

The estimated number of rows, at the bottom of Figure 20-9, is 3.05628. It's exactly the same as the calculation. Note, though, that the actual number of rows, at the top of Figure 20-9, is 48. This becomes important. If we look at the same properties on the same operator in the second plan, the one in Figure 20-8, we'll see that the estimated and actual number of rows are identical at 48. In this case, the optimizer decided that 48 rows returned was too many to be able to perform well through an Index Seek against the Product table. Instead, it opted to use an ordered scan (which you can verify through the properties of the Index Scan operator) and then a merge join.

Actual Number of Rows	48
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Defined Values	[Adventur
Description	Scan a par
Estimated CPU Cost	0.0001604
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	1
Estimated Number of Rows	3.05628

Figure 20-9. *Estimated versus actual number of rows*

In point of fact, the first plan was faster; however, it did result in higher I/O output. This is where we have to exercise caution. In this case, the performance was a little better, but if the system was under load, especially if it was under I/O strain, then the second plan is likely to perform faster, with fewer contentions on resources, since it has a lower number of reads overall. The caution comes from identifying which of these plans is better in particular circumstances.

To avoid this potential performance problem, use the following approach. Don't use a local variable as a filter criterion in a batch for a query like this. A local variable

is different from a parameter value, as demonstrated in Chapter 17. Create a stored procedure for the batch and execute it as follows:

```
CREATE OR ALTER PROCEDURE ProductDetails (@id INT)
AS
SELECT p.Name,
       p.ProductNumber,
       th.ReferenceOrderID
FROM Production.Product AS p
     JOIN Production.TransactionHistory AS th
       ON th.ProductID = p.ProductID
WHERE th.ReferenceOrderID = @id;
GO

EXEC ProductDetails @id = 1;
```

This approach can backfire. The process of using the values passed to a parameter is referred to as *parameter sniffing*. Parameter sniffing occurs for all stored procedures and parameterized queries automatically. Depending on the accuracy of the statistics and the values passed to the parameters, it is possible to get a bad plan using specific values and a good plan using the sampled values that occur when you have a local variable. Testing is the only way to be sure which will work best in any given situation. However, in most circumstances, you're better off having accurate values rather than sampled ones. For more details on parameter sniffing, see Chapter 17.

As a general guideline, it's best to avoid hard-coding values. If the values have to change, you may have to change them in a lot of code. If you do need to code values within your queries, local variables let you control them from a single location at the top of the batch, making the management of the code easier. However, local variables, as we've just seen, when used for data retrieval can affect plan choice. In that case, parameter values are preferred. You can even set the parameter value and provide it with a default value. These will still be sniffed as regular parameters.

Be Careful When Naming Stored Procedures

The name of a stored procedure does matter. You should not name your procedures with a prefix of `sp_`. Developers often prefix their stored procedures with `sp_` so that they can easily identify the stored procedures. However, SQL Server assumes that any stored procedure with this exact prefix is probably a system stored procedure, whose home is in the master database. When a stored procedure with an `sp_` prefix is submitted for execution, SQL Server looks for the stored procedure in the following places in the following order:

