The Programmer's Guide to SQL

CRISTIAN DARIE, KARLI WATSON
WITH CHRIS HART, KEVIN HOFFMAN, JULIAN SKINNER

Apress™

ISBN (pbk): 1-59059-218-2

Printed and bound in the United States of America 10987654321

Technical Reviewers: Cristof Falk, Slavomir Furman, Brad Maiani, Judith M. Myerson, Johan Normén, David Schultz

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Lead Editor: Tony Davis

Assistant Publisher: Grace Wong

Project Manager: Darren Murphy

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Kelly Winquist

Proofreader: Thistle Hill Publishing Services

Compositor: Kinetic Publishing Services, LLC

Indexer: John Collin

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 10

Transactions

Although it's generally considered to be an advanced topic, the concept of transactions is easy to understand because it models natural processes that happen in our everyday lives.

Transactions aren't necessarily related to financial operations, but this is the simplest way to look at them. For example, every time you buy something, a transaction happens—you pay an amount of money and then receive in return the product for which you paid. The two operations (paying the money and receiving the product) form the transaction.

Imagine that a customer called Sally wants to transfer $1,000 from her checking account to her savings account. From Sally's point of view, this operation is a single operation (*one* money transfer operation), but in fact inside the database something like this happens:

```
UPDATE Checking
   SET Balance = Balance - 1000
   WHERE Account = 'Sally';
UPDATE Savings
   SET Balance = Balance + 1000
   WHERE Account = 'Sally';
```

Of course, this piece of SQL code is a simplified version of what really happens inside a production database, but it demonstrates how a set of simple operations can form a single larger operation. In this example you have only two operations, but in reality there are likely to be many more. When you perform such a complex operation, you need to be assured it really executes as a single, atomic operation.

This isn't as easy as it sounds because you need to make sure all the constituent operations execute successfully—if any of them don't (and this can happen because of a wide area of reasons such as operating system or database software crashes, computer viruses, or hardware crashes), the larger operation fails.

In the bank scenario, two database operations need to happen in order to correctly transfer the money—if one of the two operations fails (say, a hardware failure happens after the first `UPDATE`), Sally would lose $1,000. You need to find a way to ensure the two `UPDATE` statements both either execute successfully or don't execute at all.

Fortunately, modern databases have the technology to protect transactions for you—you just need to tell the database about the statements that form a transaction, and the database will do everything for you.

In this chapter, you'll learn what transactions are and how they're implemented in databases. More specifically, this chapter answers the following questions:

- What are transactions? Why do you need them?
- What are the rules that transactions must follow?
- How can you implement database transactions?
- What are the performance problems associated with database transactions, and what are the best practices to avoid them?

This chapter also looks at the differences between the database platforms that you've been working with when it comes to implementing transactions.

Let's start with the basics....

What Is a Transaction?

Transactions are all about data being transformed from one state to another. Before giving an accurate definition for transactions, let's look at yet another scenario involving transactions—this time, an example that's not necessarily related to databases. Imagine what happens when a user who is visiting an e-commerce shop decides to buy the items in his or her shopping cart. From the user's perspective, all that's required is a simple action such as clicking the Order Items button. However, the actual processes that happen behind the scenes could be something like the following:

1. Create a new order in the database by assigning an identifier, storing a customer ID and storing the current date.
2. Clear the customer's shopping cart.
3. Charge the customer's credit card.

4. Send a request to the supplier (or the warehouse) that the ordered items be shipped to the customer.

5. Update database statistics and perform various other operations depending on the way the e-shop is designed to work.

6. Finally, send the customer a confirmation e-mail containing details about his or her order.

Of course, this list of individual tasks is purely hypothetical, but it gives you an idea about the work that needs to be done after the customer clicks a single button. This is another example of a complex operation that's composed of many smaller ones.

You need to be sure that the specified operations execute either in their entirety or not at all. The process outcome should be black or white, success or failure, nothing in between.

