FROM table1`	2
`COMMIT`	1
`go`	
`DROP TABLE table1`	1
`COMMIT`	0

As you can see in this second example, if a `BEGIN TRAN` is issued while a transaction is still active, transaction nesting will occur and a second `COMMIT` is required to finish the transaction. The main difference is that a `BEGIN TRAN` was not required to start the transaction. The first `INSERT` statement initiated the transaction. When you are running in implicit transaction mode, you don't need to issue a `BEGIN TRAN` statement; in fact, you should avoid it to prevent transaction nesting and the need for multiple commits.

Implicit Transactions Vs Explicit Transactions

When would you want to use implicit transactions versus explicit transactions? If you are porting an application from another database environment that used an implicit transaction, that application will port over more easily with fewer code changes if you run in implicit transaction mode. Also, if the application you are developing needs to be ANSI-compliant and run across multiple database platforms with minimal code changes, you might want to use implicit transactions.

If you use implicit transactions in your applications, just be sure to issue `COMMIT` statements as frequently as possible to prevent leaving transactions open and holding locks for an extended period of time, which can have an adverse impact on concurrency and overall system performance.

If your application is only going to be hosted on SQL Server, it is recommended that you use AutoCommit and explicit transactions so that changes are committed as quickly as possible and only those logical units of work that are explicitly defined will contain multiple commands within a transaction.

Transactions and Locking

SQL Server issues and holds onto locks for the duration of a transaction to ensure the isolation and consistency of the modifications. Data modifications that occur within a transaction will acquire exclusive locks, which are then held until the completion of the transaction. Shared locks, or read locks, are held for only as long as the statement needs them; usually, a shared lock is released as soon as data has been read from the resource (row, page, table). The length of time a shared lock is held can be modified by the use of keywords such as `HOLDLOCK` in a query. If this option is specified, shared locks are held onto until the completion of the transaction.

What this means for database application developers is that you should try to hold onto as few locks or as small a lock as possible for as short a time as possible to avoid locking contention between applications and to improve concurrency and application performance. The simple rule when working with transactions is "Keep them short and keep them simple!" In other words, do what you need to do in the most concise manner in the shortest possible time. Keep any extraneous commands that do not need to be part of the logical unit of work—such as select statements, dropping temp tables, setting up local variables, and so on—outside of the transaction.

To modify the manner in which a transaction and its locks can be handled by a `SELECT` statement, you can issue the `SET TRANSACTION ISOLATION LEVEL` statement. This statement allows the query to choose how much it is protected against other transactions modifying the data being used. The `SET TRANSACTION ISOLATION LEVEL` statement has the following mutually exclusive options:

- `READ COMMITTED`— This setting is the default for SQL Server. Modifications made within a transaction are locked exclusively, and the changes cannot be viewed by other user processes until the transaction completes. Commands that read data only hold shared locks on the data for as long as it is reading it. Because other transactions are not blocked from modifying the data after you have read it within your transaction, subsequent reads of the data within the transaction might encounter non-repeatable reads or phantom data.
- `READ UNCOMMITTED`— With this level of isolation, one transaction can read the modifications made by other transactions prior to being committed. This is, therefore, the least restrictive isolation level, but one that allows the reading of dirty and uncommitted data. This option has the same effect as issuing `NOLOCK` within your `SELECT` statements, but it only has to be set once for your connection. This should never be used in an application in which accuracy of the query results is required.
- `REPEATABLE READ`— When this option is set, as data is read, locks are placed and held on the data for the duration of the transaction. These locks prevent other transactions from modifying the data you have read, so that you can carry out multiple passes across the same information and get the same results each time. This isolation level is obviously more restrictive and can block other transactions.

However, although it prevents non-repeatable reads, it does not prevent the addition of new rows or phantom rows because only existing data is locked.

- `SERIALIZABLE`— This option is the most restrictive isolation level because it places a range lock on the data. This prevents any modifications to the data being read from until the end of the transaction. It also avoids phantom reads by preventing rows from being added or removed from the data range set.