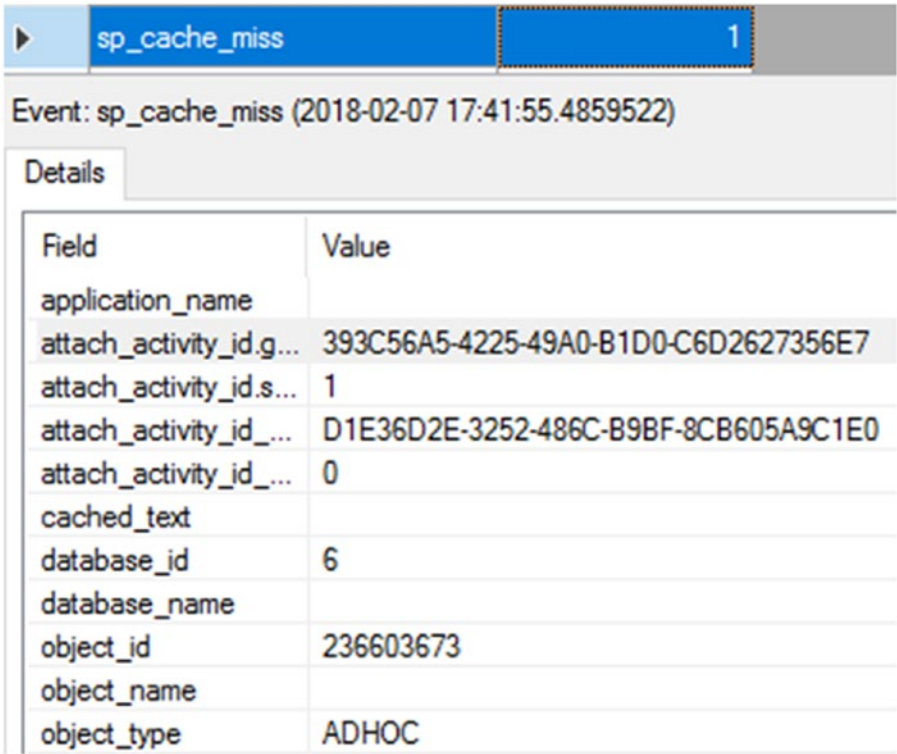
- In the master database
- In the current database based on any qualifiers provided (database name or owner)
- In the current database using `dbo` as the schema, if a schema is not specified

Therefore, although the user-created stored procedure prefixed with `sp_` exists in the current database, the master database is checked first. This happens even when the stored procedure is qualified with the database name.

To understand the effect of prefixing `sp_` to a stored procedure name, consider the following stored procedure:

```
IF EXISTS ( SELECT *
            FROM sys.objects
            WHERE object_id = OBJECT_ID(N'[dbo].[sp_Dont]')
              AND type IN (N'P', N'PC') )
    DROP PROCEDURE [dbo].[sp_Dont]
GO
CREATE PROC [sp_Dont]
AS
PRINT 'Done!'
GO
--Add plan of sp_Dont to procedure cache
EXEC AdventureWorks2017.dbo.[sp_Dont] ;
GO
--Use the above cached plan of sp_Dont
EXEC AdventureWorks2012.dbo.[sp_Dont] ;
GO
```

The first execution of the stored procedure adds the execution plan of the stored procedure to the procedure cache. A subsequent execution of the stored procedure reuses the existing plan from the procedure cache unless a recompilation of the plan is required (the causes of stored procedure recompilation are explained in Chapter 10). Therefore, the second execution of the stored procedure `spDont` shown in Figure 20-10 should find a plan in the procedure cache. This is indicated by an `SP:CacheHit` event in the corresponding Extended Events output.



Field	Value
application_name	
attach_activity_id.g...	393C56A5-4225-49A0-B1D0-C6D2627356E7
attach_activity_id.s...	1
attach_activity_id_...	D1E36D2E-3252-486C-B9BF-8CB605A9C1E0
attach_activity_id_...	0
cached_text	
database_id	6
database_name	
object_id	236603673
object_name	
object_type	ADHOC

Figure 20-10. Extended Events output showing the effect of the `sp_` prefix on a stored procedure name

Note that an `SP:CacheMiss` event is fired before SQL Server tries to locate the plan for the stored procedure in the procedure cache. The `SP:CacheMiss` event is caused by SQL Server looking in the master database for the stored procedure, even though the execution of the stored procedure is properly qualified with the user database name.