So, in more formal terms, a transaction is a logical unit of work consisting of a sequence of separate operations that brings a system from one consistent state to another. The data should be, at any moment, in a consistent state. The transaction is successful if all of its constituent operations are successful. If any of the operations fail (because of any kind of software or hardware problem), all the changes made by the successful operations are reversed, and the system is brought back to its original state. A transaction can either succeed or fail, but nothing in between. In both cases, the system is brought to a consistent state.

If a problem occurs, all the successful operations are reversed, and everything goes back to the way it was before starting the transaction. This process of reversing the successful operations is called a *transaction rollback*.

If, instead, everything goes just fine and all operations complete successfully, the changes are declared to be permanent, and you say that the transaction has been *committed*. After a transaction is committed, it can't be rolled back.

Transactions (not only the database ones) must conform to a set of properties, collectively known as the *ACID properties*.

The ACID Properties

ACID is an acronym used to describe the four properties of a transaction:

Atomicity refers to the all-or-nothing nature of a transaction: The transaction executes completely, or it doesn't execute at all. In the case of a failed transaction, all the successful individual actions need to be reverted, and the transaction is rolled back to the original state. Atomicity is perhaps the most representative property of transactions and it's typically used when describing and defining transactions. Nevertheless, the other three properties are just as important as this one.

Consistency refers to the fact that a transaction must take the system from one consistent state to another consistent state. That is, the transaction as a whole must not break any business rules specified for the environment. Contrary to the atomicity property, consistency isn't a rule that the database system can take care of: You, the programmer, must make sure the system gets to a consistent state (in respect to the rules that you define) after the transaction successfully completes.

Isolation specifies that each transaction should perform independently of the other transactions and operations that happen at the same time. In practice, this means that the outside world shouldn't see individual changes to the database while the transaction is happening. In other words, changes made by the active transaction aren't visible to other concurrently running transactions—this is essential, especially because you can't know ahead of time if the transaction will be successful. The changes made by a transaction are declared permanent and become visible to the other transactions and processes only *after* the transaction is committed.

Durability ensures that after the transaction is completed, its outcome is persisted and resists even in the event of a system failure. So, after a transaction is committed, you can be sure its results are persisted even if the power goes down one second later.

Understanding Database Transactions

In the case of database transactions, the group of actions that you attempt to execute (a process that either succeeds or fails—the all-or-nothing proposition) are SQL statements, and the system that needs to be kept in a consistent state is the database itself.

Luckily enough, SQL-99 has support for transactions, as do the major database systems. Otherwise, you would need to manually enforce the ACID rules, which would be a tough job!

In order to implement transactions, databases keep log files with everything that happens inside the transaction. When rolling back the transaction, all the successful operations are reversed based on the data in the log files, and the affected data is brought back to its previous state.

NOTE *The way log files work is considered an advanced topic and will not be covered in this book. Moreover, because each database system has its own particularities regarding this subject, you would be best off consulting the documentation for your database platform.*

SQL Server, Oracle, DB2, and MySQL do support transactions and the ACID rules. Access doesn't support transactions.

The Typical Database Transaction

Transactions work differently with each Relational Database Management System (RDBMS), but there are some concepts and steps common to all transactions. We'll describe how transactions work and then give specific examples for each RDBMS, discussing the particularities of each database platform.

Beginning the Transaction

Transactions must have a clearly defined start and end point. The point the transaction starts is very important—it's the point you can roll back to in case some failure or problem occurs in any of its constituent SQL statements. In respect to the consistency rule, at the moment a new transaction is created, the data must be in a consistent state.

The SQL-99 standard specifies the `START TRANSACTION` statement, which should mark the point at which a new transaction starts, but this command isn't implemented in the database platforms covered in this book.

Oracle and DB2 start transactions automatically. In other words, a transaction automatically begins as soon as you execute the first SQL statement. With SQL Server, you use the `BEGIN TRANSACTION` statement, which can also accept a transaction name, and with MySQL you simply use `BEGIN`. You'll see more details in the examples later in this chapter.