Transaction Isolation Levels in SQL Server

Isolation levels determine the proportion to which data being accessed or modified in one transaction is protected from changes to the data by other transactions. In theory, each transaction should be fully isolated from other transactions. However, in practice, for practical and performance reasons, this might not always be the case. In a concurrent environment in the absence of locking and isolation, the following four scenarios can happen:

- Lost update—In this scenario, no isolation is provided to a transaction from other transactions. Multiple transactions can read the same copy of data and modify it. The last transaction to modify the dataset prevails, and the changes by all other transactions are lost.
- Dirty reads—In this scenario, one transaction can read data that is being modified by other transactions. Data read by the first transaction is inconsistent because the other transaction might choose to roll back the changes.
- Nonrepeatable reads—This is somewhat similar to zero isolation. In this scenario, a transaction reads the data twice, but before the second read occurs, another transaction modifies the data; therefore, the values read by the first read will be different from those of the second read. Because the reads are not guaranteed to be repeatable each time, this scenario is called nonrepeatable reads.
- Phantom reads—This scenario is similar to nonrepeatable reads. However, instead of the actual rows that were read changing before the transaction is complete, additional rows are added to the table, resulting in a different set of rows being read the second time. Consider a scenario where Transaction A reads rows with key values within the range of 1–5 and returns three rows with key values 1, 3, and 5. Before Transaction A reads the data again within the transaction, Transaction B adds two more rows with the key values 2 and 4 and commits the changes. Assuming that Transaction A and Transaction B both can run independently without blocking each other, when Transaction A runs the query a second time, it is now going to get 5 rows with key values 1, 2, 3, 4, and 5. This phenomenon is called phantom reads because in the second pass, you are getting records you did not expect to retrieve.

Ideally, a DBMS must provide levels of isolation to prevent these types of scenarios. Sometimes, because of practical and performance reasons, databases do relax some of the rules. ANSI has defined four transaction isolation levels, each providing a different degree of isolation to cover the previous scenarios. ANSI SQL-92 defines the following four standards for transaction isolation:

- Read Uncommitted (Level 0)
- Read Committed (Level 1)
- Repeatable Read (Level 2)
- Serializable (Level 3)

SQL Server does support all these levels. Each higher level incorporates the isolation provided at the lower levels. You can set these isolation levels for your entire session by using the `SET TRANSACTION ISOLATION LEVEL` T-SQL command, or for individual `SELECT` statements by specifying the isolation level hints within the query. Using table-level hints will be covered later in this chapter in the "Table Hints for Locking" section.

Read Uncommitted

If you set the Read Uncommitted mode for a session, no isolation is provided to the `SELECT` queries in that session. A transaction that is running with this isolation level is not immune to dirty reads, nonrepeatable reads, or phantom reads.

To set the Read Uncommitted mode for a session, run the following statements from the client:

- T-SQL—`SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED`.
- ODBC—Use the function call `SQLSetConnectAttr` with `Attribute` set to `SQL_ATTR_TXN_ISOLATION` and `ValuePtr` set to `SQL_TXN_READ_UNCOMMITTED`.
- OLE DB—Use the function call `ITransactionLocal::StartTransaction` with the `isoLevel` set to `ISOLATIONLEVEL_READUNCOMMITTED`.
- ADO—Set the `IsolationLevel` property of the Connection object to `adXactReadUncommitted`.

Be careful when running queries at Read Uncommitted isolation; it is possible to read changes that have been made to data that are subsequently rolled back. In essence, the accuracy of the results cannot be guaranteed. You should only use this mode when you need to get information quickly from an OLTP database without impacting or being impacted by the ongoing updates, and when the accuracy of the results is not critical.

Read Committed

The Read Committed mode is the default locking-isolation mode for SQL Server. With Read Committed as the transaction isolation level, read operations can only read pages for transactions that have already been committed. No "dirty reads" are allowed. Locks acquired by update transactions are held for the duration of the transaction. However, in this mode, read requests release locks as soon as the query finishes reading the data. Although this improves concurrent access to the data for updates, it does not prevent nonrepeatable reads or phantom reads. For example, within a transaction, a process could read one set of rows early in the transaction, and then before reading the information again, another process could modify the resultset, resulting in a different resultset being read the second time.