This aspect of the `sp_` prefix becomes more interesting when you create a stored procedure with the name of an existing system stored procedure.

```

CREATE OR ALTER PROC sp_addmessage @param1 NVARCHAR(25)
AS
PRINT '@param1 = ' + @param1 ;
GO

EXEC AdventureWorks2017.dbo.[sp_addmessage] 'AdventureWorks';

```

The execution of this user-defined stored procedure causes the execution of the system stored procedure `sp_addmessage` from the master database instead, as you can see in Figure 20-11.

**Msg 8114, Level 16, State 5, Procedure sp_addmessage, Line 4009
Error converting data type varchar to int.**

Figure 20-11. Execution result for stored procedure showing the effect of the *sp_* prefix on a stored procedure name

Unfortunately, it is not possible to execute this user-defined stored procedure. You can see now why you should not prefix a user-defined stored procedure's name with `sp_`. Use some other naming convention. From a pure performance standpoint, this is a trivial improvement. However, if you have high volume and response time is critical, it is one more small point in your favor if you avoid the `sp_` naming standard.

Reducing the Number of Network Round-Trips

Database applications often execute multiple queries to implement a database operation. Besides optimizing the performance of the individual query, it is important that you optimize the performance of the batch. To reduce the overhead of multiple network round-trips, consider the following techniques:

- Execute multiple queries together.
- Use SET NOCOUNT.

Let's look at these techniques in a little more depth.

Execute Multiple Queries Together

It is preferable to submit all the queries of a set together as a batch or a stored procedure. Besides reducing the network round-trips between the database application and the server, stored procedures also provide multiple performance and administrative benefits, as described in Chapter 16. This means the code in the application needs to be able to deal with multiple result sets. It also means your T-SQL code may need to deal with XML data or other large sets of data, not single-row inserts or updates.

Use SET NOCOUNT

You need to consider one more factor when executing a batch or a stored procedure. After every query in the batch or the stored procedure is executed, the server reports the number of rows affected.

```
(<Number> row(s) affected)
```

This information is returned to the database application and adds to the network overhead. Use the T-SQL statement `SET NOCOUNT` to avoid this overhead.

```
SET NOCOUNT ON <SQL queries> SET NOCOUNT OFF
```

Note that the `SET NOCOUNT` statement doesn't cause any recompilation issue with stored procedures, unlike some `SET` statements, as explained in Chapter 18.

Reducing the Transaction Cost

Every action query in SQL Server is performed as an *atomic* action so that the state of a database table moves from one *consistent* state to another. SQL Server does this automatically, and it can't be disabled. If the transition from one consistent state to another requires multiple database queries, then atomicity across the multiple queries should be maintained using explicitly defined database transactions. The old and new states of every atomic action are maintained in the transaction log (on the disk) to ensure *durability*, which guarantees that the outcome of an atomic action won't be lost once it completes successfully. An atomic action during its execution is *isolated* from other database actions using database locks.

Based on the characteristics of a transaction, here are two broad recommendations to reduce the cost of the transaction:

- Reduce logging overhead.
- Reduce lock overhead.

Reduce Logging Overhead

A database query may consist of multiple data manipulation queries. If atomicity is maintained for each query separately, then a large number of disk writes are performed on the transaction log. Since disk activity is extremely slow compared to memory or CPU activity, the excessive disk activity can increase the execution time of the database functionality. For example, consider the following batch query:

```
--Create a test table
IF (SELECT OBJECT_ID('dbo.Test1')
    ) IS NOT NULL
    DROP TABLE dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 TINYINT);
GO

--Insert 10000 rows
DECLARE @Count INT = 1;
WHILE @Count <= 10000
    BEGIN
        INSERT INTO dbo.Test1
            (C1)
        VALUES (@Count % 256);
        SET @Count = @Count + 1;
    END
```

Since every execution of the INSERT statement is atomic in itself, SQL Server will write to the transaction log for every execution of the INSERT statement.

