Transactions, Locking, Blocking, and Deadlocking

This chapter explores the concepts of transactions and locking. You will learn how transactions work, and how they are managed and used. Locking is also examined, covering the different types of locks in a database and how they interact with each other (lock compatibility). Locking issues, such as long-term locking (called blocking), and deadlocking are explained. You will learn techniques for identifying and troubleshooting blocking and deadlocking problems.

Understanding Transactions

Transactions are an integral part of a relational database system. A transaction defines a single unit of work; this unit of work can include one or more Transact-SQL statements. In a transaction, all statements are applied, or none at all. Let's review an example of a book inventory system; in this scenario, when a book is purchased, two tables are impacted: BookInventory and BookSales:

<table>
<thead>
<tr>
<th>BookInventory</th>
<th>BookSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>iBookID</td>
<td>iOrderID</td>
</tr>
<tr>
<td>numQuantity</td>
<td>iBookID</td>
</tr>
<tr>
<td></td>
<td>dtSalesDate</td>
</tr>
</tbody>
</table>

If a book is sold, numQuantity is reduced in the BookInventory table. A new entry is also made to the BookSales table, documenting order number, ID of the book sold, and the date of the sale:
Transactions are used within SQL Server to prevent partial updates. A partial update occurs when one part of an interrelated process is rolled back or cancelled without rolling back all logically necessary updates:

A transaction protects a unit of work from partial updates; if one data modification operation is unsuccessful, or is cancelled, the entire transaction (all data modifications) is rolled back. This ensures database consistency.

At the beginning of a transaction, SQL Server writes any changes that the Transact-SQL statements issue to the transaction log file. If any statement fails, then the entire transaction is rolled back and none of the data modifications are made. If the transaction succeeds, the transaction is committed and modifications are made.

A transaction is bound by the ACID test. ACID stands for Atomicity, Consistency, Isolation (or Independence), and Durability:
Atomicity
An atomic transaction is an all-or-nothing entity, which carries out all its steps or none at all.

Consistency
In a consistent transaction, no data modifications should break the structural or design rules of the database.

Isolation (or Independence)
Isolated (or independent) transactions do not depend on other transactions being run concurrently.

Durability
A durable transaction has permanent effect after it has been committed; its changes should survive system failures.

Implicit Transactions
Transactions can be explicit or implicit. Implicit transactions occur when the SQL Server session is in implicit transaction mode. While a session is in implicit transaction mode, a new transaction is created automatically after any of the following statements are executed:

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

The transaction does not complete until a COMMIT or ROLLBACK statement is issued. After one of these keywords is issued, a new transaction begins when any of the aforementioned statements is executed.

Implicit transaction mode is enabled when SET IMPLICIT_TRANSACTIONS ON is executed. The default is OFF. If ANSI_DEFAULTS is ON, however, SET IMPLICIT_TRANSACTIONS is enabled. Got that? Let's organize it in a table:

<table>
<thead>
<tr>
<th>State</th>
<th>Implicit Transactions Enabled?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Server default</td>
<td>No</td>
</tr>
<tr>
<td>SET IMPLICIT_TRANSACTIONS ON</td>
<td>Yes</td>
</tr>
<tr>
<td>SET ANSI_DEFAULTS ON</td>
<td>Yes</td>
</tr>
</tbody>
</table>
You can see which user options are enabled by running `DBCC USEROPTIONS`; if the `SET ANSI_DEFAULTS` or `IMPLICIT_TRANSACTIONS` options appear in the result set, then the option is ON:

In the following example, we `SET IMPLICIT_TRANSACTIONS ON`, and create a table called `t_TrnExample`:

![Example SQL command and output]

The command completes successfully. This message is misleading, however, as the transaction itself has not been committed. If you try closing the Query Analyzer window, you will see the following warning:

![Warning dialog]

You can select Yes to commit the transaction, or No to rollback. If you select No, the transaction's updates will be undone.

Avoid using implicit transactions if possible, as they make it easier for connections to leave uncommitted transactions out in the database, holding locks on resources and reducing concurrency. Implicit transactions are useful for ensuring that database users are sure of any changes they make to the database; the user must make a decision as to committing or rolling back their transaction(s).
Explicit Transactions

Explicit transactions are those that you define yourself. Explicit transactions use the following Transact-SQL commands and keywords:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN TRANSACTION</td>
<td>Sets the starting point of an explicit transaction.</td>
</tr>
<tr>
<td>ROLLBACK TRANSACTION</td>
<td>Restores original data modified by a transaction, to the state it was in at the start of the transaction. Resources held by the transaction are freed.</td>
</tr>
<tr>
<td>COMMIT TRANSACTION</td>
<td>Ends the transaction if no errors were encountered and makes changes permanent. Resources held by the transaction are freed.</td>
</tr>
<tr>
<td>BEGIN DISTRIBUTED TRANSACTION</td>
<td>Allows you to define the beginning of a distributed transaction to be managed by Microsoft Distributed Transaction Coordinator (MS DTC). MS DTC must be running locally and remotely.</td>
</tr>
<tr>
<td>SAVE TRANSACTION</td>
<td>SAVE TRANSACTION issues a savepoint within a transaction, which allows you to define a location to which a transaction can return if part of the transaction is cancelled. A transaction must be rolled back or committed immediately after rolling back to a savepoint.</td>
</tr>
<tr>
<td>@@TRANCOUNT</td>
<td>Returns the number of active transactions for the connection. BEGIN TRANSACTION increments @@TRANCOUNT by 1, and ROLLBACK TRANSACTION and COMMIT TRANSACTION decrements @@TRANCOUNT by 1. ROLLBACK TRANSACTION to a savepoint has no impact.</td>
</tr>
</tbody>
</table>

Example: Simple explicit transaction

The following is an example of a simple transaction using BEGIN TRANSACTION and COMMIT TRANSACTION. We include two INSERT statements, one to the t_Orders table, and the other to the t_OrdersHistory table. This activity benefits from an explicit transaction, as both statements have a logical dependency on each other. If there is a system failure after the first INSERT occurs but before the second, the first INSERT will be rolled back on database recovery:

```
BEGIN TRANSACTION
INSERT t_Orders (orderId) VALUES (1)
INSERT t_OrdersHistory (orderId, doOrderDate) VALUES (1, GETDATE())
COMMIT TRANSACTION
```

(1 row(s) affected)

(1 row(s) affected)
Example: Named transaction

Naming a transaction improves code readability and helps clarify nested transactions. The syntax for BEGIN TRANSACTION from SQL Server Books Online is as follows:

```
BEGIN TRAN [ SACTION ] [ transaction_name | @tran_name_variable
[ WITH MARK [ 'description' ] ] ]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN TRAN [ SACTION ]</td>
<td>You can use either BEGIN TRAN or BEGIN TRANSACTION.</td>
</tr>
<tr>
<td>transaction_name</td>
<td>transaction_name is the name of the transaction, and can be up to 32 characters in length.</td>
</tr>
<tr>
<td>@tran_name_variable</td>
<td>@tran_name_variable is the option of using a valid variable (CHAR, VARCHAR, NCHAR, NVARCHAR).</td>
</tr>
<tr>
<td>WITH MARK 'description'</td>
<td>WITH MARK is used to mark a specific point in the transaction log. When used, a transaction log restoration can be recovered to the point prior to the MARK.</td>
</tr>
</tbody>
</table>

This example begins a transaction called update_orders. A new value is inserted into the t_Orders and t_OrdersHistory tables. After this, another transaction, called update_lastorder is created. Within the update_lastorder transaction, the t_LastOrder table is updated, and the transaction is committed. The outer transaction, update_orders, is also committed:

```
BEGIN TRANSACTION
    INSERT t_Orders (OrderId) VALUES (32)
    INSERT t_OrdersHistory (OrderId, dtOrderDate) VALUES (32, GETDATE())
    SAVE TRANSACTION orders_updated
    UPDATE t_LastOrder
    SET dtLastOrderDate = GETDATE()
    ROLLBACK TRANSACTION orders_updated
COMMIT TRANSACTION
```

The above example shows how to use named and nested transactions; when a transaction is used within another named transaction, it is called a nested transaction.

It is important to note that SQL Server mostly ignores nested transactions. If the first or outer transaction is committed, then the inner nested transactions are committed too. If a COMMIT that is not at the outermost level is called, the COMMIT is ignored. However, if the outer transaction is rolled back, all inner transactions roll back too, even if they were committed.
Example: SAVEPOINT and ROLLBACK

This example does the following:

- Inserts a row into the t_Orders and t_OrdersHistory table.
- Issues a SAVE TRANSACTION
- Makes an UPDATE to the t_LastOrder
- We then decide against this update, and issue a ROLLBACK TRANSACTION for the named savepoint.

Both the t_Orders and t_OrdersHistory changes are committed, but the t_LastOrder change is not:

```sql
BEGIN TRANSACTION
INSERT t_Orders (OrderId) VALUES (12)
INSERT t_OrdersHistory (OrderId, dOrderDate) VALUES (12, GETDATE())
SAVE TRANSACTION orders_updated
UPDATE t_LastOrder
SET dLastOrderDate = GETDATE()
ROLLBACK TRANSACTION orders_updated
COMMIT TRANSACTION
```

Example: @@Errors

You can use @@Errors to roll back a transaction if errors occur within it. This prevents partial updates and having to SET XACT_ABORT ON.

In this example, if either INSERT statement produces an error (in this case a primary key constraint error), the transaction rolls back:

```sql
BEGIN TRANSACTION
INSERT t_Orders (OrderId)
VALUES (12)
IF (@ERROR <> 0) GOTO ErrorOccurred
INSERT t_OrdersHistory (OrderId, dOrderDate)
VALUES (12, GETDATE())
IF (@ERROR <> 0) GOTO ErrorOccurred
COMMIT TRANSACTION
ErrorOccurred:
ROLLBACK TRANSACTION
```

Server: MyServer, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK_orders'. Cannot insert duplicate key in object 't_Orders'.
The statement has been terminated.
GOTO is a Transact-SQL keyword that helps you control the flow of statement execution. In this example, ErrorOccured is a label. You define a label by using a text word followed by a colon. For example:

jumptohere:

You can then execute:

GOTO jumptohere

The code will then jump to that code section. GOTO is a throwback from the file-oriented programming languages, and is most often used for jumping to error handling placed towards the end of a batch.

5.1 How to… Use a Transaction Mark

Using the WITH MARK command with the BEGIN TRANSACTION statement places a named mark in the transaction log, allowing you to restore your log to this point. The transaction must contain at least one data modification for the mark to be placed in the log. You should use this functionality for transactions that are prone to causing problems, where you may need to recover or undo the transaction to a point before the modification. We will review backup and restore techniques in the next chapter; the following, however, is an example of setting a mark:

```
BEGIN TRAN bookorderupdate WITH MARK
UPDATE BookRepository.dbo.Books
SET dtReceived = GETDATE()
COMMIT TRAN bookorderupdate
```

The following query shows the results of the update:

```
SELECT iBookId, vchBookName, dtReceived
FROM Books
```

<table>
<thead>
<tr>
<th>iBookId</th>
<th>vchBookName</th>
<th>dtReceived</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQL Server Fast Answers for...</td>
<td>2002-12-12 15:49:07.167</td>
</tr>
<tr>
<td>3</td>
<td>SQL for Everyone</td>
<td>2002-12-12 15:49:07.167</td>
</tr>
<tr>
<td>4</td>
<td>Love in the Age of SQL</td>
<td>2002-12-12 15:49:07.167</td>
</tr>
</tbody>
</table>

If this update is a mistake, you can restore the data up to the mark point. This assumes that you have already completed a full backup and transaction log backup before the transaction committed.

To restore the full backup (more details in the next chapter):

```
RESTORE DATABASE BookRepository
FROM DISK = 'J:\MSSQL\Backup\bookrepo_jul_17.bak' WITH NORECOVERY

RESTORE LOG BookRepository
FROM DISK = 'J:\MSSQL\Backup\bookrepo_jul_17_3pm.trn' WITH RECOVERY,
STOPATMARK=' bookorderupdate'
```
After the restore, the original `dtReceived` for each book is restored to the value it had prior to the change:

<table>
<thead>
<tr>
<th>iBookId</th>
<th>vchBookName</th>
<th>dtReceived</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQL Server Fast Answers for...</td>
<td>2002-02-02 00:00:00.000</td>
</tr>
<tr>
<td>2</td>
<td>SQL Fun 2nd Edition</td>
<td>2001-03-14 00:00:00.000</td>
</tr>
<tr>
<td>3</td>
<td>SQL for Everyone</td>
<td>2000-06-06 00:00:00.000</td>
</tr>
<tr>
<td>4</td>
<td>Love in the Age of SQL</td>
<td>2002-02-01 00:00:00.000</td>
</tr>
</tbody>
</table>

### Best Practices for using Transactions

- Keep transaction time short.
- Minimize resources locked by the transaction. Narrow down rows impacted by `INSERT`, `UPDATE`, and `DELETE` statements.
- Add Transact-SQL statements to a transaction where they are relevant to that transaction only.
- Do not open new transactions that require user feedback within the transaction. Open transactions can hold locks on resources, and user feedback can take an indefinite length of time to receive. Gather user feedback before issuing an explicit transaction.
- Check `@@ERROR` after issuing a DML (Data Manipulation Language) statement. If there was an error, you can roll back the transaction.
- The database option `SET XACT_ABORT` affects how errors are handled within a transaction. When set `ON`, DML statements within a transaction that raise a run-time error cause the entire transaction to roll back and terminate. When `OFF`, only the DML statement that raised the error is rolled back, and the rest of the transaction continues. Keep this option `ON` to ensure data consistency.
- If possible, do not open a transaction when browsing data.
- Use and understand the correct isolation levels. We will review isolation levels further on in this chapter.

### 5.2 How to… Display the Oldest Active Transaction with DBCC OPENTRAN

`DBCC OPENTRAN` is a Transact-SQL command that is used to view the oldest running transaction for the selected database. The `DBCC` command is very useful for troubleshooting orphaned connections (connections still open on the database but disconnected from the application or client), and identification of transactions missing a `COMMIT` or `ROLLBACK`.

This command also returns the oldest distributed and undistributed replicated transactions, if any exist within the database. If there are no active transactions, no data will be returned.

If you are having problems with your transaction log not truncating inactive portions, `DBCC OPENTRAN` can show whether an open transaction is to blame.
The syntax for `DBCC OPENTRAN` from *SQL Server Books Online* is as follows:

```sql
DBCC OPENTRAN
    ( { 'database_name' | database_id } )
[ WITH TABLERESULTS
    [ , NO_INFOMSGS ]
]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>Use either the database name or the database ID. If left blank, the <code>DBCC</code> command will run for the current database context.</td>
</tr>
<tr>
<td>database_id</td>
<td>When specified, this returns data in tabular format, meaning it can be output to a table.</td>
</tr>
<tr>
<td>WITH TABLERESULTS</td>
<td>Disables the return of informational messages.</td>
</tr>
</tbody>
</table>

In this example, we run `DBCC OPENTRAN` for the BookRepository database using `WITH TABLERESULTS`:

```
DBCC OPENTRAN [bookrepository, WITH TABLERESULTS]
```

![Table with data]

Since you now have the SPID, you can see what command is currently in the input buffer using `DBCC INPUTBUFFER (SPID)`. The SPID is a process identifier that is assigned to each user connection when a connection to SQL Server is made. The `DBCC INPUTBUFFER` command is used to view the last statement sent by the client connection to SQL Server. `DBCC INPUTBUFFER` takes one parameter, the SPID:

```
DBCC INPUTBUFFER (52)
```

```
<table>
<thead>
<tr>
<th>EventType</th>
<th>Parameters</th>
<th>EventInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANGUAGE</td>
<td>Event 0</td>
<td>BEGIN TRANSACTION</td>
</tr>
<tr>
<td>INSERT</td>
<td>t_orders</td>
<td>(OrderId)</td>
</tr>
<tr>
<td></td>
<td>VALUES (14)</td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>t_ordersHistory</td>
<td>(OrderId, dtOrderDate)</td>
</tr>
<tr>
<td></td>
<td>VALUES (14, CAST(dtOrderDate AS datetime))</td>
<td></td>
</tr>
</tbody>
</table>
```
5.3 How to... Use SET TRANSACTION ISOLATION LEVEL

This section touches on the topic of locking, which will be discussed in more detail later in this chapter.