Because Read Committed is the default isolation level for SQL Server, you do not need to do anything to set this mode. If you need to set the isolation level back to Read Committed mode for a session, run the following statements from the client:

- T-SQL—`SET TRANSACTION ISOLATION LEVEL READ COMMITTED`.
- ODBC—Use the function call `SQLSetConnectAttr` with `Attribute` set to `SQL_ATTR_TXN_ISOLATION` and `ValuePtr` set to `SQL_TXN_READ_COMMITTED`.
- OLE DB—Use the function call `ITransactionLocal::StartTransaction` with `isoLevel` set to `ISOLATIONLEVEL_READCOMMITTED`.
- ADO—Set the `IsolationLevel` property of the `Connection` object to `adXactReadcommitted`.

Repeatable Read

In Repeatable Read mode, SQL Server provides the same level of isolation for updates as in Read Committed mode, but it also allows the data to be read many times within the same transaction and guarantees that the same values will be read each time. Repeatable Read isolation mode prevents other users from updating data that has been read within the transaction until the transaction in which it was read is committed or rolled back. This way, the reading transaction will not pick up changes to the rows it read previously within the transaction. However, this isolation mode does not prevent additional rows (phantom reads) from appearing in the subsequent reads.

Although preventing nonrepeatable reads is desirable for certain transactions, it requires holding locks on the data that has been read until the transaction is completed. This reduces concurrent access for multiple update operations and causes performance degradation due to lock waits and locking contention between transactions. It can also potentially lead to deadlocks. (Deadlocking will be discussed in more detail in the "Deadlocks" section later in this chapter.)

To set Repeatable Read mode for a session, run the following statements from the client:

- T-SQL—`SET TRANSACTION ISOLATION LEVEL REPEATABLE READ`.
- ODBC—Use the function call `SQLSetConnectAttr` with `Attribute` set to `SQL_ATTR_TXN_ISOLATION` and `ValuePtr` set to `SQL_TXN_REPEATABLEREAD`.
- OLE DB—Use the function call `ITransactionLocal::StartTransaction` with `isoLevel` set to `ISOLATIONLEVEL_REPEATABLEREAD`.
- ADO—Set the `IsolationLevel` property of the `Connection` object to `adXact REPEATABLEREAD`.

Serializable

Serializable Read mode is similar to repeatable reads but adds to it the restriction that rows cannot be added to a resultset that was read previously within a transaction. This prevents phantom reads. In other words, Serializable Read locks the existing data being read as well as rows that do not yet exist. It accomplishes this by locking the data being read. In addition, SQL Server puts locks on the range of values being read so that additional rows cannot be added to the range.

For example, perhaps you run a query in a transaction that retrieves all records for the Sales table in the Pubs database for a store with the stor_id of 7066. To prevent additional sales records from being added to the sales table for this store, SQL Server locks the range of values with stor_id of 7066. It accomplishes this by using key-range locks, which will be discussed in the "Serialization and Key-Range Locking" section later in this chapter.

Although preventing phantom reads is desirable for certain transactions, Serializable mode, like Repeatable Read, reduces concurrent access for multiple update operations and can cause performance degradation due to lock waits and locking contention between transactions, and potentially lead to deadlocks.

To set the Serializable mode for a session, run the following statements from the client:

- T-SQL—`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`.
- ODBC—Use the function call `SQLSetConnectAttr` with `Attribute` set to `SQL_ATTR_TXN_ISOLATION` and `ValuePtr` set to `SQL_TXN_SERIALIZABLE`.
- OLE DB—Use the function call `ITransactionLocal::StartTransaction` with `isoLevel` set to `ISOLATIONLEVEL_SERIALIZABLE`.
- ADO—Set the `IsolationLevel` property of the `Connection` object to `adXact SERIALIZABLE`.