Executing the SQL Statements

Whether the statements execute correctly will determine whether the transaction is committed or rolled back. You'll usually need a mechanism that can test whether all the SQL statements ran without problems so that you can decide how to end the transaction (to commit it if everything went okay or to roll back if it didn't go okay).

Implementing a testing mechanism is important because, by default, the transaction is not rolled back if a noncritical error occurs. Let's take another look at the checking/savings accounts example:

```
UPDATE Checking
   SET Balance = Balance - 1000
   WHERE Account = 'Sally';
UPDATE Savings
   SET Balance = Balance + 1000
   WHERE Account = 'Sally';
```

If these two statements are part of a transaction, and one of them generates an error because it can't be executed successfully, the transaction will most likely not be rolled back by default. Instead, you need to manually test whether each statement performed successfully; if either of them didn't, you can roll back the transaction. Also, if everything runs okay, you need to manually to commit the transaction.

In this chapter's examples, you'll see how to test if any of the statements generated errors and how to react to them.

Rolling Back the Transaction

As you saw earlier, when problems occur inside the transaction, you can roll it back to its starting point or to a *savepoint* if the database system supports savepoints (we'll cover these in more detail a little later).

The SQL-99 syntax for rolling back transactions is as follows:

```
ROLLBACK [WORK] [TO SAVEPOINT savepoint_name]
```

This structure is fully supported by Oracle, DB2, and partially by MySQL, which doesn't support savepoints.

The SQL Server syntax is a bit different, but it serves the same purpose:

```
ROLLBACK TRANSACTION [<transaction name>|<savepoint name>]
```

What's important to keep in mind is that rolling back a transaction brings the data it has affected to its previous state and closes the transaction, but the execution of the batch continues normally. For example, if you have a `ROLLBACK` statement in the middle of a stored procedure, the execution doesn't stop at `ROLLBACK` (so `ROLLBACK` isn't like a `RETURN` or a `GOTO` command, and it doesn't move the execution pointer). If you want the stored procedure to stop executing when you do a `ROLLBACK`, you need to manually handle this—you'll see how to accomplish this in the upcoming examples.

Committing the Transaction

If all of the SQL commands in the transaction execute successfully (or in a way you consider to be okay), you issue a `COMMIT` command. This tells the database to persist the changes made by the transaction in the database. From this moment, the changes can't be undone using a `ROLLBACK` command.

The SQL-99 syntax for committing transactions is as follows:

```
COMMIT [WORK]
```

This syntax is supported by all major database vendors, but some of them also accept additional parameters. SQL Server supports `COMMIT WORK` but also has a `COMMIT TRANSACTION` command that accepts as a parameter the name of the transaction to be committed (SQL Server supports having transaction names).

Particulars of Database Transactions

You'll now look at some features that aren't supported by all database systems covered in this book or that are supported differently. First, we cover the concepts, and then we'll show how to apply them in a few examples.

Autocommit

A database that runs in autocommit mode will treat every SQL query as a separate transaction, without needing any additional SQL commands such as `BEGIN`, `COMMIT`, or `ROLLBACK`. After a SQL data modification statement is executed, the changes are automatically committed (so the results are considered final) by the database system.

NOTE *SQL Server and MySQL work by default in autocommit mode.*

With a database that works in autocommit mode, these two statements will be considered as two separate transactions:

```
UPDATE Checking
   SET Balance = Balance - 1000
   WHERE Account = 'Sally';
UPDATE Savings
   SET Balance = Balance + 1000
   WHERE Account = 'Sally';
```

While in autocommit mode, if you want to start a transaction formed by more than one SQL command, you must use a `BEGIN` command and then manually finish the transaction with either `ROLLBACK` or `COMMIT`. So if you want to have a single transaction containing the previous two SQL statements, you need to do something like this:

```
BEGIN WORK
UPDATE Checking
   SET Balance = Balance - 1000
   WHERE Account = 'Sally';
UPDATE Savings
   SET Balance = Balance + 1000
   WHERE Account = 'Sally';
COMMIT WORK
```

However, note that here you didn't do any checking to see whether either of the two `UPDATE` statements executed successfully. In a real-world example, you'd need to implement some error handling mechanism and `ROLLBACK` the transaction in case an error happens.