Transactions and locking go hand in hand. Depending on your application design, your transactions can significantly impact database concurrency (the number of people that can access and modify the database and database objects at the same time); this can have a significant effect on performance and scalability.

The following statement sets the default transaction locking behavior for SELECT statements used in a connection; you may only have one level set at a time. Isolation level does not change unless explicitly SET.

The syntax for SET TRANSACTION ISOLATION LEVEL from SQL Server Books Online is as follows:

```
SET TRANSACTION ISOLATION LEVEL
     {READ COMMITTED |
      READ UNCOMMITTED |
      REPEATABLE READ |
      SERIALIZABLE }
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ COMMITTED</td>
<td>Shared locks are held for the duration of the transaction; the data cannot be changed. Data inserts and modifications to the same table by other transactions are allowed, so long as the rows involved are not locked by the first transaction.</td>
</tr>
<tr>
<td>(this is the default value)</td>
<td></td>
</tr>
<tr>
<td>READ UNCOMMITTED</td>
<td>This is the least restrictive isolation level, issuing no locks on the data selected by the transaction. This provides the highest concurrency but the lowest level of data integrity, as the data that you read can be changed while you read it (these reads are known as dirty reads), or new data can be added or removed that would change your original query results.</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Dirty and nonrepeatable reads are not allowed. New rows can still be inserted by other transactions.</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>This is the most restrictive setting. Ranges of shared locks are held on the data, disallowing insertion of new rows, deletion of existing rows, and modification of rows within the selected range.</td>
</tr>
</tbody>
</table>

Later on in the chapter we will discuss the pros and cons of the different locking isolation levels.
In this example, we will set the isolation level to SERIALIZE:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
GO
BEGIN TRANSACTION
SELECT COUNT(*) FROM ImportedBooks
```

This example begins a transaction that selects the row count of the ImportedBooks table. Since we did not commit the transactions, range locks will be held on the table’s rows. If you attempt to insert rows within the range of the ImportedBooks table, you will wait indefinitely; the little Query Analyzer globe will spin until the original transaction commits or rolls back:

```
INSERT ImportedBooks
VALUES (1, 1, 1)
```

5.4 How to... Use SET CURSOR_CLOSE_ON_COMMIT

This option specifies whether a cursor is closed or open when a transaction is committed. We will cover Transact-SQL cursors in more detail in Chapter 16.

When this option is set to ON, cursors created within a transaction are no longer available once the transaction is committed or rolled back. This is an ANSI SQL-92 standard. When OFF (which is the default for SQL Server), the cursor is not closed when a transaction commits.
Below is an example that creates a cursor on the ImportedBooks table within a transaction. Because we have set CURSOR_CLOSE_ON_COMMIT ON, the cursor FETCH statement after the transaction commit will fail:

```
SET CURSOR_CLOSE_ON_COMMIT ON
GO
BEGIN tran
DECLARE cuImportedBooks CURSOR FOR
SELECT iBookId
FROM ImportedBooks
    OPEN cuImportedBooks
    COMMIT tran
    FETCH NEXT FROM cuImportedBooks
    GO
```

5.5 How to… Use SET REMOTE_PROC_TRANSACTIONS

If SET REMOTE_PROC_TRANSACTIONS is ON, a distributed transaction is started when a remote stored procedure is called from a local server transaction. This remote procedure call (RPC) uses Microsoft Distributed Transaction Coordinator (MS DTC) to manage the transaction. This is a Windows service that can be found in Service Manager, or under the Support Services folder in Enterprise Manager. MS DTC is used to handle distributed transactions, acting as the manager of the transaction. This means that MS DTC is responsible for the COMMIT or ROLLBACK phase of a transaction that traverses two SQL Server instances. MS DTC ensures all or nothing transaction behavior, even though the transaction traverses two servers.

Both the local and distributed servers must have MS DTC running to control the procedure call. This SET option overrides the sp_configure REMOTE_PROC_TRANS option. When the option is OFF, a distributed transaction is not used.

You can also start distributed transactions by using BEGIN DISTRIBUTED TRANSACTION. Note that REMOTE_PROC_TRANSACTIONS only covers remote stored procedure calls.

The following is an example of enabling the REMOTE_PROC_TRANSACTIONS SET option:
Lock Types, Compatibility, Isolation Levels, and Hints

Locking ensures that multiple users do not modify data at the same time. Depending on your lock type and isolation level, locking can also prevent users from reading data while it is being updated.

Application design and use of isolation levels can have a significant impact on database concurrency (the ability of users to access the data) and data consistency (the ability of users to read the correct data). Isolation levels can also have an impact on scalability and performance. The more users you have, the more resource locking can occur, potentially impacting query performance.

The 'Famous Four' Concurrency Problems

The 'famous four' database concurrency problems occur when more than one user attempts to:

- Read data that another is modifying.
- Modify data that another is reading.
- Modify data that another transaction is trying to modify.

The 'famous four' are as follows:

- **Dirty reads**
  Dirty reads occur while a transaction is updating a row, and a second transaction reads the row before the first transaction is committed. If the original update rolls back, the data read by the second transaction is not the same, hence a dirty read has occurred.

- **Nonrepeatable reads**
  These occur when a transaction is updating data while a second transaction is reading the same data, both before and after a change. Data retrieved from the first query does not match the second query (this presumes that the second transaction reads the data twice; once before and once after).

- **Phantom reads**
  These occur when a transaction retrieves a set of rows once, another transaction inserts or deletes a row from that same table, and the first transaction looks at the query again. The phantom is the missing or new row.

- **Lost updates**
  Lost updates occur when two transactions update a row's value, and the transaction that last updates the row 'wins'; thus the first update is lost.

Locks help alleviate these problems when applied appropriately. The following table displays the types of lock that SQL Server 2000 can issue:

<table>
<thead>
<tr>
<th>Name</th>
<th>Initial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent Lock</td>
<td>I</td>
<td>Intent locks effectively create a lock queue, designating the order of connections and their associated right to update or read resources. SQL Server uses intent locks to show future intention of acquiring locks on a specific resource.</td>
</tr>
<tr>
<td>Intent Shared</td>
<td>IS</td>
<td>Intent to read some (but not all) of the resources in the table or page using shared locks. Used for read-only operations.</td>
</tr>
</tbody>
</table>
## Transactions, Locking, Blocking, and Deadlocking

<table>
<thead>
<tr>
<th>Name</th>
<th>Initial</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent Exclusive</td>
<td>IX</td>
<td>Intent to modify some (but not all) resources in the table or page using an exclusive lock. Used for data-modification purposes.</td>
</tr>
<tr>
<td>Shared with Intent</td>
<td>SIX</td>
<td>Intent to read all resources and modify some. A combination of shared locks and exclusive locks will be placed. One lock is held for the resource, and multiple IX and IS locks can be placed.</td>
</tr>
<tr>
<td>Update Lock</td>
<td>U</td>
<td>Update locks are acquired prior to modifying the data. When the row is modified, this lock is escalated to an exclusive lock. If not modified, it is downgraded to a shared lock. This lock type prevents deadlocks if two connections hold a shared (S) lock on a resource, and attempt to convert to an exclusive (X) lock, but cannot because they are each waiting for the other transaction to release the shared (S) lock.</td>
</tr>
<tr>
<td>Exclusive Lock</td>
<td>X</td>
<td>Lock on the resource that bars any kind of access (reads or writes). Issued during INSERT, UPDATE, or DELETE statements.</td>
</tr>
<tr>
<td>Schema Modification</td>
<td>Sch-M</td>
<td>Issued when a DDL statement is executed.</td>
</tr>
<tr>
<td>Schema Stability</td>
<td>Sch-S</td>
<td>Issued when a query is being compiled. Keeps DDL operations from being performed on the table.</td>
</tr>
<tr>
<td>Bulk Update</td>
<td>BU</td>
<td>This type of lock is issued during a bulk-copy operation. Performance is increased for the bulk-copy operation, but table concurrency is reduced. Used when either TABLOCK hint or sp_tableoption 'table lock on bulk load' is enabled.</td>
</tr>
</tbody>
</table>

The following are resources to which SQL Server can apply locks:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID</td>
<td>Row identifier, designating a single table row.</td>
</tr>
<tr>
<td>Key</td>
<td>Index row lock, helping prevent phantom reads. Also called key-range lock, this lock type uses both a range and a row component. The range represents the range of index keys between two consecutive index keys. The row component represents the lock type on the index entry. Range types are as follows: RangeS_S mode – Shared range, shared resource lock. RangeS_U mode – Shared range, update resource lock. RangeI_N mode – Insert range, null resource lock (row doesn’t exist yet). RangeX_X mode – Exclusive range, exclusive resource lock.</td>
</tr>
<tr>
<td>Page</td>
<td>Referring to a 8KB data or index page.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
Not all lock types are compatible with each other; for lock types to be compatible, SQL Server must be able to place a lock on a table that has a lock already in place. Lock types must be compatible with any existing lock currently placed on a resource.