NOTE

Only one of the transaction isolation levels can be active at any given time for a user session. The isolation level you set within your application is active for the duration of the connection or until manually reset. To check the current transaction isolation level settings, run the following command:

```
DBCC USEROPTIONS
Go

Set Option                                  Value
----------------------------------          ------------
textsize                                    64512
language                                    us_english
dateformat                                  mdy
datefirst                                   7
quoted_identifier                           SET
ansi_null_dflt_on                           SET
ansi_defaults                               SET
ansi_warnings                               SET
ansi_padding                                SET
ansi_nulls                                  SET
concat_null_yields_null                     SET
isolation level                             serializable
```

SQL Server Lock Types

Locking is handled automatically within SQL Server. The Lock Manager chooses the type of locks based on the type of transaction (such as `select`, `insert`, `update`, and `delete`). The various types of locks used by Lock Manager are as follows:

- Shared
- Update
- Exclusive
- Intent
- Schema Locks
- Bulk Update Locks

As in version 7.0, the Lock Manager in SQL Server 2000 automatically adjusts the granularity of the locks (row, page, table, and so on) based on the nature of the statement that is executed and the number of rows that are affected.

Shared Locks

SQL Server uses shared locks for all read operations. A shared lock is, by definition, not exclusive. Theoretically, an unlimited number of shared locks can be held on a resource at any given time. In addition, shared locks are unique in that, by default, a process locks the resource only for the duration of the read on that page. For example, a query such as `select * from authors` would lock the first page in the `authors` table when the query starts. After data on the first page is read, the lock on that page is released, and a lock on the second page is acquired. After the second page is read, its lock is released and a lock on the third page is acquired, and so on. In this fashion, a `select` query allows other data pages that are not being read to be modified during the read operation. This increases concurrent access to the data.

Shared locks are compatible with other shared locks as well as with update locks. In this way, a shared lock does not prevent the acquisition of additional shared locks or an update lock by other processes on a given page. Multiple shared locks can be held at any given time for a number of transactions or processes. These transactions do not affect the consistency of the data. However, shared locks do prevent the acquisition of exclusive locks. Any transaction that is attempting to modify data on a page or a row on which a shared lock is placed will be blocked until all the shared locks are released.

NOTE

It is important to point out that within a transaction running at the default isolation level of Read Committed, shared locks are not held for the duration of the transaction, or even the duration of the statement that acquires the shared locks. Shared lock resources (row, page, table, and so on) are normally released as soon as the read operation on the resource is completed. SQL Server provides the `HOLDLOCK` clause to the `SELECT` statement if you want to continue holding the shared lock for the duration of the transaction. `HOLDLOCK` is explained later in this chapter in the section "Table Hints for Locking." Another way to hold shared locks for the duration of the transaction is to set the isolation level for the session or the query to repeatable reads or higher.

Update Locks

Update locks are used to lock pages that a user process would like to modify. When a transaction tries to update a row, it must first read the row to ensure that it is modifying the appropriate record. If the transaction were to put a shared lock on the resource initially, it would eventually need to get an exclusive lock on the resource to modify the record and prevent any other transaction from modifying the same record. The problem is that this could lead to deadlocks in an environment in which multiple transactions are trying to modify data on the same resource at the same time.

Update locks in SQL Server are provided to prevent this kind of deadlock scenario. Update locks are partially exclusive in that only one update lock can be acquired at a time on any resource. However, an update lock is compatible with shared locks, in that both can be acquired on the same resource simultaneously. In effect, an update lock signifies that a process wants to change a record, and keeps out other processes that also want to change that record. However, an update lock does allow other processes to acquire shared locks to read the data until the update or delete statement is finished locating the records to be affected. The process then attempts to escalate each update lock to an exclusive lock. At this time, the process will wait until all currently held shared locks on the same records are released. After the shared locks are released, the update lock is escalated to an exclusive lock. The data change is then carried out and the exclusive lock is held for the remainder of the transaction.

NOTE

Update locks are not used just for update operations. SQL Server uses update locks any time that a search for the data is required prior to performing the actual modification, such as qualified updates and deletes (that is, when a `where` clause is specified). Update locks are also used for inserts into a table with a clustered index because SQL Server must first search the data and the clustered index to identify the correct position at which to insert the new row to maintain the sort order. After SQL Server has found the correct location and begins inserting the record, it escalates the update lock to an exclusive lock.

