

User name:

Book: SQL in a Nutshell, 2nd Edition

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

5.4. Managing Transactions

Most database programming APIs provide methods of controlling transactions, setting savepoints, and modifying isolation levels. This section covers the mechanisms for controlling transactions from the ADO.NET and JDBC APIs covered in this chapter.

5.4.1. Beginning a Transaction

Beginning a transaction is the first step in guaranteeing atomicity when executing multiple SQL statements. After beginning a transaction, the transaction can be committed to make the changes performed by the executed SQL statements permanent or the transaction can be rolled back to abort the changes and leave the database unchanged.

5.4.1.1. Beginning an ADO.NET transaction

To begin a transaction using ADO.NET, invoke the `BeginTransaction` method on a connection object. A `Transaction` object is returned that can be used in the creation of ADO.NET `Command` objects that execute within the same transaction. To execute a `Command` object within the new transaction, you must first attach the `Transaction` object to the `Command` object's `Transaction` property. Following is the syntax for starting a transaction by creating a `Transaction` object in ADO.NET:

```
{Odbc|OleDb|Sql}Transaction transaction =  
    connection.BeginTransaction([IsolationLevel.  
        {Chaos | ReadCommitted | ReadUncommitted | RepeatableRead |  
          Serializable | Unspecified}]);  
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
statement.Transaction = transaction;
```

When creating the `Transaction` object, you can optionally specify the *isolation level* that is used for the transaction. The isolation level controls how much (or little) your database transactions are insulated from the effects of other transactions. The available isolation levels are:

Chaos

Pending changes of other transactions with higher isolation levels (`ReadCommitted`, `ReadUncommitted`, `RepeatableRead`, or `Serializable`) cannot be changed.

ReadUncommitted

The lowest ANSI standard isolation level, unprotected against dirty reads, non-repeatable reads, and phantom records.

ReadCommitted

The next highest ANSI standard isolation level above `ReadUncommitted`, protects against dirty reads.

RepeatableRead

The next highest ANSI standard isolation level above `ReadCommitted`, protects against dirty reads and

nonrepeatable reads.

Serializable

The highest ANSI standard isolation level. Protects against dirty reads, non-repeatable reads, and phantom records.

Unspecified

Isolation level cannot be determined. This option is not typically passed into the `BeginTransaction` method; its purpose is to provide an Unknown state for the `Connection` object's `IsolationLevel` property. If the `Transaction` has this `IsolationLevel`, even after being set to another value, then it is safe to assume that the database does not support the `IsolationLevel` that you requested.

For more information about isolation levels, see [Programming Tips and Gotchas](#) under `SET TRANSACTION Statement`.



The isolation level used during SQL statement execution can have a significant impact on the performance and scalability of a database application, as well as on the RDBMS itself. When deciding which isolation level to use, choose the lowest level that provides the appropriate isolation guarantees for the database application.

5.4.1.2. Beginning a JDBC transaction

JDBC starts a connection in *AUTO COMMIT* mode, which is a setting that automatically commits changes made by each SQL statement when it executes. To gain control over when changes are committed, the *AUTO COMMIT* setting must be turned off. JDBC's mechanism of beginning a transaction is subtle compared to ADO.NET, which uses a `BeginTransaction` method to begin a transaction. In JDBC, a transaction is started as soon as the *AUTO COMMIT* mode is turned off or just after the transaction has been committed or rolled back. Committing or rolling back the transaction doesn't turn the *AUTO COMMIT* mode back on, so multiple commits or rollbacks will need to be executed to commit/roll back multiple units of work until the connection is closed or *AUTO COMMIT* is explicitly turned back on. To turn off the *AUTO COMMIT* mode, execute the `setAutoCommit` method of the JDBC `Connection` object with a false argument:

```
connection.setAutoCommit( false );
```



Beware that invoking `setAutoCommit` to turn *AUTO COMMIT* mode on or off when a transaction has already begun will automatically commit that transaction.

JDBC also supports different isolation levels through the JDBC `Connection` interface. To set the isolation level used for new transactions, invoke the `setTransactionIsolation` method with one of the following isolation levels:

TRANSACTION_NONE

Indicates transactions are not supported on the connection. This value is not typically used to set the isolation level, but to query the connection for transaction support with the `getTransactionIsolation` method on the `Connection` interface.

TRANSACTION_READ_UNCOMMITTED

The lowest ANSI standard isolation level, unprotected against dirty reads, nonrepeatable reads, and phantom records.

TRANSACTION_READ_COMMITTED

The next highest ANSI standard isolation level above [TRANSACTION_READ_UNCOMMITTED](#). Protects against dirty reads.

TRANSACTION_REPEATABLE_READ

The next highest ANSI standard isolation level above [TRANSACTION_READ_COMMITTED](#). Protects against dirty reads and nonrepeatable reads.

TRANSACTION_SERIALIZABLE

The highest ANSI standard isolation level. Protects against dirty reads, nonrepeatable reads, and phantom records.

For a more detailed explanation of transaction isolation levels, see [Programming Tips and Gotchas](#) under [SET TRANSACTION Statement](#).

Following is example code setting the isolation level in JDBC:

```
connection.setIsolationLevel( Connection.TRANSACTION_SERIALIZABLE );
```



The isolation level used on the connection can have a significant impact on the performance and scalability of a database application, as well as on the RDBMS itself. When deciding which isolation level to use, choose the lowest level that provides the appropriate isolation guarantees for the connection.

5.4.2. Committing a Transaction

Committing a transaction is a way to explicitly close the transaction and make its database modifications permanent.

5.4.2.1. Committing an ADO.NET transaction

To commit a transaction using ADO.NET, invoke the [Commit](#) method on the [Transaction](#) object:

```
transaction.Commit( );
```

ADO.NET is more object-oriented than many database-programming APIs, as can be seen from the encapsulation of a transaction into a unique object type. This has the drawback of making the methods for explicitly ending a transaction more difficult to find than those to begin a transaction, since they are on different object types. Remember that the method to begin a transaction is on the [Connection](#) object, and the methods to commit or roll back a transaction are found on the [Transaction](#) object.

5.4.2.2. Committing a JDBC transaction

Invoking the [commit](#) method on a JDBC [Connection](#) object commits a transaction within JDBC and begins a new transaction:

```
connection.commit( );
```

If the connection is in *AUTO COMMIT* mode, you will then get a Java exception of type [SQLException](#) because the connection will not have a pending transaction.

5.4.3. Rolling Back a Transaction

Rolling back a transaction is a way to close the transaction explicitly and to discard any database modifications since the transaction was started.

5.4.3.1. Rolling back an ADO.NET transaction

To roll back a transaction using ADO.NET, invoke the `Rollback` method on the `Transaction` object:

```
transaction.Rollback( );
```

After rolling back a `Transaction` object, the `Dispose` method should be invoked to free any resources held by the transaction:

```
transaction.Dispose( );
```

5.4.3.2. Rolling back a JDBC transaction

Invoking the `rollback` method on a JDBC `Connection` object rolls back a transaction within JDBC and begins a new transaction:

```
connection.rollback( );
```

If the connection is in *AUTO COMMIT* mode, then a Java exception will be thrown of `SQLException` type.