An easy way to reduce the number of log disk writes is to include the action queries within an explicit transaction.

```
DECLARE @Count INT = 1;
DBCC SQLPERF(LOGSPACE);
BEGIN TRANSACTION
WHILE @Count <= 10000
    BEGIN
        INSERT INTO dbo.Test1
            (C1)
        VALUES (@Count % 256) ;
        SET @Count = @Count + 1 ;
    END
COMMIT
DBCC SQLPERF(LOGSPACE);
```

The defined transaction scope (between the BEGIN TRANSACTION and COMMIT pair of commands) expands the scope of atomicity to the multiple INSERT statements included within the transaction. This decreases the number of log disk writes and improves the performance of the database functionality. To test this theory, run the following T-SQL command before and after each of the WHILE loops:

```
DBCC SQLPERF(LOGSPACE);
```

This will show you the percentage of log space used. On running the first set of inserts on my database, the log went from 3.2 percent used to 3.3 percent. When running the second set of inserts, the log grew about 6 percent.

The best way is to work with sets of data rather than individual rows. A WHILE loop can be an inherently costly operation, like a cursor (more details on cursors in [Chapter 23](#)). So, running a query that avoids the WHILE loop and instead works from a set-based approach is even better.

```
SELECT TOP 10000
    IDENTITY(INT, 1, 1) AS n
INTO #Tally
FROM master.dbo.syscolumns AS sc1,
    master.dbo.syscolumns AS sc2;
DBCC SQLPERF(LOGSPACE);
```



```
BEGIN TRANSACTION
INSERT INTO dbo.Test1 (C1)
SELECT TOP 1000
      (n % 256)
FROM #Tally AS t
COMMIT
```

Running this query with the `DBCC SQLPERF()` function before and after showed less than .01 percent growth of the used space within the log, and it ran in 41ms as compared to more than 2s for the `WHILE` loop.

One area of caution, however, is that by including too many data manipulation queries within a transaction, the duration of the transaction is increased. During that time, all other queries trying to access the resources referred to in the transaction are blocked. Rollback duration and recovery time during a restore increase because of long transactions.

Reduce Lock Overhead

By default, all four SQL statements (`SELECT`, `INSERT`, `UPDATE`, and `DELETE`) use database locks to isolate their work from that of other SQL statements. This lock management adds performance overhead to the query. The performance of a query can be improved by requesting fewer locks. By extension, the performance of other queries are also improved because they have to wait a shorter period of time to obtain their own locks.

By default, SQL Server can provide row-level locks. For a query working on a large number of rows, requesting a row lock on all the individual rows adds a significant overhead to the lock-management process. You can reduce this lock overhead by decreasing the lock granularity, say to the page level or table level. SQL Server performs the lock escalation dynamically by taking into consideration the lock overheads. Therefore, generally, it is not necessary to manually escalate the lock level. But, if required, you can control the concurrency of a query programmatically using lock hints as follows:

```
SELECT * FROM <TableName> WITH(PAGLOCK) --Use page level lock
```

Similarly, by default, SQL Server uses locks for SELECT statements besides those for INSERT, UPDATE, and DELETE statements. This allows the SELECT statements to read data that isn't being modified. In some cases, the data may be quite static, and it doesn't go through much modification. In such cases, you can reduce the lock overhead of the SELECT statements in one of the following ways:

- Mark the database as READONLY.

```
ALTER DATABASE <DatabaseName> SET READ_ONLY
```

This allows users to retrieve data from the database, but it prevents them from modifying the data. The setting takes effect immediately. If occasional modifications to the database are required, then it may be temporarily converted to READWRITE mode.

```
ALTER DATABASE <DatabaseName> SET READ_WRITE
```

```
<Database modifications>
```

```
ALTER DATABASE <DatabaseName> SET READONLY
```

- Use one of the snapshot isolations.

SQL Server provides a mechanism to put versions of data into tempdb as updates are occurring, radically reducing locking overhead and blocking for read operations. You can change the isolation level of the database by using an ALTER statement.

