	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	53	6	0	DATABASE		S	GRANT
2	53	6	72057594077577216	RID	1:121321:0	X	GRANT
3	53	6	72057594077577216	PAGE	1:121321	IX	GRANT
4	53	6	1940201962	OBJECT		IX	GRANT

Figure 21-1. *Output from sys.dm_tran_locks showing the row-level lock granted to the DELETE statement*

Note I explain lock modes later in the chapter in the "Lock Modes" section.

Granting an `RID` lock to the `DELETE` statement prevents other transactions from accessing the row.

The resource locked by the `RID` lock can be represented in the following format from the `resource_description` column:

```
FileID:PageID:Slot(row)
```

In the output from the query against `sys.dm_tran_locks` in Figure 21-1, the `DatabaseID` is displayed separately under the `resource_database_id` column. The `resource_description` column value for the `RID` type represents the remaining part of the `RID` resource as `1:121321:0`. In this case, a `FileID` of `1` is the primary data file, a `PageID` of `121321` is a page belonging to the `dbo.Test1` table identified by the `C1` column, and a `Slot (row)` of `0` represents the row position within the page. You can obtain the table name and the database name by executing the following SQL statements:

```
SELECT OBJECT_NAME(1940201962),
       DB_NAME(6);
```

The row-level lock provides very high concurrency since blocking is restricted to the row under effect.

Key-Level Lock

This is a row lock within an index, and it is identified as a `KEY` lock. As you know, for a table with a clustered index, the data pages of the table and the leaf pages of the clustered index are the same. Since both of the rows are the same for a table with a clustered index, only a `KEY` lock is acquired on the clustered index row, or limited range

of rows, while accessing the rows from the table (or the clustered index). For example, consider having a clustered index on the `Test1` table.

```
CREATE CLUSTERED INDEX TestIndex ON dbo.Test1(C1);
```

Next, rerun the following code:

```
BEGIN TRAN
DELETE  dbo.Test1
WHERE   C1 = 1 ;

SELECT  dtl.request_session_id,
        dtl.resource_database_id,
        dtl.resource_associated_entity_id,
        dtl.resource_type,
        dtl.resource_description,
        dtl.request_mode,
        dtl.request_status
FROM    sys.dm_tran_locks AS dtl
WHERE   dtl.request_session_id = @@SPID ;
ROLLBACK
```

The corresponding output from `sys.dm_tran_locks` shows a `KEY` lock instead of the `RID` lock, as you can see in Figure 21-2.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	52	6	0	DATABASE		S	GRANT
2	52	6	72057594077904896	PAGE	1:34064	IX	GRANT
3	52	6	72057594077904896	KEY	(de42f79bc795)	X	GRANT
4	52	6	1940201962	OBJECT		IX	GRANT

Figure 21-2. *Output from sys.dm_tran_locks showing the key-level lock granted to the DELETE statement*

When you are querying `sys.dm_tran_locks`, you will be able to retrieve the database identifier, `resource_database_id`. You can also get information about what is being locked from `resource_associated_entity_id`; however, to get to the particular resource (in this case, the page on the key), you have to go to the `resource_ description` column for the value, which is `1:34064`. In this case, the `Index ID` of `1` is the clustered index on the `dbo.Test1` table. You also see the types of requests that are made: S, IX, X, and so on. I cover these in more detail in the upcoming "Lock Modes" section.

Note You'll learn about different values for the `IndId` column and how to determine the corresponding index name in this chapter's "Effect of Indexes on Locking" section.

Like the row-level lock, the key-level lock provides very high concurrency.

Page-Level Lock

A page-level lock is maintained on a single page within a table or an index, and it is identified as a `PAG` lock. When a query requests multiple rows within a page, the consistency of all the requested rows can be maintained by acquiring either `RID/KEY` locks on the individual rows or a `PAG` lock on the entire page. From the query plan, the lock manager determines the resource pressure of acquiring multiple `RID/KEY` locks, and if the pressure is found to be high, the lock manager requests a `PAG` lock instead.

The resource locked by the `PAG` lock may be represented in the following format in the `resource_description` column of `sys.dm_tran_locks`:

```
FileID:PageID
```

The page-level lock can increase the performance of an individual query by reducing its locking overhead, but it hurts the concurrency of the database by blocking access to all the rows in the page.

Extent-Level Lock

An extent-level lock is maintained on an extent (a group of eight contiguous data or index pages), and it is identified as an `EXT` lock. This lock is used, for example, when an `ALTER INDEX REBUILD` command is executed on a table and the pages of the table may be moved from an existing extent to a new extent. During this period, the integrity of the extents is protected using `EXT` locks.

Heap or B-tree Lock

A heap or B-tree lock is used to describe when a lock to either type of object could be made. The target object could be an unordered heap table, a table without a clustered index, or a B-tree object, usually referring to partitions. A setting within the `ALTER TABLE`

function allows you to exercise a level of control over how locking escalation (covered in the "Lock Escalation" section) is affected with the partitions. Because partitions can be across multiple filegroups, each one has to have its own data allocation definition. This is where the `HoBT` lock comes into play. It acts like a table-level lock but on a partition instead of on the table itself.

Table-Level Lock

This is the highest level of lock on a table, and it is identified as a `TAB` lock. A table-level lock on a table reserves access to the complete table and all its indexes.

When a query is executed, the lock manager automatically determines the locking overhead of acquiring multiple locks at the lower levels. If the resource pressure of acquiring locks at the row level or the page level is determined to be high, then the lock manager directly acquires a table-level lock for the query.

The resource locked by the `OBJECT` lock will be represented in `resource_description` in the following format:

```
ObjectID
```

A table-level lock requires the least overhead compared to the other locks and thus improves the performance of the individual query. On the other hand, since the table-level lock blocks all write requests on the entire table (including indexes), it can significantly hurt database concurrency.

Sometimes an application feature may benefit from using a specific lock level for a table referred to in a query. For instance, if an administrative query is executed during nonpeak hours, then a table-level lock may not impact the users of the system too much; however, it can reduce the locking overhead of the query and thereby improve its performance. In such cases, a query developer may override the lock manager's lock level selection for a table referred to in the query by using locking hints.

```
SELECT * FROM <TableName> WITH(TABLOCK)
```

But, be cautious when taking control away from SQL Server like this. Test it thoroughly prior to implementation.

Database-Level Lock

A database-level lock is maintained on a database and is identified as a DB lock. When an application makes a database connection, the lock manager assigns a database-level shared lock to the corresponding `session_id`. This prevents a user from accidentally dropping or restoring the database while other users are connected to it.

SQL Server ensures that the locks requested at one level respect the locks granted at other levels. For instance, once a user acquires a row-level lock on a table row, another user can't acquire a lock at any other level that may affect the integrity of the row. The second user may acquire a row-level lock on other rows or a page-level lock on other pages, but an incompatible page- or table-level lock containing the row won't be granted to other users.

The level at which locks should be applied need not be specified by a user or database administrator; the lock manager determines that automatically. It generally prefers row-level and key-level locks when accessing a small number of rows to aid concurrency. However, if the locking overhead of multiple low-level locks turns out to be very high, the lock manager automatically selects an appropriate higher-level lock.

Lock Operations and Modes

Because of the variety of operations that SQL Server needs to perform, an equally large and complex set of locking mechanisms are maintained. In addition to the different types of locks, there is an escalation path to change from one type of lock to another. The following sections describe these modes and processes, as well as their uses.

Lock Escalation

When a query is executed, SQL Server determines the required lock level for the database objects referred to in the query, and it starts executing the query after acquiring the required locks. During the query execution, the lock manager keeps track of the number of locks requested by the query to determine the need to escalate the lock level from the current level to a higher level.

The lock escalation threshold is determined by SQL Server during the course of a transaction. Row locks and page locks are automatically escalated to a table lock when a transaction exceeds its threshold. After the lock level is escalated to a table-level lock,

all the lower-level locks on the table are automatically released. This dynamic lock escalation feature of the lock manager optimizes the locking overhead of a query.

It is possible to establish a level of control over the locking mechanisms on a given table. For example, you can control whether lock escalation occurs. The following is the T-SQL syntax to make that change:

```
ALTER TABLE schema.table
SET (LOCK_ESCALATION = DISABLE);
```

This syntax will disable lock escalation on the table entirely (except for a few special circumstances). You can also set it to `TABLE`, which will cause the escalation to go to a table lock every single time. You can also set lock escalation on the table to `AUTO`, which will allow SQL Server to make the determination for the locking schema and any escalation necessary. If that table is partitioned, you may see the escalation change to the partition level. Again, exercise caution using these types of modifications to standard SQL Server behavior.

You also have the option to disable lock escalation on a wider basis by using trace flag 1224. This disables lock escalation based on the number of locks but leaves intact lock escalation based on memory pressure. You can also disable the memory pressure lock escalation as well as the number of locks by using trace flag 1211, but that's a dangerous choice and can lead to errors on your systems. I strongly suggest thorough testing before using either of these options.

Lock Modes

The degree of isolation required by different transactions may vary. For instance, consistency of data is not affected if two transactions read the data simultaneously; however, the consistency is affected if two transactions are allowed to modify the data simultaneously. Depending on the type of access requested, SQL Server uses different lock modes while locking resources.

- Shared (S)
- Update (U)
- Exclusive (X)

- Intent
  - Intent Shared (IS)
  - Intent Exclusive (IX)
  - Shared with Intent Exclusive (SIX)
- Schema
  - Schema Modification (Sch-M)
  - Schema Stability (Sch-S)
- Bulk Update (BU)
- Key-Range

Shared (S) Mode

Shared mode is used for read-only queries, such as a `SELECT` statement. It doesn't prevent other read-only queries from accessing the data simultaneously because the integrity of the data isn't compromised by the concurrent reads. However, concurrent data modification queries on the data are prevented to maintain data integrity. The (S) lock is held on the data until the data is read. By default, the (S) lock acquired by a `SELECT` statement is released immediately after the data is read. For example, consider the following transaction:

```
BEGIN TRAN
SELECT  *
FROM    Production.Product AS p
WHERE   p.ProductID = 1;
--Other queries
COMMIT
```

The (S) lock acquired by the `SELECT` statement is not held until the end of the transaction; instead, it is released immediately after the data is read by the `SELECT` statement under `read_ committed`, the default isolation level. This behavior of the (S) lock can be altered by using a higher isolation level or a lock hint.