If the database *isn't* in autocommit mode, it's said to be in *automatic-transactions* mode.

NOTE *Oracle and DB2 work by default in automatic-transactions mode.*

In this mode, a new transaction starts automatically with the first SQL query you type, but it's not automatically committed. In other words, the database system works as if you had already typed a `BEGIN` command and then waits for you to call `COMMIT` or `ROLLBACK` to finish the transaction. With a database working in automatic-transactions mode, you can integrate the previous two `UPDATE` commands in a transaction like this:

```
UPDATE Checking
   SET Balance = Balance - 1000
   WHERE Account = 'Sally';
UPDATE Savings
   SET Balance = Balance + 1000
   WHERE Account = 'Sally';
COMMIT WORK
```

SQL Server and MySQL work by default in autocommit mode while Oracle and DB2 work by default in automatic-transactions mode, but in all cases the default mode can be changed (although most developers prefer to keep the default mode of the database system).

For Oracle, you change the default mode with `SET AUTOCOMMIT ON/OFF`. With SQL Server you use `SET IMPLICIT_TRANSACTIONS ON/OFF`, and with MySQL you use `SET AUTOCOMMIT=0/1`. When working with DB2, you can handle this by selecting Command Center ➤ Options from the main menu of the Command Center tool and then checking or unchecking the Automatically Commit SQL Statements box in the Execution tab as appropriate (see Figure 10-1).

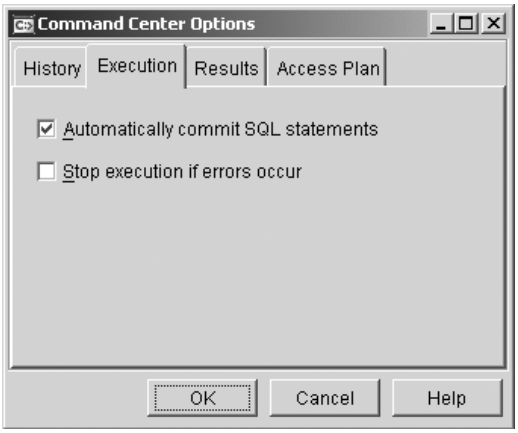

Figure 10-1. Choosing to automatically commit SQL statements

CAUTION *It's possible that your data access provider (for example, JDBC or OLEDB) will be working in autocommit mode. If you want to issue SQL statements in automatic-transactions mode, then you may have to alter the default behavior of your provider as well.*

In general, because of the *isolation* property of transactions, other users will not be able to see your changes if you forget to send a `COMMIT` command. Be sure

to always issue COMMIT statements after an update, insert, or delete when working in automatic-transactions mode.

Savepoints

A *savepoint* acts as a bookmark inside a transaction. You can roll back any actions that have been performed by a transaction to a certain savepoint, without actually closing the transaction. Rolling back to a certain savepoint brings the database to the status it was when the savepoint was created, but it doesn't end the transaction. A final ROLLBACK or COMMIT is still required.

NOTE *Savepoints are supported by Oracle, DB2, and SQL Server. MySQL supports transactions but doesn't support savepoints.*

SQL-99 specifies the SAVEPOINT command to create savepoints, and this command is implemented in Oracle and DB2. Because you can have multiple savepoints in a single transaction, you always need to supply a savepoint name:

```
SAVEPOINT <savepoint name>
```

SQL Server uses SAVE TRANSACTION instead of SAVEPOINT:

```
SAVE TRANSACTION <savepoint name>
```

This is how you would add a savepoint to the bank transactions example using the SQL Server syntax:

```
BEGIN TRANSACTION

SAVE TRANSACTION BeforeChangingBalance
UPDATE Checking
   SET Balance = Balance - 1000
   WHERE Account = 'Sally';
ROLLBACK TRANSACTION BeforeChangingBalance

UPDATE Savings
   SET Balance = Balance + 1000
   WHERE Account = 'Sally';

COMMIT TRANSACTION
```

After executing this batch, Sally will have $1,000 more in her account. The first UPDATE statement is rolled back (but the transaction still remains active!), and finally the transaction is committed.