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

- Prevent SELECT statements from requesting any lock.

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

This prevents the SELECT statement from requesting any lock, and it is applicable to SELECT statements only. Although the NOLOCK hint can't be used directly on the tables referred to in the action queries (INSERT, UPDATE, and DELETE), it may be used on the data retrieval part of the action queries, as shown here:

```
DELETE Sales.SalesOrderDetail
FROM Sales.SalesOrderDetail AS sod WITH (NOLOCK)
JOIN Production.Product AS p WITH (NOLOCK)
ON sod.ProductID = p.ProductID
AND p.ProductID = 0;
```

Just know that this leads to dirty reads, which can cause duplicate rows or missing rows and is therefore considered to be a last resort to control locking. In fact, this is considered to be quite dangerous and will lead to improper results. The best approach is to mark the database as read-only or use one of the snapshot isolation levels.

This is a huge topic, and a lot more can be said about it. I discuss the different types of lock requests and how to manage lock overhead in the next chapter. If you made any of the proposed changes to the database from this section, I recommend restoring from a backup.

Summary

As discussed in this chapter, to improve the performance of a database application, it is important to ensure that SQL queries are designed properly to benefit from performance enhancement techniques such as indexes, stored procedures, database constraints, and so on. Ensure that queries are resource friendly and don't prevent the use of indexes. In many cases, the optimizer has the ability to generate cost-effective execution plans irrespective of query structure, but it is still a good practice to design the queries properly in the first place. Even after you design individual queries for great performance, the overall performance of a database application may not be satisfactory. It is important not only to improve the performance of individual queries but also to ensure that they work well with other queries without causing serious blocking issues. In the next chapter, you will look into the different blocking aspects of a database application.

CHAPTER 21

Blocking and Blocked Processes

You would ideally like your database application to scale linearly with the number of database users and the volume of data. However, it is common to find that performance degrades as the number of users increases and as the volume of data grows. One cause for degradation, especially associated with ever-increasing scale, is blocking. In fact, database blocking is usually one of the biggest enemies of scalability for database applications.

In this chapter, I cover the following topics:

- The fundamentals of blocking in SQL Server
- The ACID properties of a transactional database
- Database lock granularity, escalation, modes, and compatibility
- ANSI isolation levels
- The effect of indexes on locking
- The information necessary to analyze blocking
- A SQL script to collect blocking information
- Resolutions and recommendations to avoid blocking
- Techniques to automate the blocking detection and information collection processes

Blocking Fundamentals

In an ideal world, every SQL query would be able to execute concurrently, without any blocking by other queries. However, in the real world, queries *do* block each other, similar to the way a car crossing through a green traffic signal at an intersection blocks other cars waiting to cross the intersection. In SQL Server, this traffic management takes the form of the *lock manager*, which controls concurrent access to a database resource to maintain data consistency. The concurrent access to a database resource is controlled across multiple database connections.

I want to make sure things are clear before moving on. Three terms are used within databases that sound the same and are interrelated but have different meanings. These are frequently confused, and people often use the terms incorrectly and interchangeably. These terms are *locking*, *blocking*, and *deadlocking*. Locking is an integral part of the process of SQL Server managing multiple sessions. When a session needs access to a piece of data, a lock of some type is placed on it. This is different from blocking, which is when one session, or thread, needs access to a piece of data and has to wait for another session's lock to clear. Finally, deadlocking is when two sessions, or threads, form what is sometimes referred to as a *deadly embrace*. They are each waiting on the other for a lock to clear. Deadlocking could also be referred to as a permanent blocking situation, but it's one that won't resolve by waiting any period of time. Deadlocking will be covered in more detail in Chapter 22. So, locks can lead to blocks, and both locks and blocks play a part in deadlocks, but these are three distinct concepts. Please understand the differences between these terms and use them correctly. It will help in your understanding of the system, your ability to troubleshoot, and your ability to communicate with other database administrators and developers.