Update (U) Mode

Update mode may be considered similar to the (S) lock, but it also includes an objective to modify the data as part of the same query. Unlike the (S) lock, the (U) lock indicates that the data is read for modification. Since the data is read with an objective to modify it, SQL Server does not allow more than one (U) lock on the data simultaneously. This rule helps maintain data integrity. Note that concurrent (S) locks on the data are allowed. The (U) lock is associated with an `UPDATE` statement, and the action of an `UPDATE` statement actually involves two intermediate steps: first read the data to be modified, and then modify the data.

Different lock modes are used in the two intermediate steps to maximize concurrency. Instead of acquiring an exclusive right while reading the data, the first step acquires a (U) lock on the data. In the second step, the (U) lock is converted to an exclusive lock for modification. If no modification is required, then the (U) lock is released; in other words, it's not held until the end of the transaction. Consider the following script, which would lead to blocking until the `UPDATE` statement is completed:

```
UPDATE Sales.Currency
SET Name = 'Euro'
WHERE CurrencyCode = 'EUR';
```

To understand the locking behavior of the intermediate steps of the `UPDATE` statement, you need to obtain data from `sys.dm_tran_locks` while queries run. You can obtain the lock status after each step of the `UPDATE` statement by following the steps outlined next. You're going have three connections open that I'll refer to as Connection 1, Connection 2, and Connection 3. This will require three different query windows in Management Studio. You'll run the queries in the connections I list in the order that I specify to arrive at a blocking situation. The point of this is to observe those blocks as they occur. Table 21-1 shows the different connections in different T-SQL query windows and the order of the queries to be run in them.

Table 21-1. *Order of the Scripts to Show UPDATE Blocking*

Script Order	T-SQL Window 1 (Connection 1)	T-SQL Window 2 (Connection 2)	T-SQL Window 3 (Connection 3)
1	BEGIN TRANSACTION LockTran2 --Retain an (S) lock on the resource SELECT * FROM Sales.Currency AS c WITH (REPEATABLEREAD) WHERE c.CurrencyCode = 'EUR' ; --Allow DMVs to be executed before second step of -- UPDATE statement is executed by transaction LockTran1 WAITFOR DELAY '00:00:10'; COMMIT		

(*continued*)

Table 21-1. (*continued*)

Script Order	T-SQL Window 1 (Connection 1)	T-SQL Window 2 (Connection 2)	T-SQL Window 3 (Connection 3)
2		BEGIN TRANSACTION LockTran1 UPDATE Sales. Currency SET Name = 'Euro' WHERE CurrencyCode = 'EUR'; -- NOTE: We're not committing yet	
3			SELECT dtl.request_ session_id, dtl.resource_database_id, dtl.resource_associated_ entity_id, dtl.resource_type, dtl.resource_ description, dtl.request_mode, dtl.request_status FROM sys.dm_tran_ locks AS dtl ORDER BY dtl.request_ session_id;

(*continued*)

Table 21-1. (*continued*)

Script Order	T-SQL Window 1 (Connection 1)	T-SQL Window 2 (Connection 2)	T-SQL Window 3 (Connection 3)
4) Wait 10 seconds			
5			`SELECT dtl.request_session_id, dtl.resource_database_id, dtl.resource_associated_entity_id, dtl.resource_type, dtl.resource_description, dtl.request_mode, dtl.request_status FROM sys.dm_tran_locks AS dtl ORDER BY dtl.request_session_id;`
6		`COMMIT`	

The `REPEATABLEREAD` locking hint, running in Connection 2, allows the `SELECT` statement to retain the (S) lock on the resource. The output from `sys.dm_tran_locks` in Connection 3 will provide the lock status after the first step of the `UPDATE` statement since the lock conversion to an exclusive (X) lock by the `UPDATE` statement is blocked by the `SELECT` statement. Next, let's look at the lock status provided by `sys.dm_tran_locks` as you go through the individual steps of the `UPDATE` statement.

Figure 21-3 shows the lock status after step 1 of the `UPDATE` statement (obtained from the output from `sys.dm_tran_locks` executed on the third connection, Connection 3, as explained previously).

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	52	6	0	DATABASE		S	GRANT
2	52	6	72057594048675840	KEY	(0d881dadfc5c)	U	GRANT
3	52	6	72057594048675840	KEY	(0d881dadfc5c)	X	CONVERT
4	52	6	1589580701	OBJECT		IX	GRANT
5	52	6	72057594048675840	PAGE	1:12304	IX	GRANT
6	53	6	72057594048675840	PAGE	1:12304	IS	GRANT
7	53	6	1589580701	OBJECT		IS	GRANT
8	53	6	72057594048675840	KEY	(0d881dadfc5c)	S	GRANT
9	53	6	0	DATABASE		S	GRANT
10	54	6	0	DATABASE		S	GRANT
11	55	9	0	DATABASE		S	GRANT
12	56	6	0	DATABASE		S	GRANT

Figure 21-3. *Output from sys.dm_tran_locks showing the lock conversion state of an UPDATE statement*

Note The order of these rows is not that important. I've ordered by `session_id` in order to group the locks from each query.

- Figure 21-4 shows the lock status after step 2 of the `UPDATE` statement.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	52	6	0	DATABASE		S	GRANT
2	52	6	72057594048675840	KEY	(0d881dadfc5c)	X	GRANT
3	52	6	1589580701	OBJECT		IX	GRANT
4	52	6	72057594048675840	PAGE	1:12304	IX	GRANT
5	53	6	0	DATABASE		S	GRANT
6	54	6	0	DATABASE		S	GRANT
7	55	9	0	DATABASE		S	GRANT
8	56	6	0	DATABASE		S	GRANT

Figure 21-4. *Output from sys.dm_tran_locks showing the final lock status held by the UPDATE statement*

From the `sys.dm_tran_locks` output after the first step of the `UPDATE` statement, you can note the following:

- A (U) lock is granted to the SPID on the data row.
- A conversion to an (X) lock on the data row is requested.

From the output of `sys.dm_tran_locks` after the second step of the `UPDATE` statement, you can see that the `UPDATE` statement holds only an (X) lock on the data row. Essentially, the (U) lock on the data row is converted to an (X) lock.

This is important, by not acquiring an exclusive lock at the first step, an UPDATE statement allows other transactions to read the data using the SELECT statement during that period. This is possible because (U) and (S) locks are compatible with each other. This increases database concurrency.

Note I discuss lock compatibility among different lock modes later in this chapter.

You may be curious to learn why a (U) lock is used instead of an (S) lock in the first step of the UPDATE statement. To understand the drawback of using an (S) lock instead of a (U) lock in the first step of the UPDATE statement, let's break the UPDATE statement into two steps.

1. Read the data to be modified using an (S) lock instead of a (U) lock.
2. Modify the data by acquiring an (X) lock.

Consider the following code:

```
BEGIN TRAN
--1.Read data to be modified using (S)lock instead of (U)lock.
--    Retain the (S)lock using REPEATABLEREAD locking hint, since
--    the original (U)lock is retained until the conversion to
--    (X)lock.
SELECT  *
FROM    Sales.Currency AS c WITH (REPEATABLEREAD)
WHERE   c.CurrencyCode = 'EUR' ;
--Allow another equivalent update action to start concurrently
WAITFOR DELAY '00:00:10' ;

--2. Modify the data by acquiring (X)lock
UPDATE  Sales.Currency WITH (XLOCK)
SET     Name = 'EURO'
WHERE   CurrencyCode = 'EUR' ;
COMMIT
```

If this transaction is executed from two connections simultaneously, then, after a delay, it causes a deadlock, as follows:

```
Msg 1205, Level 13, State 51, Line 13
Transaction (Process ID 58) was deadlocked on lock resources with another
process and has been chosen as the deadlock victim. Rerun the transaction.
```

Both transactions read the data to be modified using an (S) lock and then request an (X) lock for modification. When the first transaction attempts the conversion to the (X) lock, it is blocked by the (S) lock held by the second transaction. Similarly, when the second transaction attempts the conversion from (S) lock to the (X) lock, it is blocked by the (S) lock held by the first transaction, which in turn is blocked by the second transaction. This causes a circular block—and therefore, a deadlock.

Note Deadlocks are covered in more detail in Chapter 22.

To avoid this typical deadlock, the `UPDATE` statement uses a (U) lock instead of an (S) lock at its first intermediate step. Unlike an (S) lock, a (U) lock doesn't allow another (U) lock on the same resource simultaneously. This forces the second concurrent `UPDATE` statement to wait until the first `UPDATE` statement completes.

Exclusive (X) Mode

Exclusive mode provides an exclusive right on a database resource for modification by data manipulation queries such as `INSERT`, `UPDATE`, and `DELETE`. It prevents other concurrent transactions from accessing the resource under modification. Both the `INSERT` and `DELETE` statements acquire (X) locks at the very beginning of their execution. As explained earlier, the `UPDATE` statement converts to the (X) lock after the data to be modified is read. The (X) locks granted in a transaction are held until the end of the transaction.

The (X) lock serves two purposes.

- It prevents other transactions from accessing the resource under modification so that they see a value either before or after the modification, not a value undergoing modification.
- It allows the transaction modifying the resource to safely roll back to the original value before modification, if needed, since no other transaction is allowed to modify the resource simultaneously.

