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(*)
```