In SQL Server, a database connection is identified by a session ID. Connections may be from one or many applications and one or many users on those applications; as far as SQL Server is concerned, every connection is treated as a separate session. Blocking between two sessions accessing the same piece of data at the same time is a natural phenomenon in SQL Server. Whenever two sessions try to access a common database resource in conflicting ways, the lock manager ensures that the second session waits until the first session completes its work in conjunction with the management of transactions within the system. For example, a session might be modifying a table record while another session tries to delete the record. Since these two data access requests are incompatible, the second session will be blocked until the first session completes its task.

On the other hand, if the two sessions try to read a table concurrently, both requests are allowed to execute without blocking, since these data access requests are compatible with each other.

Usually, the effect of blocking on a session is quite small and doesn't affect its performance noticeably. At times, however, because of poor query and/or transaction design (or maybe bad luck), blocking can affect query performance significantly. In a database application, every effort should be made to minimize blocking and thereby increase the number of concurrent users who can use the database.

With the introduction of in-memory tables in SQL Server 2014, locking, at least for these tables, takes on whole new dimensions. I'll cover their behavior separately in Chapter [24](#).

Understanding Blocking

In SQL Server, a database query can execute as a logical unit of work in itself, or it can participate in a bigger logical unit of work. A bigger logical unit of work can be defined using the `BEGIN TRANSACTION` statement along with `COMMIT` and/or `ROLLBACK` statements. Every logical unit of work must conform to a set of four properties called *ACID* properties:

- Atomicity
- Consistency
- Isolation
- Durability

I cover these properties in the sections that follow because understanding how transactions work is fundamental to understanding blocking.

Atomicity

A logical unit of work must be *atomic*. That is, either all the actions of the logical unit of work are completed or no effect is retained. To understand the atomicity of a logical unit of work, consider the following example:

```
USE AdventureWorks2017;
GO
DROP TABLE IF EXISTS dbo.ProductTest;
GO
```

```

CREATE TABLE dbo.ProductTest (ProductID INT
                                CONSTRAINT ValueEqualsOne CHECK
(ProductID = 1));
GO
--All ProductIDs are added into ProductTest as a logical unit of work
INSERT INTO dbo.ProductTest
SELECT p.ProductID
FROM Production.Product AS p;
GO
SELECT pt.ProductID
FROM dbo.ProductTest AS pt; --Returns 0 rows

```

SQL Server treats the preceding INSERT statement as a logical unit of work. The CHECK constraint on column ProductID of the dbo.ProductTest table allows only the value of 1. Although the ProductID column in the Production.Product table starts with the value of 1, it also contains other values. For this reason, the INSERT statement won't add any records at all to the dbo.ProductTest table, and an error is raised because of the CHECK constraint. Thus, atomicity is automatically ensured by SQL Server.

So far, so good. But in the case of a bigger logical unit of work, you should be aware of an interesting behavior of SQL Server. Imagine that the previous insert task consists of multiple INSERT statements. These can be combined to form a bigger logical unit of work, as follows:

```

BEGIN TRAN
--Start: Logical unit of work
--First:
INSERT INTO dbo.ProductTest
        SELECT p.ProductID
        FROM Production.Product AS p;
--Second:
INSERT INTO dbo.ProductTest
VALUES (1);
COMMIT --End: Logical unit of work
GO

```

With the `dbo.ProductTest` table already created in the preceding script, the `BEGIN TRAN` and `COMMIT` pair of statements defines a logical unit of work, suggesting that all the statements within the transaction should be atomic in nature. However, the default behavior of SQL Server doesn't ensure that the failure of one of the statements within a user-defined transaction scope will undo the effect of the prior statements. In the preceding transaction, the first `INSERT` statement will fail as explained earlier, whereas the second `INSERT` is perfectly fine. The default behavior of SQL Server allows the second `INSERT` statement to execute, even though the first `INSERT` statement fails. A `SELECT` statement, as shown in the following code, will return the row inserted by the second `INSERT` statement:

```
SELECT *
FROM    dbo.ProductTest; --Returns a row with t1.c1 = 1
```

The atomicity of a user-defined transaction can be ensured in the following two ways:

- `SET XACT_ABORT ON`
- Explicit rollback

Let's look at these briefly.