The following table lists the lock types, and their associated compatibility types:

<table>
<thead>
<tr>
<th>Lock Type</th>
<th>Compatible With...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared (S) locks</td>
<td>Compatible with intent shared (IS), other shared (S), and update (U) locks.</td>
</tr>
<tr>
<td>Update (U) locks</td>
<td>Compatible with intent shared (IS) and shared (S) locks.</td>
</tr>
<tr>
<td>Exclusive (X) locks</td>
<td>Compatible with any other lock types.</td>
</tr>
<tr>
<td>Intent Shared (IS) locks</td>
<td>Compatible with all locks except exclusive (X) locks.</td>
</tr>
<tr>
<td>Shared with Intent Exclusive (SIX) locks</td>
<td>Compatible with intent shared (IS) locks.</td>
</tr>
<tr>
<td>Intent Exclusive (IX) locks</td>
<td>Compatible with intent shared (IS) and intent exclusive (IX) locks. This is because IX intends to update some rows, but not all.</td>
</tr>
<tr>
<td>Schema Stability (Sch-S) locks</td>
<td>Compatible with all lock modes except schema modification (Sch-M).</td>
</tr>
<tr>
<td>Schema Modification (Sch-M) locks</td>
<td>Incompatible with all locks.</td>
</tr>
<tr>
<td>Bulk Update (BU) locks</td>
<td>Compatible only with schema stability (Sch-S) and other bulk update (BU) locks.</td>
</tr>
<tr>
<td>RangeS_S mode</td>
<td>Compatible with shared (S), rangeS_S, and rangeS_U locks.</td>
</tr>
<tr>
<td>RangeS_U mode</td>
<td>Compatible with shared (S) and RangeS_S locks.</td>
</tr>
<tr>
<td>RangeI_N mode</td>
<td>Compatible with shared, update, exclusive, and RangeI_N locks; this is because the inserted rows do not yet exist.</td>
</tr>
<tr>
<td>RangeX_X mode</td>
<td>Compatible with any other lock type.</td>
</tr>
</tbody>
</table>

Locks are allocated and escalated automatically by SQL Server; escalation means that finer grain locks (row and page locks) are converted into coarse-grain table locks. Locks take up system memory, so converting many locks into one larger lock can free up memory resources. Row level locking (versus page or table locking) does increase database concurrency, but at a cost of memory overhead.

Although you cannot control lock escalation, you do have control over locking isolation levels and locking hints in your queries. Below is a review of each isolation level, this time including what type of issue the isolation level resolves:
Transactions, Locking, Blocking, and Deadlocking

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>What Does It Solve?</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ COMMITTED</td>
<td>While READ COMMITTED is set, shared locks are held for the duration of the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transaction. The data cannot be changed. Data inserts and modifications to</td>
<td>This is the default isolation level for SQL Server.</td>
</tr>
<tr>
<td></td>
<td>the same table are allowed by other transactions, so long as the rows</td>
<td>Remember this always; it means that your SELECT statements, unless otherwise</td>
</tr>
<tr>
<td></td>
<td>involved are not locked by the first transaction.</td>
<td>specified, cannot read uncommitted data. Dirty reads are prevented.</td>
</tr>
<tr>
<td>READ UNCOMMITTED</td>
<td>This is the least restrictive isolation level, issuing no locks on the data</td>
<td>It does not solve concurrency problems, but it does help with locking contention</td>
</tr>
<tr>
<td></td>
<td>selected by the transaction. This provides the highest concurrency but the</td>
<td>difficulties. You can read data while it is being updated or selected, so you do</td>
</tr>
<tr>
<td></td>
<td>lowest amount of data integrity. Data that you read can be changed while you</td>
<td>not have to wait to retrieve the data. If this is used with static data, or data</td>
</tr>
<tr>
<td></td>
<td>read it (dirty reads), or new data can be added or removed that would change</td>
<td>that doesn't require exact approximations, READ UNCOMMITTED may make sense for</td>
</tr>
<tr>
<td></td>
<td>your original query results.</td>
<td>your query.</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Dirty and nonrepeatable reads are not allowed. New rows can still be inserted</td>
<td>Prevents nonrepeatable and dirty reads.</td>
</tr>
<tr>
<td></td>
<td>by other transactions.</td>
<td></td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>This is the most restrictive setting. Ranges of shared locks are held on the</td>
<td>Prevents phantom, nonrepeatable, and dirty reads.</td>
</tr>
<tr>
<td></td>
<td>data, preventing insertion of new rows, deletion of existing rows, and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>modification of rows within the selected range.</td>
<td></td>
</tr>
</tbody>
</table>

Locking hints are another weapon in your locking strategy arsenal, and bypass SQL Server's dynamic locking behavior. SQL Server cannot always guess your intentions, so adding locking hints can help SQL Server make the best locking decision. For example, if you have a table with static data that can use the NOLOCK hint without risk of dirty data, doing so will lower the overhead of the locking that SQL Server would use by default.

On the other hand, hints can also do more harm than good. Using ROWLOCK, for example, on a million-row table could cause outrageous memory overhead, as opposed to SQL Server dynamically escalating a rowlock to a page or table lock. It is best to use hints carefully, and not forget where you placed them in your application. Schemas and data distribution can change over time, and your previously well-performing query could suffer from the use of a hint.

Locking hints are used with the SELECT, INSERT, UPDATE, and DELETE statements; not all hints can be used with each DML type (SELECT, INSERT, UPDATE, and DELETE). Keep in mind that locking hints override the default or specified isolation level for the user session.
The syntax for locking hint usage from *SQL Server Books Online* is as follows:

```
FROM
< table_source > ::= table_name [ [ AS ] table_alias ] [ WITH ( < table_hint > [ ,...,n ] ) ]
< table_hint > ::= ( INDEX { index_val [ ,...,n ] }
  | FASTFIRSTROW
  | HOLDLOCK
  | NOLOCK
  | PAGLOCK
  | READCOMMITTED
  | READPAST
  | READUNCOMMITTED
  | REPEATABLEREAD
  | ROWLOCK
  | SERIALIZABLE
  | TABLOCK
  | TABLOCKX
  | UPDLOCK
  | XLOCK )
```

There are two categories of lock hints: **granularity hints** and **isolation-level hints**. Granularity hints impact the resource locking used by a query; isolation-level hints impact the lock types used on the locked resource.