Exclusive Locks

As mentioned earlier, an exclusive lock is granted to a transaction when it is ready to perform data modification. An exclusive lock on a resource makes sure that no other transaction can interfere with the data locked by the transaction that is holding the exclusive lock. SQL Server releases the exclusive lock at the end of the transaction.

Exclusive locks are incompatible with any other lock type. If an exclusive lock is held on a resource, any other read or data modification requests for the same resource by other processes will be forced to wait until the exclusive lock is released. Likewise, if a resource currently has read locks held on it by other processes, the exclusive lock request is forced to wait in a queue for the resource to become available.

Intent Locks

Intent locks are not really a locking mode, but a mechanism to indicate at a higher level of granularity the type of locks held at a lower level. The types of intent locks mirror the lock types previously discussed: shared intent locks, exclusive intent locks, and update intent locks. SQL Server Lock Manager uses intent locks as a mechanism to indicate that a shared, update, or exclusive lock is held at a lower level. For example, a shared intent lock on a table by a process signifies that the process currently holds a shared lock on a row or page within the table. The presence of the intent lock prevents other transactions from attempting to acquire a lock on the table.

Intent locks improve locking performance by allowing SQL Server to examine locks at the table level to determine the types of locks held on the table, rather than searching through the multiple locks at the page or row level within the table. Intent locks also prevent two transactions that are both holding locks at a lower level on a resource from attempting to escalate those locks to a higher level while the other transaction still holds the intent lock. This prevents deadlocks during lock escalation.

There are three types of intent locks that you will typically see when monitoring locking activity: intent shared (IS) locks, intent exclusive (IX) locks, and shared with intent exclusive (SIX) locks. The IS lock indicates that the process currently holds, or has the intention of holding, shared locks on lower-level resources (row or page). The IX lock indicates that the process currently holds, or has the intention of holding, exclusive locks on lower-level resources. The SIX (pronounced as the letters S-I-X, not like the number six) lock occurs under special circumstances when a transaction is holding a shared lock on a resource, and later in the transaction, an IX lock is needed. At that point, the IS lock is converted to an SIX lock.

In the following example, the `SELECT` statement run at the serializable level acquires a shared table lock. It then needs an exclusive lock to update the row in the `sales_big` table.

```
SET TRANSACTION ISOLATION LEVEL serializable
go
BEGIN TRAN
 select sum(qty) FROM sales_big
UPDATE sales_big
    SET qty = 0
    WHERE sales_id = 1001
COMMIT TRAN
```

Because the transaction initially acquired a Shared (S) table lock and then needed an exclusive row lock, which requires an intent exclusive (IX) lock on the table within the same transaction, the S lock is converted to an SIX lock.

NOTE

If only a few rows were in `sales_big`, SQL Server might only acquire individual row or key locks rather than a table-level lock. SQL Server would then have an intent shared (IS) lock on the table rather than a full shared (S) lock. In that instance, the `UPDATE` statement would then acquire a single exclusive lock to apply the update to a single row, and the X lock at the key level would result in the IS locks at the page and table level being converted to an IX lock at the page and table level for the remainder of the transaction.

Schema Locks

SQL Server uses schema locks to maintain structural integrity of SQL Server tables. Unlike other types of locks that provide isolation for the data, schema locks provide isolation for the schema of database objects, such as tables, views, and indexes within a transaction. The Lock Manager uses two types of schema locks:

- Schema stability locks—When a transaction is referencing either an index or a data page, SQL Server places a schema stability lock on the object. This ensures that no other process can modify the schema of an object—such as dropping an index or dropping or altering a stored procedure or table—while other processes are still referencing the object.
- Schema modification locks—When a process needs to modify the structure of an object (alter the table, recompile a stored procedure, and so on), the Lock Manager places a schema modification lock on the object. For the duration of this lock, no other transaction can reference the object until the changes are complete and committed.

Bulk Update Locks

Bulk Update locks are a special type of lock used only when bulk copying data into a table using the `bcp` utility or the `BULK INSERT` command. This special lock is used for these operations only when either the `TABLOCK` hint is specified to `bcp` or the `BULK INSERT` command, or when the `table lock on bulk load` table option has been set for the table. Bulk Update locks allow multiple bulk copy processes to bulk copy data into the same table in parallel, while preventing other processes that are not bulk copying data from accessing the table.