Intent Shared (IS), Intent Exclusive (IX), and Shared with Intent Exclusive (SIX) Modes

Intent Shared, Intent Exclusive, and Shared with Intent Exclusive locks indicate that the query intends to grab a corresponding (S) or (X) lock at a lower lock level. For example, consider the following transaction on the `Sales.Currency` table:

```
BEGIN TRAN
DELETE  Sales.Currency
WHERE   CurrencyCode = 'ALL';

SELECT  tl.request_session_id,
        tl.resource_database_id,
        tl.resource_associated_entity_id,
        tl.resource_type,
        tl.resource_description,
        tl.request_mode,
        tl.request_status
FROM    sys.dm_tran_locks tl;

ROLLBACK TRAN
```

Figure 21-5 shows the output from `sys.dm_tran_locks`.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	52	6	0	DATABASE		S	GRANT
2	54	6	0	DATABASE		S	GRANT
3	55	9	0	DATABASE		S	GRANT
4	52	6	72057594053918720	KEY	(f9b93c451603)	X	GRANT
5	52	6	72057594053918720	PAGE	1:9336	IX	GRANT
6	52	6	1589580701	OBJECT		IX	GRANT
7	52	6	72057594048675840	KEY	(cadf591d32de)	X	GRANT
8	52	6	72057594048675840	PAGE	1:12304	IX	GRANT

Figure 21-5. *Output from sys.dm_tran_locks showing the intent locks granted at higher levels*

The (IX) lock at the table level (`PAGE`) indicates that the `DELETE` statement intends to acquire an (X) lock at a page, row, or key level. Similarly, the (IX) lock at the page level (`PAGE`) indicates that the query intends to acquire an (X) lock on a row in the page. The (IX) locks at the higher levels prevent another transaction from acquiring an incompatible lock on the table or on the page containing the row.

Flagging the intent lock—(IS) or (IX)—at a corresponding higher level by a transaction, while holding the lock at a lower level, prevents other transactions from acquiring an incompatible lock at the higher level. If the intent locks were not used, then a transaction trying to acquire a lock at a higher level would have to scan through the lower levels to detect the presence of lower-level locks. While the intent lock at the higher levels indicates the presence of a lower level lock, the locking overhead of acquiring a lock at a higher level is optimized. The intent locks granted to a transaction are held until the end of the transaction.

Only a single (SIX) lock can be placed on a given resource at once. This prevents updates made by other transactions. Other transactions can place (IS) locks on the lower-level resources while the (SIX) lock is in place.

Furthermore, there can be a combination of locks requested (or acquired) at a certain level and the intention of having a lock (or locks) at a lower level. For example, there can be (SIU) and (UIX) lock combinations indicating that an (S) or a (U) lock has been acquired at the corresponding level and that (U) or (X) lock(s) are intended at a lower level.

Schema Modification (Sch-M) and Schema Stability (Sch-S) Modes

Schema Modification and Schema Stability locks are acquired on a table by SQL statements that depend on the schema of the table. A DDL statement, working on the schema of a table, acquires an (Sch-M) lock on the table and prevents other transactions from accessing the table. An (Sch-S) lock is acquired for database activities that depend on the schema but do not modify the schema, such as a query compilation. It prevents an (Sch-M) lock on the table, but it allows other locks to be granted on the table.

Since, on a production database, schema modifications are infrequent, (Sch-M) locks don't usually become a blocking issue. And because (Sch-S) locks don't block other locks except (Sch-M) locks, concurrency is generally not affected by (Sch-S) locks either.

Bulk Update (BU) Mode

The Bulk Update lock mode is unique to bulk load operations. These operations are the older-style `bcp` (bulk copy), the `BULK INSERT` statement, and inserts from the `OPENROWSET` using the `BULK` option. As a mechanism for speeding up these processes, you can provide

a TABLOCK hint or set the option on the table for it to lock on bulk load. The key to (BU) locking mode is that it will allow multiple bulk operations against the table being locked but prevent other operations while the bulk process is running.

Key-Range Mode

The Key-Range mode is applicable only while the isolation level is set to Serializable (you'll learn more about transaction isolation levels in the later "Isolation Levels" section). The Key-Range locks are applied to a series, or range, of key values that will be used repeatedly while the transaction is open. Locking a range during a serializable transaction ensures that other rows are not inserted within the range, possibly changing result sets within the transaction. The range can be locked using the other lock modes, making this more like a combined locking mode rather than a distinctively separate locking mode. For the Key-Range lock mode to work, an index must be used to define the values within the range.

Lock Compatibility

SQL Server provides isolation to a transaction by preventing other transactions from accessing the same resource in an incompatible way. However, if a transaction attempts a compatible task on the same resource, then to increase concurrency, it won't be blocked by the first transaction. SQL Server ensures this kind of selective blocking by preventing a transaction from acquiring an incompatible lock on a resource held by another transaction. For example, an (S) lock acquired on a resource by a transaction allows other transactions to acquire an (S) lock on the same resource. However, an (Sch-M) lock on a resource by a transaction prevents other transactions from acquiring any lock on that resource.

Isolation Levels

The lock modes explained in the previous section help a transaction protect its data consistency from other concurrent transactions. The degree of data protection or isolation a transaction gets depends not only on the lock modes but also on the isolation level of the transaction. This level influences the behavior of the lock modes. For example, by default an (S) lock is released immediately after the data is read; it isn't held

until the end of the transaction. This behavior may not be suitable for some application functionality. In such cases, you can configure the isolation level of the transaction to achieve the desired degree of isolation.

SQL Server implements six isolation levels, four of them as defined by ISO:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

Two other isolation levels provide row versioning, which is a mechanism whereby a version of the row is created as part of data manipulation queries. This extra version of the row allows read queries to access the data without acquiring locks against it. The extra two isolation levels are as follows:

- Read Committed Snapshot (actually part of the Read Committed isolation)
- Snapshot

The four ISO isolation levels are listed in increasing order of degree of isolation. You can configure them at either the connection or query level by using the `SET TRANSACTION ISOLATION LEVEL` statement or the locking hints, respectively. The isolation level configuration at the connection level remains effective until the isolation level is reconfigured using the `SET` statement or until the connection is closed. All the isolation levels are explained in the sections that follow.

Read Uncommitted

Read Uncommitted is the lowest of the four isolation levels, and it allows `SELECT` statements to read data without requesting an (S) lock. Since an (S) lock is not requested by a `SELECT` statement, it neither blocks nor is blocked by the (X) lock. It allows a `SELECT` statement to read data while the data is under modification. This kind of data read is called a *dirty read.*

Assume you have an application in which the amount of data modification is extremely minimal and that your application doesn't require much in the way of accuracy from the `SELECT` statement it issues to read data. In this case, you can use the Read Uncommitted isolation level to avoid having some other data modification activity block the `SELECT` statement.

You can use the following SET statement to configure the isolation level of a database connection to the Read Uncommitted isolation level:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

You can also achieve this degree of isolation on a query basis using the NOLOCK locking hint.

```
SELECT  *
FROM    Production.Product AS p WITH (NOLOCK);
```

The effect of the locking hint remains applicable for the query and doesn't change the isolation level of the connection.

The Read Uncommitted isolation level avoids the blocking caused by a SELECT statement, but you should not use it if the transaction depends on the accuracy of the data read by the SELECT statement or if the transaction cannot withstand a concurrent change of data by another transaction.

It's important to understand what is meant by a dirty read. Lots of people think this means that, while a field is being updated from Tusa to Tulsa, a query can still read the previous value or even the updated value, prior to the commit. Although that is true, much more egregious data problems could occur. Since no locks are placed while reading the data, indexes may be split. This can result in extra or missing rows of data returned to the query. To be clear, using Read Uncommitted in any environment where data manipulation as well as data reads are occurring can result in unanticipated behaviors. The intention of this isolation level is for systems primarily focused on reporting and business intelligence, not online transaction processing. You may see radically incorrect data because of the use of uncommitted data. This fact cannot be over-emphasized.

Read Committed

The Read Committed isolation level prevents the dirty read caused by the Read Uncommitted isolation level. This means that (S) locks are requested by the SELECT statements at this isolation level. This is the default isolation level of SQL Server. If needed, you can change the isolation level of a connection to Read Committed by using the following SET statement:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

The Read Committed isolation level is good for most cases, but since the (S) lock acquired by the SELECT statement isn't held until the end of the transaction, it can cause nonrepeatable read or phantom read issues, as explained in the sections that follow.

The behavior of the Read Committed isolation level can be changed by the READ_COMMITTED_SNAPSHOT database option. When this is set to ON, row versioning is used by data manipulation transactions. This places an extra load on tempdb because previous versions of the rows being changed are stored there while the transaction is uncommitted. This allows other transactions to access data for reads without having to place locks on the data, which can improve the speed and efficiency of all the queries in the system without resulting in the issues generated by page splits with NOLOCK or READ UNCOMMITTED. In Azure SQL Database, the default setting is READ_COMMITTED_SNAPSHOT.

Next, modify the AdventureWorks2017 database so that READ_COMMITTED_SNAPSHOT is turned on.

```
ALTER DATABASE AdventureWorks2017 SET READ_COMMITTED_SNAPSHOT ON;
```

Now imagine a business situation. The first connection and transaction will be pulling data from the Production.Product table, acquiring the color of a particular item.

```
BEGIN TRANSACTION;
SELECT  p.Color
FROM    Production.Product AS p
WHERE   p.ProductID = 711;
```

A second connection is made with a new transaction that will be modifying the color of the same item.

```
BEGIN TRANSACTION ;
UPDATE  Production.Product
SET     Color = 'Coyote'
WHERE   ProductID = 711;
SELECT  p.Color
FROM    Production.Product AS p
WHERE   p.ProductID = 711;
```

Running the SELECT statement after updating the color, you can see that the color was updated. But if you switch back to the first connection and rerun the original SELECT statement (don't run the BEGIN TRAN statement again), you'll still see the color as Blue. Switch back to the second connection and finish the transaction.

```
COMMIT TRANSACTION;
```

Switching again to the first transaction, commit that transaction, and then rerun the original SELECT statement. You'll see the new color updated for the item, Coyote. You can reset the isolation level on AdventureWorks2017 before continuing.

```
ALTER DATABASE AdventureWorks2017 SET READ_COMMITTED_SNAPSHOT OFF;
```

Note If the tempdb is filled, data modification using row versioning will continue to succeed, but reads may fail since the versioned row will not be available. If you enable any type of row versioning isolation within your database, you must take extra care to maintain free space within tempdb.