The granularity hints are:

<table>
<thead>
<tr>
<th>Granularity Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGLOCK</td>
<td>Tells SQL Server to use page locks instead of a single table lock. Use this hint if you are seeing a table lock being used unnecessarily by SQL Server (where page locks would be more efficient).</td>
</tr>
<tr>
<td>NOLOCK</td>
<td>Same behavior as UNCOMMITTED READ isolation level. No locks of any kind are issued, but dirty reads are possible. Only usable with <code>SELECT</code> statement. Use this hint when you feel confident that locks will be unnecessary (read-only databases, or updates to data that are on off-periods).</td>
</tr>
<tr>
<td>ROWLOCK</td>
<td>Issues row-level locks instead of page and table-level locks. Be careful about applying this to too many rows (to avoid memory problems).</td>
</tr>
<tr>
<td>TABLOCK</td>
<td>Issues a table lock instead of a row or page lock. Lock is held until the end of the statement, not the transaction. Use this if page locks are granted, but you intend to lock the entire table.</td>
</tr>
<tr>
<td>TABLOCKX</td>
<td>Issues an exclusive lock on a table. No reads or updates are allowed by other connections until the end of the statement or transaction.</td>
</tr>
</tbody>
</table>

The isolation-level hints are as follows (notice some are considered both granularity and isolation hints):
<table>
<thead>
<tr>
<th>Isolation Level Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOLDLOCK</td>
<td>Issues a shared lock until completion of the transaction. Does not release lock after table, row, or data page has been modified. Same behavior as SERIALIZABLE isolation level.</td>
</tr>
<tr>
<td>NOLOCK</td>
<td>Same behavior as uncommitted read isolation level. No locks of any kind are issued, but dirty reads are possible. Only useable with SELECT statements.</td>
</tr>
<tr>
<td>READCOMMITTED</td>
<td>SQL Server 2000 default isolation level of READ COMMITTED.</td>
</tr>
<tr>
<td>READPAST</td>
<td>Within a SELECT statement, designating READPAST will cause the query to skip any locked rows. Applies to transactions using READ COMMITTED isolation level. Only reads past row level locks.</td>
</tr>
<tr>
<td>READUNCOMMITTED</td>
<td>Same as NOLOCK hint.</td>
</tr>
<tr>
<td>REPEATABLEREAD</td>
<td>Same behavior as REPEATABLE READ isolation level.</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>Same as HOLDLOCK. Uses SERIALIZABLE isolation level.</td>
</tr>
<tr>
<td>UPDLOCK</td>
<td>With UPDLOCK, you can read data without blocking other readers, and update the data later knowing that data has not been changed since the last read.</td>
</tr>
<tr>
<td>XLOCK</td>
<td>Issues an exclusive lock that will be held until the end of the transaction for all data processed. Can be included with the PAGLOCK or TABLOCK hints.</td>
</tr>
</tbody>
</table>

The following is an example of using the SERIALIZABLE locking hint:

```
BEGIN TRANSACTION
SELECT * FROM Customers
WITH (SERIALIZABLE)
```

Using `sp_lock`, you can see that the query (since it is not committed yet) holds key locks on all the rows in the range. `sp_lock` is a system stored procedure that reports information on locks held within the SQL Server instance:
The `ObjId` in the results output is the object ID of the resource being locked. To view the object name, run the following command with the integer value from `sp_lock`:

```
SELECT OBJECT_NAME (2073056421)
```

We will go into more detail on `sp_lock` later in the chapter.

In the next example, we run the same query, but this time with the `NOLOCK` hint:

```
BEGIN TRANSACTION
SELECT * FROM Customers
WITH (NOLOCK)
```

```
Looking at\textit{sp\_lock}, you will not see any locks held on the \textit{Customers} table:

![Image of \textit{sp\_lock} table]

In the next example, we want to update the \textit{Customers} table's update date for all rows. We run the following statement:

![Image of BEGIN TRANSACTION]

The statement has not committed, so by running \textit{sp\_lock}, we see that exclusive locks have been placed on the key resources:

![Image of \textit{sp\_lock} table]
This is a case where SQL Server decides to place key locks when a table lock would have worked just as well. If you know that all rows must be updated, you can use a coarser-grained resource lock, like the following:

```
BEGIN TRANSACTION
UPDATE CUSTOMERS
WITH (TABLOCK)
SET dUpdateDate = GETDATE()

[1 row(s) affected]
```

The result of `sp_lock` after using the WITH (TABLOCK) statement is as follows:

```
<table>
<thead>
<tr>
<th>spid</th>
<th>dbid</th>
<th>Objid</th>
<th>Indid</th>
<th>Type</th>
<th>Resource</th>
<th>Mode</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51</td>
<td>4</td>
<td>0</td>
<td>DB</td>
<td>S</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>4</td>
<td>0</td>
<td>DB</td>
<td>S</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>53</td>
<td>5</td>
<td>0</td>
<td>DB</td>
<td>S</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>53</td>
<td>203658421</td>
<td>0</td>
<td>TAB</td>
<td>X</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>56</td>
<td>5</td>
<td>0</td>
<td>DB</td>
<td>S</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>57</td>
<td>5</td>
<td>0</td>
<td>DB</td>
<td>S</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>57</td>
<td>8557543</td>
<td>0</td>
<td>TAB</td>
<td>IS</td>
<td>GRANT</td>
<td></td>
</tr>
</tbody>
</table>
```

Instead of multiple key locks being issued, one exclusive table lock is created. For multiple rows in a large table (if you know that you will be updating all rows), it may make more sense to issue a table lock upfront, rather than issue hundreds of key locks (which consume memory).

### 5.6 How to... View Locking Activity in Enterprise Manager

Enterprise Manager allows you to view locking activity by SPID (process ID) and by database object. You can also view the process info (host, last batch time, login, database) and locking activity by object. To view locking activity in Enterprise Manager:

- Expand Server Group and registration.
- Expand the Management folder.
- Expand the Current Activity node.
- Click Process Info to see SPIDs (process IDs) in the database.
In the right pane, double-click a process ID to view the last command that it ran. From this window, you can also send the connection a message, or KILL the process. Select Close when finished with the Process Details dialog box.

To see locked resources by process ID, expand the Locks / Process ID node. Click a specific SPID to see what objects are being locked, the lock type, mode, status, owner, Index, and resource.

Expand the Locks / Object node to view locks by object resource. Click a specific object to view associated process IDs issuing locks.

When a blocking situation is in progress, the locks by process id view will display the process IDs being blocked, and the process ID doing the blocking. For example:

To refresh the activity to reflect the current activity of the server, right-click the Current Activity node and select Refresh.

Remember that you are looking at past history, and should refresh this view often, particularly if you plan on killing any process IDs.

5.7 How to… Use sp_who

Stored procedure sp_who displays information on current users and processes. sp_who returns:

- System process ID
- Execution context ID
- Status
- Login name
- Host or computer name
- Whether or not the process is currently blocked by another process
To display all processes on the server in Query Analyzer, run:

```
EXEC sp_who
```

For example:

```
sp_who
```

To specify activity for a specific process ID, execute:

```
EXEC sp_who SPID
```

For example:

```
sp_who 12
```

To specify only active processes, execute:

```
EXEC sp_who 'active'
```

For example:
5.8 How to… Use DBCC INPUTBUFFER and DBCC OUTPUTBUFFER

The DBCC INPUTBUFFER command is used to view the last statement sent by the client connection to SQL Server. The syntax for DBCC INPUTBUFFER is:

```
DBCC INPUTBUFFER (SPID)
```

SPID is the process ID, which you can get from viewing current activity in Enterprise Manager or executing `sp_who`. The output from DBCC INPUTBUFFER is good information to note when you encounter a blocking situation. We will review blocking and deadlocking in the next section, but it is important to be aware that identification of the blocking statement will assist you when troubleshooting the issue.

The following example of running DBCC INPUTBUFFER shows the Event Type, Parameters, and Event Info columns. For Event Type, the values RPC, Language, or No Event are returned. Parameters show either 0 for text, or an integer for the number of parameters. Event Info for an RPC (remote procedure call) will contain the procedure name. Event Type for a language or No Event will contain the first 255 characters of the text sent by the client:

```
DBCC OUTPUTBUFFER shows the information sent to the client in hexadecimal and ASCII format for the process ID specified. For example, for a SPID requesting all rows from the Customers table:
```