We'll see examples with SQL Server, Oracle, and DB2 savepoints in the following sections.

Transactions at Work

Here, we'll provide a closer look at how transactions are supported by the database products you've been investigating so far.

SQL Server

SQL Server works by default in autocommit mode, with each SQL command treated as a separate transaction, unless you use BEGIN TRANSACTION to start a multistatement transaction.

TIP *You can set SQL Server to work with implicit transactions (by turning off autocommit mode) by using* SET IMPLICIT_TRANSACTIONS ON. *When working in implicit transactions mode, SQL Server assumes the* BEGIN TRANSACTION *command before the first SQL query is issued. In the following examples, we assume you'll work with the default transactions mode of SQL Server, which requires you to manually start multistatement transactions with* BEGIN TRANSACTION.

The statements that deal with transactions in SQL Server are as follows:

```
BEGIN TRANSACTION [<transaction name>]
COMMIT TRANSACTION [<transaction name>]
ROLLBACK TRANSACTION [<transaction name>|<savepoint name>]
SAVE TRANSACTION <savepoint name>
```

The TRANSACTION keyword can be optionally replaced by its shorter form, TRAN.

BEGIN, COMMIT, and ROLLBACK receive an optional transaction name. Using names for transactions doesn't bring any new functionality, but it can help for

readability if you use suggestive names, especially in stored procedures or batches, that have more transactions.

If ROLLBACK doesn't have a parameter or if the parameter specifies a transaction name, the whole transaction is rolled back. If a savepoint name is received instead, the transaction is rolled back to the specified savepoint.

Note that after rolling back to a savepoint, the transaction is still active. It still needs a final ROLLBACK or COMMIT, with either no parameter or with the transaction's name as a parameter to finalize the transaction.

EXAMPLE: SQL SERVER TRANSACTIONS AND SAVEPOINTS

In this example, you'll see a simple SQL Server transaction. In this transaction, you'll add two new students to the Student table from the InstantUniversity database. However, the second INSERT operation is rolled back using a savepoint. Finally, the transaction is committed:

```
BEGIN TRANSACTION MyTransaction;
INSERT INTO Student (StudentID, Name) VALUES (98, 'Good Student');
SAVE TRANSACTION BeforeAddingBadStudent;
INSERT INTO Student (StudentID, Name) VALUES (99, 'Bad Student');
ROLLBACK TRANSACTION BeforeAddingBadStudent;
COMMIT TRANSACTION MyTransaction;
SELECT * FROM Student;
```

The example starts by declaring the new transaction with this well-known command:

```
BEGIN TRANSACTION MyTransaction
```

Then you add Good Student to the Student table:

```
INSERT INTO Student (StudentID, Name) VALUES (98, 'Good Student')
```

Then you add yet another student to the table. However, the operation is rolled back using the BeforeAddingBadStudent savepoint:

```
SAVE TRANSACTION BeforeAddingBadStudent
INSERT INTO Student (StudentID, Name) VALUES (99, 'Bad Student')
ROLLBACK TRANSACTION BeforeAddingBadStudent
```

Practically, this piece of code actually does nothing because the changes are annulled. Also, after rolling back to a savepoint, the transaction is still active, so

you still have the power to commit it or roll it back completely. In the end, you commit the transaction and display the contents of the `Employee` table:

```
COMMIT TRANSACTION MyTransaction
SELECT * FROM Student
```

If you had rolled back the transaction instead of committing it, the changes done by both `INSERT` statements would have been void, and the `Student` table would have ended up being just as it was before starting the transaction.

SQL Server Transactions and Error Handling

The first example was interesting enough, but it didn't show how to roll back the whole transaction in case something bad happens in any of the queries (that was the whole point after all, right?).