SET XACT_ABORT ON

You can modify the atomicity of the `INSERT` task in the preceding section using the `SET XACT_ABORT ON` statement.

```
SET XACT_ABORT ON;
GO
BEGIN TRAN
    --Start: Logical unit of work
    --First:
    INSERT INTO dbo.ProductTest
        SELECT p.ProductID
        FROM    Production.Product AS p;
    --Second:
    INSERT INTO dbo.ProductTest
    VALUES (1);
COMMIT
```



```
--End: Logical unit of work GO
SET XACT_ABORT OFF;
GO
```

The SET XACT_ABORT statement specifies whether SQL Server should automatically roll back and abort an entire transaction when a statement within the transaction fails. The failure of the first INSERT statement will automatically suspend the entire transaction, and thus the second INSERT statement will not be executed. The effect of SET XACT_ABORT is at the connection level, and it remains applicable until it is reconfigured or the connection is closed. By default, SET XACT_ABORT is OFF.

Explicit Rollback

You can also manage the atomicity of a user-defined transaction by using the TRY/CATCH error-trapping mechanism within SQL Server. If a statement within the TRY block of code generates an error, then the CATCH block of code will handle the error. If an error occurs and the CATCH block is activated, then the entire work of a user-defined transaction can be rolled back, and further statements can be prevented from execution, as follows:

```
BEGIN TRY
    BEGIN TRAN
    --Start: Logical unit of work
    --First:
    INSERT INTO dbo.ProductTest
    SELECT p.ProductID
    FROM Production.Product AS p

    Second:
    INSERT INTO dbo.ProductTest (ProductID)
    VALUES (1)
    COMMIT --End: Logical unit of work
END TRY
BEGIN CATCH
    ROLLBACK
    PRINT 'An error occurred'
    RETURN
END CATCH
```

The ROLLBACK statement rolls back all the actions performed in the transaction until that point. For a detailed description of how to implement error handling in SQL Server-based applications, please refer to the MSDN Library article titled “Using TRY...CATCH in Transact SQL” (<http://bit.ly/PN1AHF>).

Since the atomicity property requires that either all the actions of a logical unit of work are completed or no effects are retained, SQL Server *isolates* the work of a transaction from that of others by granting it exclusive rights on the affected resources. This means the transaction can safely roll back the effect of all its actions, if required. The exclusive rights granted to a transaction on the affected resources block all other transactions (or database requests) trying to access those resources during that time period. Therefore, although atomicity is required to maintain the integrity of data, it introduces the undesirable side effect of blocking.

Consistency

A unit of work should cause the state of the database to travel from one *consistent* state to another. At the end of a transaction, the state of the database should be fully consistent. SQL Server always ensures that the internal state of the databases is correct and valid by automatically applying all the constraints of the affected database resources as part of the transaction. SQL Server ensures that the state of internal structures, such as data and index layout, are correct after the transaction. For instance, when the data of a table is modified, SQL Server automatically identifies all the indexes, constraints, and other dependent objects on the table and applies the necessary modifications to all the dependent database objects as part of the transaction. That means that SQL Server will maintain the physical consistency of the data and the objects.

The logical consistency of the data is defined by the business rules and should be put in place by the developer of the database. A business rule may require changes to be applied on multiple tables, certain types of data to be restricted, or any number of other requirements. The database developer should accordingly define a logical unit of work to ensure that all the criteria of the business rules are taken care of. Further, the developer will ensure that the appropriate constructs are put in place to support the business rules that have been defined. SQL Server provides different transaction management features that the database developer can use to ensure the logical consistency of the data.