```
DBCC OUTPUTBUFFER (54)
```

```
01000000 00 01 0a 0f 00 36 0e 00 32 00 0c 00 06 00 04 00 08 00 04 00 05 00 06 00 07 00 08 00 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 80 00 81 00 82 00 83 00 84 00 85 00 86 00 87 00 88 00 89 00 8a 00 8b 00 8c 00 8d 00 8e 00 8f 00 90 00 91 00 92 00 93 00 94 00 95 00 96 00 97 00 98 00 99 00 9a 00 9b 00 9c 00 9d 00 9e 00 9f 00 a0 00 a1 00 a2 00 a3 00 a4 00 a5 00 a6 00 a7 00 a8 00 a9 00 aa 00 ab 00 ac 00 ad 00 ae 00 af 00 b0 00 b1 00 b2 00 b3 00 b4 00 b5 00 b6 00 b7 00 b8 00 b9 00 ba 00 bb 00 bc 00 bd 00 be 00 bf 00 c0 00 c1 00 c2 00 c3 00 c4 00 c5 00 c6 00 c7 00 c8 00 c9 00 ca 00 cb 00 cc 00 cd 00 ce 00 cf 00 d0 00 d1 00 d2 00 d3 00 d4 00 d5 00 d6 00 d7 00 d8 00 d9 00 da 00 db 00 dc 00 dd 00 de 00 df 00 e0 00 e1 00 e2 00 e3 00 e4 00 e5 00 e6 00 e7 00 e8 00 e9 00 ea 00 eb 00 ec 00 ed 00 ee 00 ef 00 f0 00 f1 00 f2 00 f3 00 f4 00 f5 00 f6 00 f7 00 f8 00 f9 00 fa 00 fb 00 fc 00 fd 00 fe 00 ff

```
```
Chapter 5

If you look carefully, you will see words in the third column. For greater legibility, there are stored procedures on the Internet and referenced in SQL Server message boards that allow you to decipher the ASCII format and view the actual query results returned to the client. This is useful if you are trying to see what sort or amount of data is being returned to the client. This command can only be run by members of the sysadmin group.

5.9 How to... Use fn_get_sql

SQL Server 2000 Service Pack 3 has introduced a new function called fn_get_sql. Like DBCC INPUTBUFFER, this function is used to return the SQL string for an active connection. Unlike DBCC INPUTBUFFER, fn_get_sql is able to return SQL strings greater than 255 characters.

One drawback of this function is that it won't usually return SQL strings for zero cost query plans. These are basic ad-hoc queries or calls that SQL Server usually does not cache because they have low overhead. The function also cannot return bulk copy statements or those statements with string literals exceeding 8KB, because these are not cached either.

The syntax for fn_get_sql is:

::fn_get_sql ( @SqlHandle )

The @SqlHandle parameter refers to the new column added in SP3 to sysprocesses, the sql_handle column. This new column represents the current executing batch, and is a binary(20) data type.

The following example query populates the @Handle parameter via a query to sysprocesses. (You must be sysadmin to execute this function):

In this example, a variable was used to hold the sql_handle for the spid 51. This variable was then fed to the fn_get_sql function, returning the text of the query in the text column.

The encrypted field returns 0 if not encrypted, and 1 if encrypted. The dbid and objectid fields will return NULL if the SQL Statement is an ad hoc SQL statement. If the procedure captured is grouped, the number column is the number of the stored procedure group. Otherwise the number column is NULL for ad hoc SQL statements or 0 for statements that are not procedures.

5.10 How to... Use SET LOCK_TIMEOUT

When a transaction or statement is blocked, it is waiting for a lock on a resource to be released. SQL Server offers the SET LOCK_TIMEOUT option to specify how long a statement should wait for a lock to be released. The syntax from SQL Server Books Online is as follows:

SET LOCK_TIMEOUT timeout_period
timeout_period is the number of milliseconds the statement should wait. For example, setting the timeout to five seconds (5000 milliseconds):

```
SET LOCK_TIMEOUT 5000
```

This setting doesn’t impact how long a resource can be held by a process, only how long it has to wait for another process to give up the goods. Rather than setting this lock timeout, you may be better off examining why inappropriate locking is occurring.

### Blocking and Deadlocking Defined

**Blocking** occurs when a transaction is locking resources that one or more other transactions want to read or modify. Short-term blocking is usually okay, depending on your application requirements. Poorly-designed applications can cause long-term blocking, unnecessarily holding locks on resources and keeping other sessions from reading or updating those resources. A blocked process can wait indefinitely, until:

- It times out (based on `SET LOCK_TIMEOUT`)
- The server goes down
- The connection finishes its updates
- Something happens to the original transaction to cause it to release its locks on the resource

**Deadlocking** occurs when one user session (let’s call it Session 1) has locks on a resource that another user session (let’s call it Session 2) wants to modify, and Session 2 has locks on resources that Session 1 needs to modify. Neither Session 1 nor Session 2 can continue until the other releases the locks, so SQL Server chooses one of the sessions in the deadlock as the **deadlock victim**. This deadlock victim has its session killed and transactions rolled back:
Why Blocking and Deadlocking Happen

Before trying to troubleshoot blocking and deadlocking, we should understand why they happen in the first place. There are various reasons for why long-term blocking occurs:

- The issue may be normal database blocking or excessive long-term blocking. If users are waiting long periods of time, system processes are standing still, and you see long-standing blocks, or numerous blocks in a short period of time, investigate further.
- Without proper indexing, blocking issues can grow. Row locks on a table without an index can cause SQL Server to acquire a table lock, blocking out other transactions.
- Badly written applications can cause blocking. This is a broad category, so let's deal with it in more detail:
  - Transactions that BEGIN and then request user feedback or interaction. This is usually when an end user is allowed to enter data in a GUI while a transaction remains open. While open, any resources referenced by the transaction may be held with locks.
  - Transactions that BEGIN and then look up data that could have been referenced prior to the transaction starting.
  - Using locking hints inappropriately; for example, if the application only needs one row, but uses a table lock instead.
  - The application uses long-running transactions, updating many rows or many tables within one transaction.

Reasons why deadlocks occur:

- The application accesses tables in different orders. For example, Session A updates Customers and then Orders, whereas Session B updates Orders and then Customers. This increases the chance of two processes deadlocking, rather than accessing and updating a table in a serialized fashion.
- The application uses long-running transactions, updating many rows or many tables within one transaction.
- In some situations, SQL Server issues several row locks, which it later decides must be escalated to page or table locks. If these rows exist on the same data pages, and two sessions are both trying to escalate the lock granularity on the same page, a deadlock can occur.

5.11 How to... Identify and Resolve Blocking

First we will review resolution of blocking issues on your server, keeping in mind that we are talking about short-term resolution. This means identification of the offending session and, if possible, stopping the session or transaction. If this is not possible, we could issue a **KILL** statement to get rid of the blocking process. This often does not solve the original problem of why the blocking occurred, so check *Troubleshooting Blocking and Deadlocking Problems* later in the chapter.

- In Query Analyzer, run **sp_who**.
- Look at the **BlkBy** column. When a number exists in this column, this identifies the SPID that is blocking that current row's process. You could have a chain of blocking (numerous blocked or blocking processes), so you may need to run through this process many times. In order to solve the long-term issue that may be causing the blocking, track the SPID doing the blocking, and also the login, host, database name, command, program name, and last batch columns.
Run `sp_lock`, including the SPID that is blocking other processes. For example:

```
EXEC sp_lock 53
```

This returns:

From the results, you should track the type of locks, lock mode, and objects impacted. The options for status are `GRANT` (lock was obtained), `WAIT` (blocking is occurring), or `CNVT` (lock is converting to another lock). To see the object name from the `ObjId` integer, execute:

```
SELECT OBJECT_NAME (integer)
```

This returns:

Now that you know which statement is blocking the resources, and which resources are being blocked, it is time to take a look at the statement issuing the block, by running `DBCC INPUTBUFFER (SPID)`.
If possible, ask your client or application contact to stop the transaction blocking the resources. Make sure that this is not a long-running transaction that is critical to the business (you do not want to stop a payroll process midstream). In that case, you may want to let it run and attempt to fix the problem later. In either case, if you cannot stop the process in an elegant way, you can run the KILL process to stop the blocking:

```sql
KILL SPID
```

For example:

```sql
KILL 53
```

The command completed successfully.