Repeatable Read

The Repeatable Read isolation level allows a SELECT statement to retain its (S) lock until the end of the transaction, thereby preventing other transactions from modifying the data during that time. Database functionality may implement a logical decision inside a transaction based on the data read by a SELECT statement within the transaction. If the outcome of the decision is dependent on the data read by the SELECT statement, then you should consider preventing modification of the data by other concurrent transactions. For example, consider the following two transactions:

- *Normalize the price for* ProductID = *1*: For ProductID = 1, if Price > 10, then decrease the price by 10.
- *Apply a discount*: For products with Price > 10, apply a discount of 40 percent.

Now consider the following test table:

```
DROP TABLE IF EXISTS dbo.MyProduct;
GO
CREATE TABLE dbo.MyProduct (ProductID INT,
                            Price MONEY);
INSERT INTO dbo.MyProduct
VALUES (1, 15.0);
```

You can write the two transactions like this:

```
DECLARE @Price INT ;
BEGIN TRAN NormailizePrice
SELECT  @Price = mp.Price
FROM    dbo.MyProduct AS mp
WHERE   mp.ProductID = 1 ;
/*Allow transaction 2 to execute*/
WAITFOR DELAY '00:00:10' ;
IF @Price > 10
    UPDATE  dbo.MyProduct
    SET     Price = Price - 10
    WHERE   ProductID = 1 ;
COMMIT

--Transaction 2 from Connection 2
BEGIN TRAN ApplyDiscount
UPDATE  dbo.MyProduct
SET     Price = Price * 0.6 --Discount = 40%
WHERE   Price > 10 ;
COMMIT
```

On the surface, the preceding transactions may look good, and yes, they do work in a single-user environment. But in a multiuser environment, where multiple transactions can be executed concurrently, you have a problem here!

To figure out the problem, let's execute the two transactions from different connections in the following order:

1. Start transaction 1 first.
2. Start transaction 2 within ten seconds of the start of transaction 1.

As you may have guessed, at the end of the transactions, the new price of the product (with `ProductID = 1`) will be `-1.0`. Ouch—it appears that you're ready to go out of business!

The problem occurs because transaction 2 is allowed to modify the data while transaction 1 has finished reading the data and is about to make a decision on it. Transaction 1 requires a higher degree of isolation than that provided by the default isolation level (Read Committed).

As a solution, you want to prevent transaction 2 from modifying the data while transaction 1 is working on it. In other words, provide transaction 1 with the ability to read the data again later in the transaction without being modified by others. This feature is called *repeatable read.* Considering the context, the implementation of the solution is probably obvious. After re-creating the sample table, you can write this:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
GO
--Transaction 1 from Connection 1
DECLARE @Price INT ;
BEGIN TRAN NormalizePrice
SELECT  @Price = Price
FROM    dbo.MyProduct AS mp
WHERE   mp.ProductID = 1 ;
/*Allow transaction 2 to execute*/
WAITFOR DELAY  '00:00:10' ;
IF @Price > 10
    UPDATE  dbo.MyProduct
    SET     Price = Price - 10
    WHERE   ProductID = 1 ;
COMMIT
GO
SET TRANSACTION ISOLATION LEVEL READ COMMITTED --Back to default
GO
```

Increasing the isolation level of transaction 1 to Repeatable Read will prevent transaction 2 from modifying the data during the execution of transaction 1. Consequently, you won't have an inconsistency in the price of the product. Since the intention isn't to release the (S) lock acquired by the SELECT statement until the end of the transaction, the effect of setting the isolation level to Repeatable Read can also be implemented at the query level using the lock hint.

```
DECLARE @Price INT ;
BEGIN TRAN NormalizePrice
SELECT  @Price = Price
FROM    dbo.MyProduct AS mp WITH (REPEATABLEREAD)
WHERE   mp.ProductID = 1 ;
/*Allow transaction 2 to execute*/
WAITFOR DELAY  '00:00:10'
IF @Price > 10
    UPDATE  dbo.MyProduct
    SET     Price = Price - 10
    WHERE   ProductID = 1 ;
COMMIT
```

This solution prevents the data inconsistency of MyProduct.Price, but it introduces another problem to this scenario. On observing the result of transaction 2, you realize that it could cause a deadlock. Therefore, although the preceding solution prevented the data inconsistency, it is not a complete solution. Looking closely at the effect of the Repeatable Read isolation level on the transactions, you see that it introduced the typical deadlock issue avoided by the internal implementation of an UPDATE statement, as explained previously. The SELECT statement acquired and retained an (S) lock instead of a (U) lock, even though it intended to modify the data later within the transaction. The (S) lock allowed transaction 2 to acquire a (U) lock, but it blocked the (U) lock's conversion to an (X) lock. The attempt of transaction 1 to acquire a (U) lock on the data at a later stage caused a circular block, resulting in a deadlock.

To prevent the deadlock and still avoid data corruption, you can use an equivalent strategy as adopted by the internal implementation of the UPDATE statement. Thus, instead of requesting an (S) lock, transaction 1 can request a (U) lock by using an UPDLOCK locking hint when executing the SELECT statement.

```
DECLARE @Price INT ;
BEGIN TRAN NormalizePrice
SELECT  @Price = Price
FROM    dbo.MyProduct AS mp WITH (UPDLOCK)
WHERE   mp.ProductID = 1 ;
/*Allow transaction 2 to execute*/
WAITFOR DELAY  '00:00:10'
IF @Price > 10
    UPDATE  dbo.MyProduct
    SET     Price = Price - 10
    WHERE   ProductID = 1 ;
COMMIT
```

This solution prevents both data inconsistency and the possibility of the deadlock. If the increase of the isolation level to Repeatable Read had not introduced the typical deadlock, then it would have done the job. Since there is a chance of a deadlock occurring because of the retention of an (S) lock until the end of a transaction, it is usually preferable to grab a (U) lock instead of holding the (S) lock, as just illustrated.

Serializable

Serializable is the highest of the six isolation levels. Instead of acquiring a lock only on the row to be accessed, the Serializable isolation level acquires a range lock on the row and the next row in the order of the data set requested. For instance, a SELECT statement executed at the Serializable isolation level acquires a (RangeS-S) lock on the row to be accessed and the next row in the order. This prevents the addition of rows by other transactions in the data set operated on by the first transaction, and it protects the first transaction from finding new rows in its data set within its transaction scope. Finding new rows in a data set within a transaction is also called a *phantom read.*

To understand the need for a Serializable isolation level, let's consider an example. Suppose a group (with GroupID = 10) in a company has a fund of $100 to be distributed

among the employees in the group as a bonus. The fund balance after the bonus payment should be $0. Consider the following test table:

```
DROP TABLE IF EXISTS dbo.MyEmployees;
GO
CREATE TABLE dbo.MyEmployees (EmployeeID INT,
                              GroupID INT,
                              Salary MONEY);
CREATE CLUSTERED INDEX i1 ON dbo.MyEmployees (GroupID);

--Employee 1 in group 10
INSERT INTO dbo.MyEmployees
VALUES (1, 10, 1000),
       --Employee 2 in group 10
       (2, 10, 1000),
       --Employees 3 & 4 in different groups
       (3, 20, 1000),
       (4, 9, 1000);
```

The described business functionality may be implemented as follows:

```
DECLARE @Fund MONEY = 100,
    @Bonus MONEY,
    @NumberOfEmployees INT;

BEGIN TRAN PayBonus
SELECT  @NumberOfEmployees = COUNT(*)
FROM    dbo.MyEmployees
WHERE   GroupID = 10;

/*Allow transaction 2 to execute*/
WAITFOR DELAY  '00:00:10';

IF @NumberOfEmployees > 0
    BEGIN
        SET @Bonus = @Fund / @NumberOfEmployees;
        UPDATE  dbo.MyEmployees
        SET     Salary = Salary + @Bonus
```

```
        WHERE   GroupID = 10;
        PRINT 'Fund balance =
' + CAST((@Fund - (@@ROWCOUNT * @Bonus)) AS VARCHAR(6)) + '   $';
    END
COMMIT
```

You'll see the returned value as a fund balance of $0 since the updates complete successfully. The PayBonus transaction works well in a single-user environment. However, in a multiuser environment, there is a problem.

Consider another transaction that adds a new employee to GroupID = 10 as follows and is executed concurrently (immediately after the start of the PayBonus transaction) from a second connection:

```
BEGIN TRAN NewEmployee
INSERT  INTO MyEmployees
VALUES  (5, 10, 1000);
COMMIT
```

The fund balance after the PayBonus transaction will be -$50! Although the new employee may like it, the group fund will be in the red. This causes an inconsistency in the logical state of the data.

To prevent this data inconsistency, the addition of the new employee to the group (or data set) under operation should be blocked. Of the five isolation levels discussed, only Snapshot isolation can provide a similar functionality, since the transaction has to be protected not only on the existing data but also from the entry of new data in the data set. The Serializable isolation level can provide this kind of isolation by acquiring a range lock on the affected row and the next row in the order determined by the MyEmployees.il index on the GroupID column. Thus, the data inconsistency of the PayBonus transaction can be prevented by setting the transaction isolation level to Serializable.

Remember to re-create the table first.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
GO
DECLARE @Fund MONEY = 100,
    @Bonus MONEY,
    @NumberOfEmployees INT;
```

```
BEGIN TRAN PayBonus
SELECT  @NumberOfEmployees = COUNT(*)
FROM    dbo.MyEmployees
WHERE   GroupID = 10;

/*Allow transaction 2 to execute*/
WAITFOR DELAY  '00:00:10';
IF @NumberOfEmployees > 0
    BEGIN
        SET @Bonus = @Fund / @NumberOfEmployees;
        UPDATE  dbo.MyEmployees
        SET     Salary = Salary + @Bonus
        WHERE   GroupID = 10;

        PRINT 'Fund balance =
' + CAST((@Fund - (@@ROWCOUNT * @Bonus)) AS VARCHAR(6)) + '   $';
    END
COMMIT
GO
--Back to default
SET TRANSACTION ISOLATION LEVEL READ COMMITTED ;
GO
```