Table Hints for Locking

As mentioned previously in this chapter in the "Transaction Isolation Levels in SQL Server" section, you can set an isolation level for your connection using the `SET TRANSACTION ISOLATION LEVEL` command. This command sets a global isolation level for your entire session, which is useful if you want to provide a consistent isolation level for your application. However, sometimes you will want to specify different isolation levels for different queries in the system, or different isolation levels for different tables within a single query. SQL Server allows you to do this by supporting table hints in the `FROM` clause of `SELECT`, `UPDATE`, and `DELETE` statements. This allows you to override the isolation level that is currently set at the session level.

In this chapter, you have seen that locking is dynamic and automatic in SQL Server. Based on certain factors (such as SARGs, key distribution, data volume, and so on), the query optimizer chooses the granularity of the lock (row, page, or table level) on a resource. Although it is usually best to leave such decisions to the cost-based optimizer, you might encounter certain situations in which you want to force a different lock granularity on a resource than what the optimizer has chosen. SQL Server provides additional table hints that you can use in the query to force lock granularity for various tables that are participating in a join.

SQL Server also automatically determines the lock type (`SHARED`, `UPDATE`, `EXCLUSIVE`) to use on a resource depending on the type of command being executed on the resource. For example, a `SELECT` statement will use a shared lock. SQL Server 2000 also provides additional table hints to override the default lock type.

The table hints to override the lock isolation, granularity, or lock type for a table can be provided in the `FROM` clause of the query by using the `WITH` operator. The following is the syntax of the `FROM` clause when using table hints:

```
FROM table_name [ [AS] table_alias ] WITH ( table_hint [ ,...n ] ) [,
...n]
```

The following sections discuss the various locking hints that can be passed to an optimizer to manage isolation levels and the lock granularity of a query.

NOTE

Although many of the table-locking hints can be combined, you cannot combine more than one hint at a time on a table from the granularity and isolation level hints. Also, the `NOLOCK`, `READUNCOMMITTED`, and `READPAST` hints described in the following sections cannot be used on tables that are the target of `INSERT`, `UPDATE`, or `DELETE` queries.

Transaction Isolation-Level Hints

SQL Server provides a number of hints that you can use in a query to override the default transaction isolation level. These hints are described as follows:

- `HOLDLOCK`— Within a transaction, a shared lock on a resource (row, page, table) is released as soon as the T-SQL statement that is holding the shared lock is finished with the resource. To maintain a shared lock for the duration of the entire statement, or for the entire transaction if the statement is in a transaction, use the `HOLDLOCK` clause in the statement. The following example demonstrates the usage of the `HOLDLOCK` statement within a transaction:

```
declare @seqno int
begin transaction
-- get a UNIQUE sequence number from sequence table
SELECT @seqno = isnull(seq#,0) + 1
from sequence WITH (HOLDLOCK)
     -- in the absence of HOLDLOCK, shared lock will be released
     -- and if some other concurrent transaction ran the same
     -- command, both of them could get the same sequence number
UPDATE sequence
set    seq# = @seqno
     --now go do something else with this unique sequence number
commit tran
```

NOTE

As discussed earlier in this chapter in the "Deadlocks" section, using `HOLDLOCK` in this manner leads to potential deadlocks between processes that are executing this transaction at the same time. For this reason, the `HOLDLOCK` hint, as well as the `REPEATABLEREAD` and `SERIALIZABLE` hints, should be used sparingly if at all. In this example, it might be better for the `SELECT` statement to use an update or exclusive lock on the `sequence` table using the hints discussed later in the section in the "Lock Type Hints" section. Another option would be to use an application lock as discussed previously in this chapter in the "Using Application Locks" section.