So, SQL Server works with the logical, business-defined, constraints that ensure a business-oriented data consistency to create a physical consistency on the underlying structures. The consistency characteristic of the logical unit of work blocks all other

transactions (or database requests) trying to access the affected objects during that time period. Therefore, even though consistency is required to maintain a valid logical and physical state of the database, it also introduces the same side effect of blocking.

Isolation

In a multiuser environment, more than one transaction can be executed simultaneously. These concurrent transactions should be isolated from one another so that the intermediate changes made by one transaction don't affect the data consistency of other transactions. The degree of *isolation* required by a transaction can vary. SQL Server provides different transaction isolation features to implement the degree of isolation required by a transaction.

Note Transaction isolation levels are explained later in the chapter in the “Isolation Levels” section.

The isolation requirements of a transaction operating on a database resource can block other transactions trying to access the resource. In a multiuser database environment, multiple transactions are usually executed simultaneously. It is imperative that the data modifications made by an ongoing transaction be protected from the modifications made by other transactions. For instance, suppose a transaction is in the middle of modifying a few rows in a table. During that period, to maintain database consistency, you must ensure that other transactions do not modify or delete the same rows. SQL Server logically isolates the activities of a transaction from that of others by blocking them appropriately, which allows multiple transactions to execute simultaneously without corrupting one another's work.

Excessive blocking caused by isolation can adversely affect the scalability of a database application. A transaction may inadvertently block other transactions for a long period of time, thereby hurting database concurrency. Since SQL Server manages isolation using locks, it is important to understand the locking architecture of SQL Server. This helps you analyze a blocking scenario and implement resolutions.

Note The fundamentals of database locks are explained later in the chapter in the “Capturing Blocking Information” section.

Durability

Once a transaction is completed, the changes made by the transaction should be *durable*. Even if the electrical power to the machine is tripped off immediately after the transaction is completed, the effect of all actions within the transaction should be retained. SQL Server ensures durability by keeping track of all pre- and post-images of the data under modification in a transaction log as the changes are made. Immediately after the completion of a transaction, SQL Server ensures that all the changes made by the transaction are retained—even if SQL Server, the operating system, or the hardware fails (excluding the log disk). During restart, SQL Server runs its database recovery feature, which identifies the pending changes from the transaction log for completed transactions and applies them to the database resources. This database feature is called *roll forward*.

The recovery interval period depends on the number of pending changes that need to be applied to the database resources during restart. To reduce the recovery interval period, SQL Server intermittently applies the intermediate changes made by the running transactions as configured by the recovery interval option. The recovery interval option can be configured using the `sp_configure` statement. The process of intermittently applying the intermediate changes is referred to as the *checkpoint* process. During restart, the recovery process identifies all uncommitted changes and removes them from the database resources by using the pre-images of the data from the transaction log.

Starting with SQL Server 2016, the default value of the `TARGET_RECOVERY_TIME` has been changed from 0, which means that the database will be doing all automatic checkpoints, to one minute. The default interval for automatic is also one minute, but now, the control is being set through the `TARGET_RECOVERY_TIME` value by default. If you need to change the frequency of the checkpoint operation, use `sp_configure` to change the recovery interval value. Setting this value means that the database is using indirect checkpoints. Instead of relying on the automatic checkpoints, you can use indirect checkpoints. This is a method to basically make the checkpoints occur all the time in order to meet the recovery interval. For systems with an extremely high number of data modifications, you might see high I/O because of indirect checkpoints. Starting in SQL Server 2016, all new databases created are automatically using indirect checkpoints because the `TARGET_INTERVAL_TIME` has been set. Any databases migrated from previous versions will be using whichever checkpoint method they had in that previous version. You may want to change their behavior as well. Using indirect checkpoints can result, for most systems, in a more consistent checkpoint behavior and faster recovery.