The effect of the Serializable isolation level can also be achieved at the query level by using the HOLDLOCK locking hint on the SELECT statement, as shown here:

```
DECLARE @Fund MONEY = 100,
    @Bonus MONEY,
    @NumberOfEmployees INT ;

BEGIN TRAN PayBonus
SELECT  @NumberOfEmployees = COUNT(*)
FROM    dbo.MyEmployees WITH (HOLDLOCK)
WHERE   GroupID = 10 ;

/*Allow transaction 2 to execute*/
WAITFOR DELAY  '00:00:10' ;

IF @NumberOfEmployees > 0
```

```
    BEGIN
        SET @Bonus = @Fund / @NumberOfEmployees
        UPDATE  dbo.MyEmployees
        SET     Salary = Salary + @Bonus
        WHERE   GroupID = 10 ;

        PRINT 'Fund balance =
' + CAST((@Fund - (@@ROWCOUNT * @Bonus)) AS VARCHAR(6)) + '   $' ;
    END
COMMIT
```

You can observe the range locks acquired by the `PayBonus` transaction by querying `sys.dm_tran_locks` from another connection while the `PayBonus` transaction is executing, as shown in Figure 21-6.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	54	5	0	DATABASE		S	GRANT
2	53	5	0	DATABASE		S	GRANT
3	52	5	0	DATABASE		S	GRANT
4	52	5	72057594071678976	KEY	(fca1e333d991)	RangeS-S	GRANT
5	52	5	72057594071678976	KEY	(1c95cdbc1d2d)	RangeS-S	GRANT
6	53	5	72057594071678976	PAGE	1:24272	IX	GRANT
7	52	5	72057594071678976	PAGE	1:24272	IS	GRANT
8	52	5	72057594071678976	KEY	(241332e1ddb0)	RangeS-S	GRANT
9	53	5	2020202247	OBJECT		IX	GRANT
10	52	5	2020202247	OBJECT		IS	GRANT
11	52	5	72057594071678976	KEY	(69c872e07e60)	RangeS-S	GRANT
12	53	5	72057594071678976	KEY	(69c872e07e60)	RangeI-N	WAIT

Figure 21-6. *Output from sys.dm_tran_locks showing range locks granted to the serializable transaction*

The output of `sys.dm_tran_locks` shows that shared-range (RangeS-S) locks are acquired on three index rows: the first employee in `GroupID = 10`, the second employee in `GroupID = 10`, and the third employee in `GroupID = 20`. These range locks prevent the entry of any new employee in `GroupID = 10`.

The range locks just shown introduce a few interesting side effects.

- No new employee with a `GroupID` between 10 and 20 can be added during this period. For instance, an attempt to add a new employee with a `GroupID` of 15 will be blocked by the `PayBonus` transaction.

```
BEGIN TRAN NewEmployee
INSERT  INTO dbo.MyEmployees
VALUES  (6, 15, 1000);
COMMIT
```

- If the data set of the `PayBonus` transaction turns out to be the last set in the existing data ordered by the index, then the range lock required on the row, after the last one in the data set, is acquired on the last possible data value in the table.

  To understand this behavior, let's delete the employees with a `GroupID > 10` to make the `GroupID = 10` data set the last data set in the clustered index (or table).

```
DELETE  dbo.MyEmployees
WHERE   GroupID > 10;
```

Run the updated bonus and `newemployee` again. Figure 21-7 shows the resultant output of `sys.dm_tran_locks` for the `PayBonus` transaction.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	54	5	0	DATABASE		S	GRANT
2	53	5	0	DATABASE		S	GRANT
3	52	5	0	DATABASE		S	GRANT
4	52	5	72057594071678976	KEY	(ffffffffffff)	RangeS-S	GRANT
5	53	5	72057594071678976	KEY	(ffffffffffff)	RangeI-N	WAIT
6	52	5	72057594071678976	KEY	(e5d9a62bf821)	RangeS-S	GRANT
7	52	5	72057594071678976	KEY	(fca1e333d991)	RangeS-S	GRANT
8	52	5	72057594071678976	KEY	(1c95cdbc1d2d)	RangeS-S	GRANT
9	53	5	72057594071678976	PAGE	1:24272	IX	GRANT
10	52	5	72057594071678976	PAGE	1:24272	IS	GRANT
11	52	5	72057594071678976	KEY	(241332e1ddb0)	RangeS-S	GRANT
12	53	5	2020202247	OBJECT		IX	GRANT
13	52	5	2020202247	OBJECT		IS	GRANT

Figure 21-7. *Output from sys.dm_tran_locks showing extended range locks granted to the serializable transaction*

The range lock on the last possible row (KEY = ffffffffffff) in the clustered index, as shown in Figure 21-7, will block the addition of employees with all GroupIDs greater than or equal to 10. You know that the lock is on the last row, not because it's displayed in a visible fashion in the output of sys.dm_tran_locks but because you cleaned out everything up to that row previously. For example, an attempt to add a new employee with GroupID = 999 will be blocked by the PayBonus transaction.

```
BEGIN TRAN NewEmployee
INSERT  INTO dbo.MyEmployees
VALUES  (7, 999, 1000);
COMMIT
```

Guess what will happen if the table doesn't have an index on the GroupID column (in other words, the column in the WHERE clause)? While you're thinking, I'll re-create the table with the clustered index on a different column.

```
DROP TABLE IF EXISTS dbo.MyEmployees;
GO
CREATE TABLE dbo.MyEmployees (EmployeeID INT,
                              GroupID INT,
                              Salary MONEY);
CREATE CLUSTERED INDEX i1 ON dbo.MyEmployees (EmployeeID);

--Employee 1 in group 10
INSERT INTO dbo.MyEmployees
VALUES (1, 10, 1000),
       --Employee 2 in group 10
       (2, 10, 1000),
       --Employees 3 & 4 in different groups
       (3, 20, 1000),
       (4, 9, 1000);
```

Now rerun the updated bonus query and the new employee query. Figure 21-8 shows the resultant output of sys.dm_tran_locks for the PayBonus transaction.

Once again, the range lock on the last possible row (KEY = ffffffffffff) in the new clustered index, as shown in Figure 21-8, will block the addition of any new row to the table. I will discuss the reason behind this extensive locking later in the chapter in the "Effect of Indexes on the Serializable Isolation Level" section.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	54	5	0	DATABASE		S	GRANT
2	53	5	0	DATABASE		S	GRANT
3	52	5	0	DATABASE		S	GRANT
4	52	5	72057594071678976	KEY	(ffffffffffff)	RangeS-S	GRANT
5	53	5	72057594071678976	KEY	(ffffffffffff)	RangeI-N	WAIT
6	52	5	72057594071678976	KEY	(e5d9a62bf821)	RangeS-S	GRANT
7	52	5	72057594071678976	KEY	(ddfc91a29454)	RangeS-S	GRANT
8	52	5	72057594071678976	KEY	(fca1e333d991)	RangeS-S	GRANT
9	52	5	72057594071678976	KEY	(1c95cdbc1d2d)	RangeS-S	GRANT
10	53	5	72057594071678976	PAGE	1:24272	IX	GRANT
11	52	5	72057594071678976	PAGE	1:24272	IS	GRANT
12	52	5	72057594071678976	KEY	(241332e1ddb0)	RangeS-S	GRANT
13	53	5	2020202247	OBJECT		IX	GRANT
14	52	5	2020202247	OBJECT		IS	GRANT

Figure 21-8. *Output from sys.dm_tran_locks showing range locks granted to the serializable transaction with no index on the WHERE clause column*

As you've seen, the Serializable isolation level not only holds the share locks until the end of the transaction like the Repeatable Read isolation level but also prevents any new row from appearing in the data set by holding range locks. Because this increased blocking can hurt database concurrency, you should avoid the Serializable isolation level. If you have to use Serializable, then be sure you have good indexes and queries in place to optimize performance in order to minimize the size and length of your transactions.

Snapshot

Snapshot isolation is the second of the row-versioning isolation levels available in SQL Server since SQL Server 2005. Unlike Read Committed Snapshot isolation, Snapshot isolation requires an explicit call to `SET TRANSACTION ISOLATION LEVEL` at the start of the transaction. It also requires setting the isolation level on the database. Snapshot isolation is meant as a more stringent isolation level than the Read Committed Snapshot isolation. Snapshot isolation will attempt to put an exclusive lock on the data it intends to modify. If that data already has a lock on it, the snapshot transaction will fail. It provides transaction-level read consistency, which makes it more applicable to financial-type systems than Read Committed Snapshot.

Effect of Indexes on Locking

Indexes affect the locking behavior on a table. On a table with no indexes, the lock granularities are RID, PAG (on the page containing the RID), and TAB. Adding indexes to the table affects the resources to be locked. For example, consider the following test table with no indexes:

```
DROP TABLE IF EXISTS dbo.Test1;
GO

CREATE TABLE dbo.Test1 (C1 INT,
                        C2 DATETIME);

INSERT INTO dbo.Test1
VALUES (1, GETDATE());
```

Next, observe the locking behavior on the table for the transaction:

```
BEGIN TRAN LockBehavior
UPDATE  dbo.Test1 WITH (REPEATABLEREAD)  --Hold all acquired locks
SET     C2 = GETDATE()
WHERE   C1 = 1 ;
--Observe lock behavior from another connection
WAITFOR DELAY  '00:00:10' ;
COMMIT
```

Figure 21-9 shows the output of sys.dm_tran_locks applicable to the test table.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	52	6	0	DATABASE		S	GRANT
2	62	6	0	DATABASE		S	GRANT
3	55	9	0	DATABASE		S	GRANT
4	62	6	2020202247	OBJECT		IX	GRANT
5	62	6	72057594078232576	PAGE	1:42448	IX	GRANT
6	62	6	72057594078232576	RID	1:42448:0	X	GRANT

Figure 21-9. *Output from sys.dm_tran_locks showing the locks granted on a table with no index*

The following locks are acquired by the transaction:

- An (IX) lock on the table
- An (IX) lock on the page containing the data row
- An (X) lock on the data row within the table

When the `resource_type` is an object, the `resource_associated_entity_id` column value in `sys.dm_tran_locks` indicates the `objectid` of the object on which the lock is placed. You can obtain the specific object name on which the lock is acquired from the `sys.object` system table, as follows:

```
SELECT OBJECT_NAME(<object_id>);
```

The effect of the index on the locking behavior of the table varies with the type of index on the `WHERE` clause column. The difference arises from the fact that the leaf pages of the nonclustered and clustered indexes have a different relationship with the data pages of the table. Let's look into the effect of these indexes on the locking behavior of the table.