While we are on the topic of the KILL keyword, it is important to understand that some KILL statements take longer to roll back a transaction than others. SQL Server 2000 added the ability to check the status of the KILL rollback by running KILL SPID WITH STATUSONLY:

```sql
KILL 54 WITH STATUSONLY
```

For example:
Regarding block troubleshooting in Enterprise Manager, we reviewed functionality of the Current Activity node in a previous section. To troubleshoot blocking, follow these steps:

- Expand Server Group and registration.
- Expand the Management folder.
- Expand Current Activity.
- Expand Locks / Process ID.
- Click the blocking SPID to see which locks are being held on which resources in the right-hand pane.
- Click the Process Info node and check the right-hand pane for details on the blocking SPID.
- Inform the user, application, or process that is causing the blocking – giving them an opportunity to either finish the process, or shut the process down. If this is a process that has no owner and can be removed, right-click the SPID and select Kill Process.
- Select YES in the dialog box to kill the process.
- Right-click and select Refresh to see a new view of the processes.

### 5.12 How to… Identify Deadlocks with Trace Flag 1204

When a deadlock occurs, the deadlock victim will receive an error like this:

Server: Msg 1205, Level 13, State 50, Line 6
Transaction (Process ID 60) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

You will also see a message in the SQL error log something like this:
Unfortunately, the default message does not include detailed information on the resources involved and the commands being run.

In Query Analyzer, you can activate trace flag number 1204 to show more information on deadlocking errors. Trace flags are used within SQL Server to enable or disable specific SQL Server instance characteristics temporarily. Traces are enabled using the DBCC TRACEON command, and disabled using DBCC TRACEOFF.

When trace flag 1204 is enabled, it returns the types of lock participating in the deadlock and the statements involved. Trace flag 1204 is often coupled with trace flag 3605, which writes the results of flag 1204 to the SQL error log. Including trace flag -1 means that the traces will apply to all sessions, and not just the current user session where the trace is activated.

Beginning in SQL Server 2000 Service Pack 3, the trace flag 1204 will automatically write the results of the flag to the SQL error log. This means that with SP3, you need not enable the trace flag 3605.

Trace flags -1 and 3605 are undocumented, so there are no guarantees that these will be available in future service packs or versions of SQL Server. Below is an example of enabling all three trace flags with DBCC TRACEON:

```
DBCC TRACEON (3605, 1204, -1)
```

To check currently enabled flags for all sessions, run:

```
DBCC TRACESTATUS (-1)
```

You will not see -1 as one of the trace flags, but you can test to see if it worked by opening a new window and running DBCC TRACESTATUS (-1). If you see the other two trace flags, you know that it worked, as trace flag -1 applies TRACEON settings to all sessions.
If a deadlock occurs after enabling these flags, you will see output like the following in the SQL error log:

```
[2002-07-10 10:32:19.51] spid65 Transaction (Process ID 65) was de-allocated on lock resources with another pr...
[2002-07-10 10:32:19.51] spid65 Resource Owner: ...
[2002-07-10 10:32:19.51] spid65 Requested By: ...
[2002-07-10 10:32:19.51] spid64 Owner: 0x45c000 Mode: S Flag: Ref: Life: 02000000 SPID:59 ESSID:0
[2002-07-10 10:32:19.51] spid64 Grant List::...
[2002-07-10 10:32:19.51] spid64 KEV: 519779576790 (0024466bc4e0) CleanCrit: Mode: S Flags: 0x0
[2002-07-10 10:32:19.51] spid64 Requested By: ...
[2002-07-10 10:32:19.51] spid64 Grant List::...
[2002-07-10 10:32:19.51] spid64 KEV: 519779576790 (0024466bc4e0) CleanCrit: Mode: S Flags: 0x0
[2002-07-10 10:32:19.51] spid64 Node:1
[2002-07-10 10:32:19.51] spid64 Requested By: ...
```

This SQL error log output shows the DBCC INPUTBUFFER results for the deadlock victim, DBCC INPUTBUFFER for the surviving process, and lock types held on the participating resources. Deadlock error log output depends on the type of deadlock that occurred. For an in-depth review of interpreting the SQL error log trace 1204 output, see the SQL Server Books Online topics, Deadlocks Involving Locks, Deadlocks Involving Parallelism, and Deadlocks Involving Threads.

When you have finished troubleshooting a deadlocking situation, make sure to disable the trace flags as follows. If you keep flags running, you risk increasing the overhead of SQL Server, and potentially reducing the SQL Server instance performance:

```
DBCC TRACEOFF (3605, 1204)
```

### 5.13 How to… Enable Trace Flags at SQL Server Startup

You can also set trace flags to start when SQL Server is restarted, using startup parameters. For example:

- Expand Server Group, right-click the registration and select Properties.
At the SQL Server Properties window, select the Startup Parameters button.

For each startup parameter you wish to have, type -Tnumber and press the Add button, where Tnumber is the letter T and the trace flag number:

If you wanted both trace flags 1204 and 3605, your startup parameters should look something like this:

Select OK when finished. Select OK at the SQL Server properties window. The service must restart for these trace flags to take effect. To remove them, you must go back to Startup Parameters, select Remove for each trace flag, and restart the SQL Server service.

5.14 How to... SET DEADLOCK_PRIORITY

You can increase the likelihood of a session being chosen as a deadlock victim by using the SET DEADLOCK_PRIORITY statement.

The syntax from SQL Server Books Online is:

```
SET DEADLOCK_PRIORITY { LOW | NORMAL | @deadlock_var }
```
LOW identifies the session to be the preferred deadlock victim, and NORMAL returns the session to the default deadlock-handling method. @deadlock_var has the character value 3 for LOW and 6 for NORMAL. Notice that there is not a HIGH (a deadlock always involves two processes; therefore, designating LOW will be enough to ensure the other session has the higher priority).

In the following example, we set the priority to LOW for our query:

```
SET DEADLOCK_PRIORITY LOW
GO
UPDATE Employees
SET dtupdateDate = GETDATE()
```

5.15 How to... Capture Deadlock Activity using SQL Profiler

All DBAs and developers should become familiar with SQL Profiler, as it is an excellent tool for capturing activity on your SQL Server instance.

SQL Profiler is able to capture data on the following deadlock-related events:

- Lock:Deadlock – captures deadlock event.
- Lock:Deadlock Chain – captures events leading up to deadlock.

Because you can see when a deadlock occurs by activating trace 1204, the first piece of information is less impressive. Where SQL Profiler excels, however, is in its ability to capture SQL events leading up to the deadlock. Unlike DBCC INPUTBUFFER, where you are limited to 255 characters, SQL Profiler can capture the entire statement. It can also capture the starting class of events, showing SQL statements that ran, but may not complete (because one of them was chosen as a deadlock victim).

Events to capture, depending on your SQL activity:

- SQL:BatchStarting – TSQL batch has started.
- SQL:StmtStarting – TSQL statement has started (lowest granularity).
- RPC:Starting – remote procedure call started.
- SP:Starting – stored procedure call started.
- SP:StmtStarting – statement within a stored procedure call started (lowest granularity).
Some of these events may show duplicate information, as statements vary in granularity, and \texttt{stmt} type events are smaller than batch and sp or \texttt{rpc} events. Batch and \texttt{stmt} types may capture the same statement (if one batch consists of one statement, for example); however, this will not hinder your ability to identify the deadlocked query.

To create a trace to capture deadlock activity and the statements leading up to it:

- Go to Start | Programs | Microsoft SQL Server | Profiler.
- Click the New Trace button:

- Enter the server and login information. Select OK.
- Type in a trace name if you wish, and select a file or table to which you will save your output. If saving to a file, you can enable file rollover at a maximum file size in megabytes. If you save your data to a table, you can set the maximum number of rows (in the thousands). Alternatively, you can decide to save the trace file later, after you have captured the necessary data. You can also enable a trace stop time if you prefer:
In the Event tab select the existing event classes and remove them by clicking the Remove button. Expand the Locks, Stored Procedures, and T-SQL event classes, and select the events we discussed earlier. Your screen should look like the example below:

Select the Data Columns tab. Click the Remove button to remove fields you do not need (such as duration, CPU, Reads, Writes), and add data columns you do need by selecting them in the left pane and clicking the Add button. Select only the fields that you will be interested in using, as each field adds to the size of the resulting trace file. Use the Up and Down buttons to rearrange the order of columns in your output:
Click the Filters tab. If you have a tremendous amount of activity on your server, you may want to filter database ID, login, or host name. You can also exclude system IDs by clicking the Exclude system IDs checkbox. Be careful not to filter out any activities that may be causing the deadlocking:

Click the Run button to begin.

