The durability property isn't a direct cause of most blocking since it doesn't require the actions of a transaction to be isolated from those of others. But in an indirect way, it increases the duration of the blocking. Since the durability property requires saving the pre- and post-images of the data under modification to the transaction log on disk, it increases the duration of the transaction and therefore the possibility of blocking.

Introduced in SQL Server 2014 is the ability to reduce latency, the time waiting on a query to commit and write to the log, by modifying the durability behavior of a given database. You can now use delayed durability. This means that when a transaction completes, it reports immediately to the application as a successful transaction, reducing latency. But the writes to the log have not yet occurred. This may also allow for more transactions to be completed while still waiting on the system to write all the output to the transaction log. While this may increase apparent speed within the system, as well as possibly reducing contention on transaction log I/O, it's inherently a dangerous choice. This is a difficult recommendation to make. Microsoft suggests three possible situations that may make it attractive.

- *You don't care about the possible loss of some data*: Since you can be in a situation where you need to restore to a point in time from log backups, by choosing to put a database in delayed durability you may lose some data when you have to go to a restore situation.
- *You have a high degree of contention during log writes*: If you're seeing a lot of waits while transactions get written to the log, delayed durability could be a viable solution. But, you're also going to want to be tolerant of data loss, as discussed earlier.
- *You're experiencing high overall resource contention*: A lot of resource contention on the server comes down to the locks being held longer. If you're seeing lots of contention and you're seeing long log writes or also seeing contention on the log and you have a high tolerance for data loss, this may be a viable way to help reduce the system's contention.

In other works, I recommend using delayed durability only if you meet all those criteria, with the first being the most important. Also, don't forget about the changes to the checkpoint behavior noted earlier. If you're in a high-volume system, with lots of data changes, you may need to adjust the recovery interval to assist with system behavior as well.

Note Out of the four ACID properties, the `isolation` property, which is also used to ensure atomicity and consistency, is the main cause of blocking in a SQL Server database. In SQL Server, isolation is implemented using locks, as explained in the next section.

Locks

When a session executes a query, SQL Server determines the database resources that need to be accessed; and, if required, the lock manager grants different types of locks to the session. The query is blocked if another session has already been granted the locks; however, to provide both transaction isolation and concurrency, SQL Server uses different lock granularities, as explained in the sections that follow.

Lock Granularity

SQL Server databases are maintained as files on the physical disk. In the case of a traditional nondatabase file such as an Excel file on a desktop machine, the file may be written to by only one user at a time. Any attempt to write to the file by other users fails. However, unlike the limited concurrency on a nondatabase file, SQL Server allows multiple users to modify (or access) contents simultaneously, as long as they don't affect one another's data consistency. This decreases blocking and improves concurrency among the transactions.

To improve concurrency, SQL Server implements lock granularities at the following resource levels and in this order:

- Row (`RID`)
- Key (`KEY`)
- Page (`PAG`)
- Extent (`EXT`)
- Heap or B-tree (`HoBT`)
- Table (`TAB`)
- File (`FIL`)

- Application (APP)
- MetaData (MDT)
- Allocation Unit (AU)
- Database (DB)

Let's take a look at these lock levels in more detail.

Row-Level Lock

This lock is maintained on a single row within a table and is the lowest level of lock on a database table. When a query modifies a row in a table, an RID lock is granted to the query on the row. For example, consider the transaction on the following test table:

```
DROP TABLE IF EXISTS dbo.Test1;
CREATE TABLE dbo.Test1 (C1 INT);
INSERT INTO dbo.Test1
VALUES (1);
GO

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 dynamic management view sys.dm_tran_locks can be used to display the lock status. The query against sys.dm_tran_locks in Figure 21-1 shows that the DELETE statement acquired, among other locks, an exclusive RID lock on the row to be deleted.

	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