Effect of a Nonclustered Index

Because the leaf pages of the nonclustered index are separate from the data pages of the table, the resources associated with the nonclustered index are also protected from corruption. SQL Server automatically ensures this. To see this in action, create a nonclustered index on the test table.

```
CREATE NONCLUSTERED INDEX iTest ON dbo.Test1(C1);
```

On running the `LockBehavior` transaction again and querying `sys.dm_tran_locks` from a separate connection, you get the result shown in Figure 21-10.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	52	6	0	DATABASE		S	GRANT
2	62	6	0	DATABASE		S	GRANT
3	55	9	0	DATABASE		S	GRANT
4	62	6	72057594078298112	PAGE	1:50688	IU	GRANT
5	62	6	2020202247	OBJECT		IX	GRANT
6	62	6	72057594078232576	PAGE	1:42448	IX	GRANT
7	62	6	72057594078232576	RID	1:42448:0	X	GRANT
8	62	6	72057594078298112	KEY	(bb13f7b7fe64)	U	GRANT

Figure 21-10. *Output from sys.dm_tran_locks showing the effect of a nonclustered index on locking behavior*

The following locks are acquired by the transaction:

- An (IU) lock on the page containing the nonclustered index row
- A (U) lock on the nonclustered index row within the index page
- An (IX) lock on the table
- An (IX) lock on the page containing the data row
- An (X) lock on the data row within the data page

Note that only the row-level and page-level locks are directly associated with the nonclustered index. The next higher level of lock granularity for the nonclustered index is the table-level lock on the corresponding table.

Thus, nonclustered indexes introduce an additional locking overhead on the table. You can avoid the locking overhead on the index by using the `ALLOW_ROW_LOCKS` and `ALLOW_PAGE_LOCKS` options in `ALTER INDEX`. Understand, though, that this is a trade-off that could involve a loss of performance, and it requires careful testing to ensure it doesn't negatively impact your system.

```
ALTER INDEX iTest ON dbo.Test1
    SET (ALLOW_ROW_LOCKS = OFF ,ALLOW_PAGE_LOCKS= OFF);

BEGIN TRAN LockBehavior
UPDATE  dbo.Test1 WITH (REPEATABLEREAD)  --Hold all acquired locks
SET     C2 = GETDATE()
WHERE   C1 = 1;

--Observe lock behavior using sys.dm_tran_locks
--from another connection
WAITFOR DELAY  '00:00:10';
COMMIT

ALTER INDEX iTest ON dbo.Test1
    SET (ALLOW_ROW_LOCKS = ON ,ALLOW_PAGE_LOCKS= ON);
```

You can use these options when working with an index to enable/disable the `KEY` locks and `PAG` locks on the index. Disabling just the `KEY` lock causes the lowest lock granularity on the index to be the `PAG` lock. Configuring lock granularity on the index remains effective until it is reconfigured.

Note Modifying locks like this should be a last resort after many other options have been tried. This could cause significant locking overhead that would seriously impact the performance of the system.

Figure 21-11 displays the output of `sys.dm_tran_locks` executed from a separate connection.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	52	6	0	DATABASE		S	GRANT
2	62	6	0	DATABASE		S	GRANT
3	55	9	0	DATABASE		S	GRANT
4	62	6	2020202247	OBJECT		X	GRANT

Figure 21-11. *Output from sys.dm_tran_locks showing the effect of sp_index option on lock granularity*

The only lock acquired by the transaction on the test table is an (X) lock on the table.

You can see from the new locking behavior that disabling the KEY lock escalates lock granularity to the table level. This will block every concurrent access to the table or to the indexes on the table; consequently, it can seriously hurt the database concurrency. However, if a nonclustered index becomes a point of contention in a blocking scenario, then it may be beneficial to disable the PAG locks on the index, thereby allowing only KEY locks on the index.

Note Using this option can have serious side effects. You should use it only as a last resort.

Effect of a Clustered Index

Since for a clustered index the leaf pages of the index and the data pages of the table are the same, the clustered index can be used to avoid the overhead of locking additional pages (leaf pages) and rows introduced by a nonclustered index. To understand the locking overhead associated with a clustered index, convert the preceding nonclustered index to a clustered index.

```
CREATE CLUSTERED INDEX iTest ON dbo.Test1(C1) WITH DROP_EXISTING;
```

If you run the locking script again and query `sys.dm_tran_locks` in a different connection, you should see the resultant output for the `LockBehavior` transaction on `iTest` shown in Figure 21-12.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
1	52	6	0	DATABASE		S	GRANT
2	62	6	0	DATABASE		S	GRANT
3	55	9	0	DATABASE		S	GRANT
4	62	6	72057594078363648	KEY	(de42f79bc795)	X	GRANT
5	62	6	2020202247	OBJECT		IX	GRANT
6	62	6	72057594078363648	PAGE	1:62608	IX	GRANT

Figure 21-12. *Output from sys.dm_tran_locks showing the effect of a clustered index on locking behavior*

The following locks are acquired by the transaction:

- An (IX) lock on the table
- An (IX) lock on the page containing the clustered index row
- An (X) lock on the clustered index row within the table or clustered index

The locks on the clustered index row and the leaf page are actually the locks on the data row and data page, too, since the data pages and the leaf pages are the same. Thus, the clustered index reduced the locking overhead on the table compared to the nonclustered index.

Reduced locking overhead of a clustered index is another benefit of using a clustered index over a heap.

Effect of Indexes on the Serializable Isolation Level

Indexes play a significant role in determining the amount of blocking caused by the Serializable isolation level. The availability of an index on the `WHERE` clause column (that causes the data set to be locked) allows SQL Server to determine the order of the rows to be locked. For instance, consider the example used in the section on the Serializable isolation level. The `SELECT` statement uses a filter on the `GroupID` column to form its data set, like so:

```
DECLARE @NumberOfEmployees INT;
SELECT  @NumberOfEmployees = COUNT(*)
```

```
FROM    dbo.MyEmployees WITH (HOLDLOCK)
WHERE   GroupID = 10;
```

A clustered index is available on the `GroupID` column, allowing SQL Server to acquire a (RangeS-S) lock on the row to be accessed and the next row in the correct order.

If the index on the `GroupID` column is removed, then SQL Server cannot determine the rows on which the range locks should be acquired since the order of the rows is no longer guaranteed. Consequently, the `SELECT` statement acquires an (IS) lock at the table level instead of acquiring lower-granularity locks at the row level, as shown in Figure 21-13.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
11	62	6	2004202190	OBJECT		IS	GRANT

Figure 21-13. *Output from sys.dm_tran_locks showing the locks granted to a SELECT statement with no index on the WHERE clause column*

By failing to have an index on the filter column, you significantly increase the degree of blocking caused by the Serializable isolation level. This is another good reason to have an index on the `WHERE` clause columns.

Capturing Blocking Information

Although blocking is necessary to isolate a transaction from other concurrent transactions, sometimes it may rise to excessive levels, adversely affecting database concurrency. In the simplest blocking scenario, the lock acquired by a session on a resource blocks another session requesting an incompatible lock on the resource. To improve concurrency, it is important to analyze the cause of blocking and apply the appropriate resolution.

In a blocking scenario, you need the following information to have a clear understanding of the cause of the blocking:

- *The connection information of the blocking and blocked sessions*: You can obtain this information from the `sys.dm_os_waiting_tasks` dynamic management view.
- *The lock information of the blocking and blocked sessions*: You can obtain this information from the `sys.dm_tran_locks` DMO.

- *The SQL statements last executed by the blocking and blocked sessions:* You can use the `sys.dm_exec_requests` DMV combined with `sys.dm_exec_sql_text` and `sys.dm_exec_queryplan` or Extended Events to obtain this information.

You can also obtain the following information from SQL Server Management Studio by running the Activity Monitor. The Processes page provides connection information of all SPIDs. This shows blocked SPIDS, the process blocking them, and the head of any blocking chain with details on how long the process has been running, its SPID, and other information. It is possible to put Extended Events to work using the blocking report to gather a lot of the same information. For immediate checks on locking, use the DMOs; for extended monitoring and historical tracking, you'll want to use Extended Events. You can find more on this in the "Extended Events and the blocked_process_report Event" section.

To provide more power and flexibility to the process of collecting blocking information, a SQL Server administrator can use SQL scripts to provide the relevant information listed here.

Capturing Blocking Information with SQL

To arrive at enough information about blocked and blocking processes, you can bring several dynamic management views into play. This query will show information necessary to identify blocked processes based on those that are waiting. You can easily add filtering to access only those processes blocked for a certain period of time or only within certain databases, among other options.

```
SELECT  dtl.request_session_id AS WaitingSessionID,
        der.blocking_session_id AS BlockingSessionID,
        dowt.resource_description,
        der.wait_type,
        dowt.wait_duration_ms,
        DB_NAME(dtl.resource_database_id) AS DatabaseName,
        dtl.resource_associated_entity_id AS WaitingAssociatedEntity,
        dtl.resource_type AS WaitingResourceType,
        dtl.request_type AS WaitingRequestType,
        dest.[text] AS WaitingTSql,
        dtlbl.request_type BlockingRequestType,
```

```
        destbl.[text] AS BlockingTsql
FROM    sys.dm_tran_locks AS dtl
JOIN    sys.dm_os_waiting_tasks AS dowt
        ON dtl.lock_owner_address = dowt.resource_address
JOIN    sys.dm_exec_requests AS der
        ON der.session_id = dtl.request_session_id
CROSS APPLY sys.dm_exec_sql_text(der.sql_handle) AS dest
LEFT JOIN sys.dm_exec_requests derbl
        ON derbl.session_id = dowt.blocking_session_id
OUTER APPLY sys.dm_exec_sql_text(derbl.sql_handle) AS destbl
LEFT JOIN sys.dm_tran_locks AS dtlbl
        ON derbl.session_id = dtlbl.request_session_id;
```

To understand how to analyze a blocking scenario and the relevant information provided by the blocker script, consider the following example. First, create a test table.

```
DROP TABLE IF EXISTS dbo.BlockTest;
GO

CREATE TABLE dbo.BlockTest (C1 INT,
                            C2 INT,
                            C3 DATETIME);

INSERT INTO dbo.BlockTest
VALUES (11, 12, GETDATE()),
       (21, 22, GETDATE());
```

Now open three connections and run the following two queries concurrently. Once you run them, use the blocker script in the third connection. Execute the following code in one connection:

```
BEGIN TRAN User1
UPDATE  dbo.BlockTest
SET     C3 = GETDATE();
```

Next, execute this code while the `User1` transaction is executing:

```
BEGIN TRAN User2
SELECT  C2
FROM    dbo.BlockTest
WHERE   C1 = 11;
COMMIT
```

This creates a simple blocking scenario where the `User1` transaction blocks the `User2` transaction.