Once a deadlock occurs, you can track the order of events. Here is an example of trace output:

<table>
<thead>
<tr>
<th>StartTime</th>
<th>TextData</th>
<th>SPID</th>
<th>EventClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002-07-18 15:18...</td>
<td>StartTrace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>SET TRANSACTION ISOLATION LEVEL READ...</td>
<td>51</td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>SET TRANSACTION ISOLATION LEVEL READ...</td>
<td>51</td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>BEGIN TRAN</td>
<td></td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>SELECT * FROM FactoryParts WHERE pa...</td>
<td>51</td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>WAITFOR DELAY '00:00:00'</td>
<td></td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>SET TRANSACTION ISOLATION LEVEL READ...</td>
<td>64</td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>SET TRANSACTION ISOLATION LEVEL READ...</td>
<td>64</td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>BEGIN TRAN</td>
<td></td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>SELECT * FROM FactoryParts WHERE pa...</td>
<td>54</td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>WAITFOR DELAY '00:00:00'</td>
<td></td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>UPDATE FactoryParts SET vendor_id='...</td>
<td>51</td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>UPDATE FactoryParts SET vendor_id='...</td>
<td>64</td>
<td>SQL:StartStarting</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>Deadlock Chain SPID = 61</td>
<td>4</td>
<td>Lock:Deadlock Chain</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>Deadlock Chain SPID = 64</td>
<td>4</td>
<td>Lock:Deadlock Chain</td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>54</td>
<td>Lock:Deadlock</td>
<td></td>
</tr>
<tr>
<td>2002-07-18 15:18...</td>
<td>COMMIT TRAN</td>
<td>51</td>
<td>SQL:StartStarting</td>
</tr>
</tbody>
</table>
Transactions, Locking, Blocking, and Deadlocking

Notice the last few rows showing the deadlock chain – designating SPIDs 51 and 54 as members of the deadlock:

| 2002-07-18 15:18... Deadlock Chain SPID = 51 | 4 | Lock:Deadlock Chain |
| 2002-07-18 15:18... Deadlock Chain SPID = 54 | 4 | Lock:Deadlock Chain |

Next notice the Lock:Deadlock event, with a SPID of 54, meaning this was the deadlock victim:

| 2002-07-18 15:18... | 54 | Lock:Deadlock |

If you look at the last SQL:StmtStarting event prior to the deadlock for SPID 54, you will see:

| 2002-07-18 15:18... | UPDATE FactoryParts SET ... | 54 | SQL:StmtStarting |
| 2002-07-18 15:18... | Deadlock Chain SPID = 51 | 4 | Lock:Deadlock Chain |
| 2002-07-18 15:18... | Deadlock Chain SPID = 54 | 4 | Lock:Deadlock Chain |
| 2002-07-18 15:18... | COMMIT TRAN | 51 | SQL:StmtStarting |

The lower gray pane will show the entire query (no 255 character limitation), designating the part of the statement that became a deadlock victim.

From here you can troubleshoot the application calls (see the next section for tips).

If you would like to save this trace file:

- In the File menu, select Save As | Trace File to save as a file.
- In the Save As window, select the file name and location of your .trc file. Select the Save button:
To save the trace to a table:

- Select File | Save As | Trace Table.
- Select the server and connection login. Select OK.
- Select the database and type in or select the destination table to which you wish to save the trace file. Select OK:

To save the template (thus saving your events, filters, and data columns so you can re-use the settings in future traces):

- Select File | Save As | Trace Template.
- Name your template something indicative of what it traces, such as Deadlock Detection. Select Save when finished:

- You will see the following dialog box:
Troubleshooting Blocking and Deadlocking Problems

We will now bring together all the technologies we have discussed in the three following deadlock troubleshooting scenarios.

**Scenario 1: Out of Order Updates**

In this first scenario, you notice deadlock error messages appearing in the SQL log throughout the day. You decide to turn on trace flags 1204, 3605, and -1 to get more detail about which process ends up being the deadlock victim, and which one survives (remember that beginning with SQL Server 2000 Service Pack 3, the trace flag 3605 is no longer necessary). A deadlock occurs again, and this time you have more detailed information (because of the trace flags).

You see in the **INPUTBUFFER** output in the SQL Log that the deadlock victim is a stored procedure call (execution) that updates the Orders and OrderDetails tables. The 'winner' is a different stored procedure, which updates the OrderDetails, and then Orders tables.

You check the stored procedures, and see that they are both updating the same tables, but in a different order. You decide to redesign the stored procedures to update the tables in the same order.
Scenario 2: Conflicting Updates During Busy Periods

During the evening, you download several million rows of data to an Orders table. Later, you notice that deadlocks are appearing in your SQL log. You decide to enable the trace flags 1204, 3605, and -1. You also decide to turn on SQL Profiler to trace the events as they occur, since they seem to occur predictably (every evening).

When the deadlocks occur again, you see that the victim is sometimes an UPDATE statement, and sometimes a SELECT statement. You have the exact time that these deadlocks occurred in the SQL error log, so you check the SQL Profiler for the activity that was running around this time.

You see that certain users are running reports against the data as it is being loaded. To reduce contention and deadlocking, you create a read-only copy of the table for reporting purposes, using transactional replication (see Chapter 8). This allows the data load to proceed without interruption, and reporting against the data to continue.

Scenario 3: JDBC Driver Settings

You have a new Web application running against your SQL Server instance. The database is very small, but has many users. The Web application is Java-based, and uses the Microsoft JDBC driver to connect.

As activity increases, you sometimes see long-term blocking. Every so often you see deadlocking. You turn on the trace flags and SQL Profiler.

The deadlock victims vary, sometimes being INSERT statements, sometimes UPDATE statements, and sometimes SELECT statements. You notice that the long-term blocking almost always precedes the deadlocks, and you ask the Java developer why this is happening.

She tells you that the user connections and updates should be taking no longer than 1 second. After much digging, she finds out that the autocommit function of her JDBC call is not enabled. The autocommit JDBC setting determines whether or not a transaction is explicitly committed. If FALSE, transactions that are not explicitly committed or rolled back may hold on to resources until the connection is closed.

As you have seen, SQL Server tools were insufficient to solve this problem. Sometimes you must look at the entire picture, asking other people questions about how the application is designed.

Tips for Avoiding and Minimizing Deadlocks

The following tips will help you to avoid or minimize deadlocks in your database:

- Keep transaction time short.
- Minimize resources locked.
- Update your tables in the same order.
- If you have relatively static data, or data that is loaded only periodically, consider placing your database into a read-only state. This removes the locking problem entirely.
- Watch SQL Server’s lock escalation behavior. If you see frequent escalations from row or page locks to table locks, consider adding locking hints to force either finer or coarser grain locks. SQL Profiler has the Lock:Escalation event to help you keep an eye on this.
- If the data is static or not updated frequently, consider using the NOLOCK hint, which places no locks on the resource. Do not use this if you need the data for mission-critical and accuracy-critical tasks.
- Do not allow user input during an open transaction. Also, make sure to pull data needed for the transaction BEFORE the transaction begins.
- Make sure to program mission-critical processes to restart or redo any transactions that were chosen as deadlock victims. This should be done only if you cannot solve the deadlocking problem.
- Use the `UPDLOCK` hint to prevent shared locks on data you are likely, but not certain, to update. This is better than an exclusive lock, as users can still read the data, but cannot put a lock on the data and risk a deadlock situation.
- SQL Server 2000 excels at set processing rather than cursor processing. Set processing works with blocks of rows at the same time, whereas cursor processing works with one row at a time. Transact-SQL cursors have a well-deserved reputation for causing blocking and performance issues. The reason is that, depending on the cursor fetch option, a cursor can hold many locks while it runs a row-by-row processing routine.
- The lower the isolation level you are able to use, the better.
- Using `bound connections` allows two or more connections to act as if they are part of the same connection, allowing them to update resources held by one another.
- Normalized databases are suggested for transaction processing environments. Databases that are normalized well by design help improve concurrency and reduce blocking and deadlocking, although a bad query can still be enough to cause major blocking or deadlocking problems.