- `NOLOCK`— You can use this option to specify that no shared lock be placed on the resource and that requests for update or exclusive locks be denied. This option is similar to running a query at isolation level 0 (`READUNCOMMITTED`), which allows the query to ignore exclusive locks and read uncommitted changes. The `NOLOCK` option is a useful feature in reporting environments, where the accuracy of the results is not critical.
- `READUNCOMMITTED`— This is the same as specifying the Read Uncommitted mode when using the `SET TRANSACTION ISOLATION LEVEL` command, and it is the same as the `NOLOCK` table hint.
- `READCOMMITTED`— This is the same as specifying the Read Committed mode when you use the `SET TRANSACTION ISOLATION LEVEL` command. The query will wait for exclusive locks to be released before reading the data. This is the default locking isolation mode for SQL Server.
- `REPEATABLEREAD`— This is the same as specifying Repeatable Read mode with the `SET TRANSACTION ISOLATION LEVEL` command. It prevents nonrepeatable reads within a transaction, and behaves similarly to using the `HOLDLOCK` hint.
- `SERIALIZABLE`— This is the same as specifying Serializable mode with the `SET TRANSACTION ISOLATION LEVEL` command. It prevents phantom reads within a transaction, and behaves similarly to using the `HOLDLOCK` hint.
- `READPAST`— This hint applies only to the `SELECT` statement. By specifying this option, you can skip over the rows that are locked by other transactions, returning the rows that can be read. In the absence of the `READPAST` option, the `SELECT` statement will wait (or time out if lock timeout values are set) until the locks are released on the rows by other transactions. A statement that uses the `READPAST` clause can only read past row locks that are held by transactions running in Read Committed mode. This lock hint is useful when reading information from a SQL Server table used as a work queue.

Lock Granularity Hints

You can use the following optimizer hints to override lock granularity:

- `ROWLOCK`— You can use this option to force the Lock Manager to place a row-level lock on a resource instead of a page-level or a table-level lock. This can be used in conjunction with the `XLOCK` lock type hint to force exclusive row locks.
- `PAGLOCK`— You can use this option to force a page-level lock on a resource instead of a row-level or table-level lock. This can be used in conjunction with the `XLOCK` lock type hint to force exclusive page locks.
- `TABLOCK`— You can use this option to force a table-level lock instead of a row-level or a page-level lock. This option can be used in conjunction with the `HOLDLOCK` table hint to hold the table lock until the end of the transaction.
- `TABLOCKX`— You can use this option to force a table-level exclusive lock instead of a row-level or a page-level lock. No shared or update locks are granted to other transactions as long as this option is in effect. If you are planning maintenance on a SQL Server table and you don't want interference from other transactions, this is one of the ways to essentially put a table into a single user mode.

Lock Type Hints

You can use the following optimizer hints to override the lock type that SQL Server uses:

- `UPDLOCK`— This option is similar to `HOLDLOCK` except that whereas `HOLDLOCK` uses a shared lock on the resource, `UPDLOCK` places an update lock on the resource for the duration of the transaction. This allows other processes to read the information, but not acquire update or exclusive locks on the resource. This option provides read repeatability within the transaction while preventing deadlocks that can result when using `HOLDLOCK`.
- `XLOCK`— This option is similar to `HOLDLOCK` except that whereas `HOLDLOCK` uses a shared lock on the resource, `XLOCK` places an exclusive lock on the resource for the duration of the transaction. This prevents other processes from acquiring locks on the resource.

An Overview of SQL Server Security

SQL Server security is built on a two-tier model. The first tier is access to SQL Server, which involves the person attempting to connect being authenticated as a valid SQL Server account, or login as it is known. Think of a login as being similar to entering an office tower and signing in with the security guard. The guard verifies that you have business in the building, and you head for the elevators. The second tier involves access to the databases. As SQL Server supports multiple databases, each database has its own security layer that provides access to that database through accounts known as users. These users are then mapped to the server logins to provide access. As users are created on a database-by-database basis, access can be restricted to one or many databases as needed. If you go back to the office building example, this would be like having an access card for the elevator that only allows you to get off at certain floors.

The key points to remember in SQL Server security are that logins are server-wide and give access to SQL Server, while users are database-specific and give access only to the database in which they are created. By mapping logins to users, a connection is made to SQL Server, and access is allowed to the database.