The output of the blocker script provides information immediately useful to begin resolving blocking issues. First, you can identify the specific session information, including the session ID of both the blocking and waiting sessions. You get an immediate resource description from the waiting resource, the wait type, and the length of time in milliseconds that the process has been waiting. It's that value that allows you to provide a filter to eliminate short-term blocks, which are part of normal processing.

The database name is supplied because blocking can occur anywhere in the system, not just in AdventureWorks2017. You'll want to identify it where it occurs. The resources and types from the basic locking information are retrieved for the waiting process.

The blocking request type is displayed, and both the waiting T-SQL and blocking T-SQL, if available, are displayed. Once you have the object where the block is occurring, having the T-SQL so that you can understand exactly where and how the process is either blocking or being blocked is a vital part of the process of eliminating or reducing the amount of blocking. All this information is available from one simple query. Figure 21-14 shows the sample output from the earlier blocked process.

Figure 21-14. *Output from the blocker script*

Be sure to go back to Connection 1 and commit or roll back the transaction.

Extended Events and the blocked_process_report Event

Extended Events provides an event called `blocked_process_report`. This event works off the blocked process threshold that you need to provide to the system configuration. This script sets the threshold to five seconds:

```
EXEC sp_configure 'show advanced option', '1';
RECONFIGURE;
EXEC sp_configure
    'blocked process threshold',
    5;
RECONFIGURE;
```

This would normally be a very low value in most systems. If you have an established performance service level agreement (SLA), you could use that as the threshold. Once the value is set, you can configure alerts so that e-mails, tweets, or instant messages are sent if any process is blocked longer than the value you set. It also acts as a trigger for the extended event. The default value for the `blocked process threshold` is zero, meaning that it never actually fires. If you are going to use Extended Events to track blocked processes, you will want to adjust this value from the default.

To set up a session that captures the `blocked_process_report`, first open the Extended Events session properties window. (Although you should use scripts to set up this event in a production environment, I'll show how to use the GUI.) Provide the session with a name and then navigate to the Events page. Type **block** into the "Event library" text box, which will find the `blocked_process_report` event. Select that event by clicking the right arrow. You should see something similar to Figure 21-15.

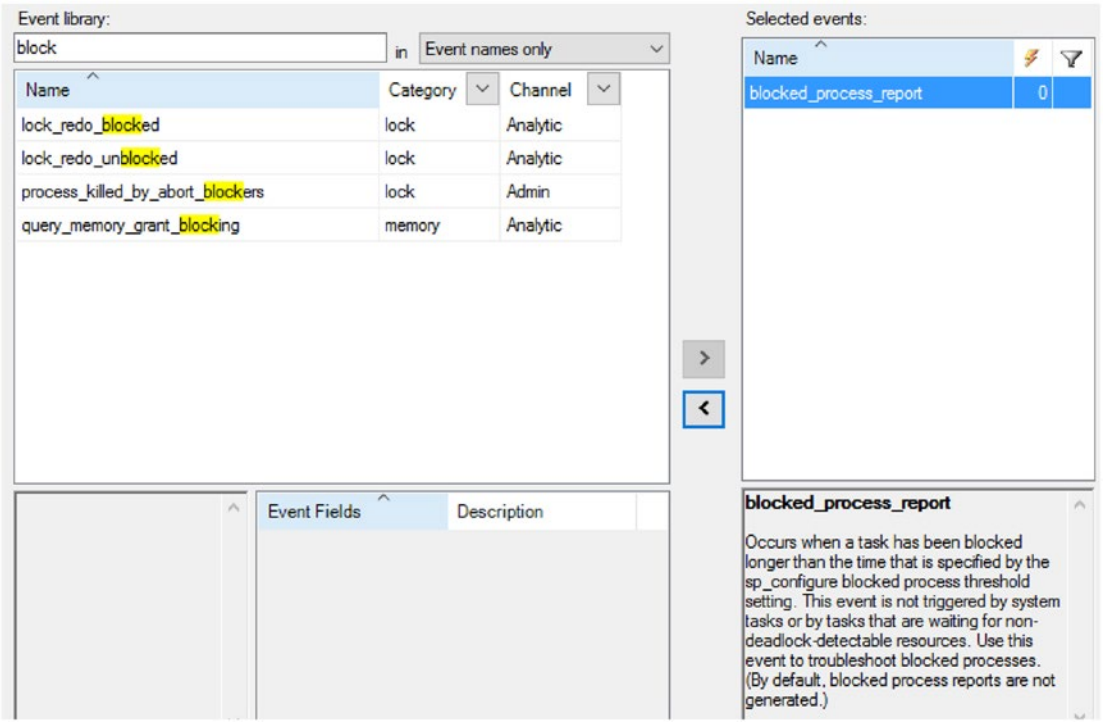
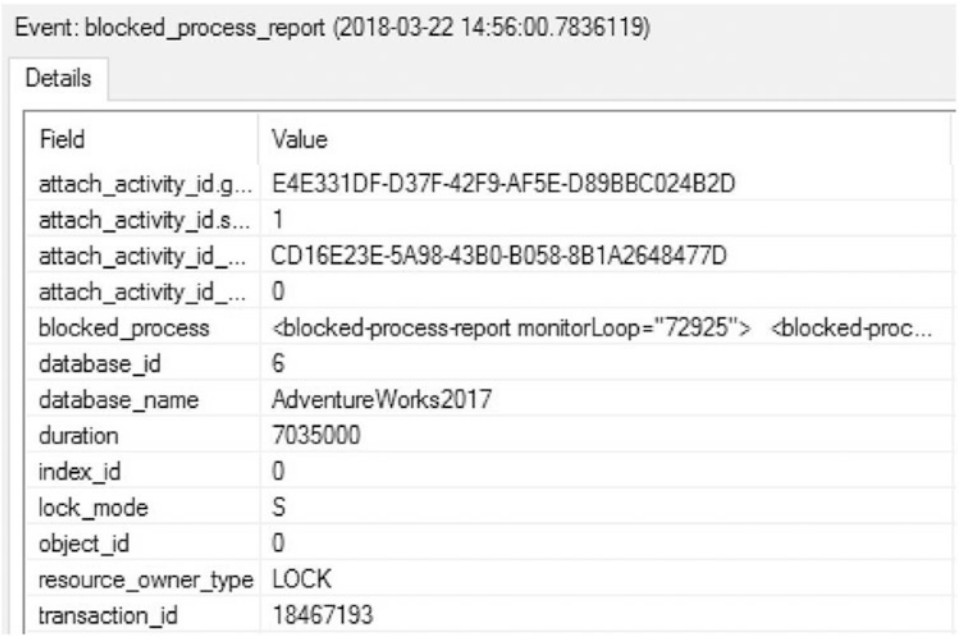


Figure 21-15. *The blocked process report event selected in the Extended Events window*

The event fields are all preselected for you. If you still have the queries running from the previous section that created the block, all you need to do now is click the Run button to capture the event. Otherwise, go back to the queries we used to generate the blocked process report in the previous section and run them in two different connections. After the blocked process threshold is passed, you'll see the event fire…and fire. It will fire every five seconds if that's how you've configured it and you're leaving the connections running. The output in the live data stream looks like Figure 21-16.


Event: blocked_process_report (2018-03-22 14:56:00.7836119)

Details

Field	Value
attach_activity_id.g...	E4E331DF-D37F-42F9-AF5E-D89BBC024B2D
attach_activity_id.s...	1
attach_activity_id_...	CD16E23E-5A98-43B0-B058-8B1A2648477D
attach_activity_id_...	0
blocked_process	<blocked-process-report monitorLoop="72925"> <blocked-proc...
database_id	6
database_name	AdventureWorks2017
duration	7035000
index_id	0
lock_mode	S
object_id	0
resource_owner_type	LOCK
transaction_id	18467193

Figure 21-16. *Output from the blocked_process_report event*

Some of the information is self-explanatory; to get into the details, you need to look at the XML generated in the `blocked_process` field.

```
<blocked-process-report monitorLoop="72925">
 <blocked-process>
  <process id="process1edc0b0c108" taskpriority="0" logused="0"
   waitresource="RID: 6:1:72793:0" waittime="7035" ownerId="18467193"
   transactionname="User2" lasttranstarted="2018-03-22T14:55:53.743"
   XDES="0x1edf1bc0490" lockMode="S" schedulerid="1" kpid="14036"
   status="suspended" spid="53" sbid="0" ecid="0" priority="0" trancount="1"
   lastbatchstarted="2018-03-22T14:55:53.743" lastbatchcompleted="2018-
   03-22T14:55:53.740" lastattention="1900-01-01T00:00:00.740"
   clientapp="Microsoft SQL Server Management Studio - Query" hostname="WIN-
   8A2LQANSO51" hostpid="5540" loginname="WIN-8A2LQANSO51\Administrator"
   isolationlevel="read committed (2)" xactid="18467193" currentdb="6"
   lockTimeout="4294967295" clientoption1="671090784" clientoption2="390200">
   <executionStack>
    <frame line="2" stmtstart="24" stmtend="118" sqlhandle="0x02000000ccf3e60
     45e680885750c3f36d7cc549d8ff01368000000000000000000000000000000000000000"/>
```

```
    <frame line="2" stmtstart="36" stmtend="134" sqlhandle="0x0200000063e12d309
     fa7874804b7b56c7be7beecf2a0255b0000000000000000000000000000000000000000"/>
   </executionStack>
   <inputbuf>
BEGIN TRAN User2
SELECT  C2
FROM    dbo.BlockTest
WHERE   C1 = 11;
COMMIT   </inputbuf>
  </process>
 </blocked-process>
 <blocking-process>
  <process status="sleeping" spid="62" sbid="0" ecid="0" priority="0"
   trancount="1" lastbatchstarted="2018-03-22T14:55:50.923"
   lastbatchcompleted="2018-03-22T14:55:50.927" lastattention="1900-01-
   01T00:00:00.927" clientapp="Microsoft SQL Server Management Studio -
   Query" hostname="WIN-8A2LQANSO51" hostpid="5540" loginname="WIN-
   8A2LQANSO51\Administrator" isolationlevel="read committed
   (2)" xactid="18467189" currentdb="6" lockTimeout="4294967295"
   clientoption1="671090784" clientoption2="390200">
   <executionStack />
   <inputbuf>
BEGIN TRAN User1
UPDATE  dbo.BlockTest
SET     C3 = GETDATE();   </inputbuf>
  </process>
 </blocking-process>
</blocked-process-report>
```

The elements are clear if you look through this XML. `<blocked-process>` shows information about the process that was blocked, including familiar information such as the session ID (labeled with the old-fashioned SPID here), the database ID, and so on. You can see the query in the `<inputbuf>` element. Details such as the `lockMode` are available within the `<process>` element. Note that the XML doesn't include some of the other information that you can easily get from T-SQL queries, such as the query string of the blocked and waiting processes. But with the SPID available, you can get them from

the cache, if available, or you can combine the Blocked Process report with other events such as `rpc_starting` to show the query information. However, doing so will add to the overhead of using those events long term within your database. If you know you have a blocking problem, this can be part of a short-term monitoring project to capture the necessary blocking information.

Blocking Resolutions

Once you've analyzed the cause of a block, the next step is to determine any possible resolutions. Here are a few techniques you can use to do this:

- Optimize the queries executed by blocking and blocked SPIDs.
- Decrease the isolation level.
- Partition the contended data.
- Use a covering index on the contended data.

Note A detailed list of recommendations to avoid blocking appears later in the chapter in the "Recommendations to Reduce Blocking" section.

To understand these resolution techniques, let's apply them in turn to the preceding blocking scenario.

Optimize the Queries

Optimizing the queries executed by the blocking and blocked processes helps reduce the blocking duration. In the blocking scenario, the queries executed by the processes participating in the blocking are as follows:

- Blocking process:

```
BEGIN TRAN User1
UPDATE  dbo.BlockTest
SET     C3 = GETDATE();
```

- Blocked process:

```
BEGIN TRAN User2
SELECT  C2
FROM    dbo.BlockTest
WHERE   C1 = 11;
COMMIT
```

Note that beyond the missing `COMMIT` for the first query, running `UPDATE` without a `WHERE` clause is certainly potentially problematic and will not perform well. It will get worse over time as the data scales. However, it is just a test for demonstration purposes.

Next, let's analyze the individual SQL statements executed by the blocking and blocked SPIDs to optimize their performance.

- The `UPDATE` statement of the blocking SPID accesses the data without a `WHERE` clause. This makes the query inherently costly on a large table. If possible, break the action of the `UPDATE` statement into multiple batches using appropriate `WHERE` clauses. Remember to try to use set-based operations such as a `TOP` statement to limit the rows. If the individual `UPDATE` statements of the batch are executed in separate transactions, then fewer locks will be held on the resource within one transaction and for shorter time periods. This could also help reduce or avoid lock escalation.
- The `SELECT` statement executed by the blocked SPID has a `WHERE` clause on the `C1` column. From the index structure on the test table, you can see that there is no index on this column. To optimize the `SELECT` statement, you could create a clustered index on the `C1` column.

```
CREATE CLUSTERED INDEX i1 ON dbo.BlockTest(C1);
```

Note Since the `example` table fits within one page, adding the clustered index won't make much difference to the query performance. However, as the number of rows in the table increases, the beneficial effect of the index will become more pronounced.

Optimizing the queries reduces the duration for which the locks are held by the processes. The query optimization reduces the impact of blocking, but it doesn't prevent the blocking completely. However, as long as the optimized queries execute within acceptable performance limits, a small amount of blocking may be ignored.

Decrease the Isolation Level

Another approach to resolve blocking can be to use a lower isolation level, if possible. The SELECT statement of the User2 transaction gets blocked while requesting an (S) lock on the data row. The isolation level of this transaction can be mitigated by taking advantage of SNAPSHOT isolation level Read Committed Snapshot so that the (S) lock is not requested by the SELECT statement. The Read Committed Snapshot isolation level can be configured for the connection using the SET statement.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
GO
BEGIN TRAN User2
SELECT  C2
FROM    dbo.BlockTest
WHERE   C1 = 11;
COMMIT
GO
--Back to default
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
GO
```

This example shows the utility of reducing the isolation level. Using this SNAPSHOT isolation is radically preferred over using any of the methods that produce dirty reads that could lead to incorrect data or missing or extra rows.

Partition the Contended Data

When dealing with large data sets or data that can be discretely stored, it is possible to apply table partitioning to the data. Partitioned data is split horizontally, that is, by certain values (such as splitting sales data up by month, for example). This allows the transactions to execute concurrently on the individual partitions, without blocking each

other. These separate partitions are treated as a single unit for querying, updating, and inserting; only the storage and access are separated by SQL Server. It should be noted that partitioning is available only in the Developer and Enterprise editions of SQL Server.

In the preceding blocking scenario, the data could be separated by date. This would entail setting up multiple filegroups if you're concerned with performance (or just put everything on `PRIMARY` if you're worried about management) and splitting the data per a defined rule. Once the `UPDATE` statement gets a `WHERE` clause, then it and the original `SELECT` statement will be able to execute concurrently on two separate partitions. This does require that the `WHERE` clause filters only on the partition key column. As soon as you get other conditions in the mix, you're unlikely to benefit from partition elimination, which means performance could be much worse, not better.

Note Partitioning, if done properly, can improve both performance and concurrency on large data sets. But, partitioning is almost exclusively a data management solution, not a performance tuning option.

In a blocking scenario, you should analyze whether the query of the blocking or the blocked process can be fully satisfied using a covering index. If the query of one of the processes can be satisfied using a covering index, then it will prevent the process from requesting locks on the contended resource. Also, if the other process doesn't need a lock on the covering index (to maintain data integrity), then both processes will be able to execute concurrently without blocking each other.

For instance, in the preceding blocking scenario, the `SELECT` statement by the blocked process can be fully satisfied by a covering index on the `C1` and `C2` columns.

```
CREATE NONCLUSTERED INDEX iAvoidBlocking ON dbo.BlockTest(C1,  C2) ;
```

The transaction of the blocking process need not acquire a lock on the covering index since it accesses only the `C3` column of the table. The covering index will allow the `SELECT` statement to get the values for the `C1` and `C2` columns without accessing the base table. Thus, the `SELECT` statement of the blocked process can acquire an (S) lock on the covering-index row without being blocked by the (X) lock on the data row acquired by the blocking process. This allows both transactions to execute concurrently without any blocking.

Consider a covering index as a mechanism to "duplicate" part of the table data in which consistency is automatically maintained by SQL Server. This covering index, if mostly read-only, can allow some transactions to be served from the "duplicate" data while the base table (and other indexes) can continue to serve other transactions. The trade-offs to this approach are the need for additional storage and the potential for additional overhead during data modification.

Recommendations to Reduce Blocking

Single-user performance and the ability to scale with multiple users are both important for a database application. In a multiuser environment, it is important to ensure that the database operations don't hold database resources for a long time. This allows the database to support a large number of operations (or database users) concurrently without serious performance degradation. The following is a list of tips to reduce/avoid database blocking:

- Keep transactions short.
  - Perform the minimum steps/logic within a transaction.
  - Do not perform costly external activity within a transaction, such as sending an acknowledgment e-mail or performing activities driven by the end user.
  - Optimize queries.
  - Create indexes as required to ensure optimal performance of the queries within the system.
  - Avoid a clustered index on frequently updated columns. Updates to clustered index key columns require locks on the clustered index and all nonclustered indexes (since their row locator contains the clustered index key).
  - Consider using a covering index to serve the blocked `SELECT` statements.

- Use query timeouts or a resource governor to control runaway queries. For more on the resource governor, consult Books Online: http://bit.ly/1jiPhfS.
  - Avoid losing control over the scope of the transactions because of poor error-handling routines or application logic.
  - Use `SET XACT_ABORT ON` to avoid a transaction being left open on an error condition within the transaction.
  - Execute the following SQL statement from a client error handler (`TRY/CATCH`) after executing a SQL batch or stored procedure containing a transaction.

    ```
    IF @@TRANCOUNT > 0 ROLLBACK
    ```

- Use the lowest isolation level required.
  - Consider using row versioning, one of the `SNAPSHOT` isolation levels, to help reduce contention.

Automation to Detect and Collect Blocking Information

In addition to capturing information using extended events, you can automate the process of detecting a blocking condition and collecting the relevant information using SQL Server Agent. SQL Server provides the Performance Monitor counters shown in Table 21-2 to track the amount of wait time.

Table 21-2. *Performance Monitor Counters*

Object	Counter	Instance	Description
`SQLServer:Locks` (for SOL Server named instance `MSSOL$<InstanceName>:Locks`)	`Average Wait Time(ms)`	`_Total`	Average amount of wait time for each lock that resulted in a wait
	`Lock Wait Time (ms)`	`_Total`	Total wait time for locks in the last second