Let's see how to do that in another example.

EXAMPLE: SQL SERVER TRANSACTIONS WITH ERROR HANDLING

In this example, you'll insert a number of rows into the `Student` table as part of a transaction, and after each insert, you'll check whether an error occurred. If it did, you'll roll back the transaction. If all the inserts succeed, you commit the transaction. In both cases, you display a message saying whether the transaction was committed or rolled back:

```
BEGIN TRANSACTION MyTransaction

INSERT INTO Student (StudentID, Name) VALUES (101, 'Dave')
IF @@ERROR != 0
BEGIN
   ROLLBACK TRANSACTION MyTransaction
   PRINT 'Cannot insert Dave! Transaction rolled back.'
   RETURN
END
```

```
INSERT INTO Student (StudentID, Name) VALUES (102, 'Claire')
IF @@ERROR != 0
BEGIN
   ROLLBACK TRANSACTION MyTransaction
   PRINT 'Cannot insert Claire! Transaction rolled back.'
   RETURN
END

INSERT INTO Student (StudentID, Name) VALUES (103, 'Anne')
IF @@ERROR != 0
BEGIN
   ROLLBACK TRANSACTION MyTransaction
   PRINT 'Cannot insert Anne! Transaction rolled back.'
   RETURN
END

COMMIT TRANSACTION MyTransaction
IF @@ERROR != 0
   PRINT 'Could not COMMIT transaction'
ELSE
   PRINT 'Transaction committed.'
```

This example looks a bit different from the previous one. The batch of statements tries to insert three records to the `Student` table. If any of the `INSERT` statements fail, you roll back the transaction, display a message on the screen, and stop the execution.

You achieve this by verifying the `@@ERROR` system variable after each statement that could generate an error:

```
INSERT INTO Student (StudentID, Name) VALUES (101, 'Dave')
IF @@ERROR != 0
BEGIN
   ROLLBACK TRANSACTION MyTransaction
   PRINT 'Cannot insert Dave! Transaction rolled back.'
   RETURN
END
```

The `@@ERROR` system variable is automatically updated after each SQL query, so it is necessary to test it each time you do something to the database. Another important thing to note is that you call `RETURN` after rolling back the transaction. This wouldn't be necessary if you had rolled back to a savepoint, but if you roll back the entire transaction, the execution should stop; otherwise, the following SQL statements would mistakenly assume that they're part of a transaction and call `ROLLBACK` or `COMMIT` at certain points.

You can also test the outcome of the COMMIT command. This way you deal with the case where there's a problem and the transaction can't be committed. The most likely reason for this is if you already finalized the transaction (say, rolled back the transaction when an INSERT failed without calling RETURN after ROLLBACK). This is a great debugging technique because it provides feedback about the transaction's output, but once you're confident that the SQL script is well constructed, you can omit this:

```
COMMIT TRANSACTION MyTransaction
IF @@ERROR != 0
   PRINT 'Could not COMMIT transaction'
ELSE
   PRINT 'Transaction committed.'
```

The first time the batch is executed, the output should read as follows:

```
Transaction committed.
```

However, if you now execute the batch again, the transaction will be rolled back because you can't insert two rows with the same primary key value into a table:

```
Violation of PRIMARY KEY constraint PK__Student__59063A47'.

Cannot insert duplicate key in object 'Student'.
The statement has been terminated.
Cannot insert Dave! Transaction rolled back.
```

Oracle

Oracle transactions work by default in automatic-transactions mode. In other words, it starts a new multistatement transaction with the first SQL query you type. This mode is used by default in DB2 as well, and it's the mode used in these examples.

COMMIT and ROLLBACK are still your best friends when working in automatic-transactions mode. You create savepoints using the SAVEPOINT command, and if you want to roll back to a savepoint, you must use ROLLBACK TO <*savepoint name*>.

EXAMPLE: ORACLE TRANSACTIONS AND SAVEPOINTS

To start a new transaction, you just need to start executing SQL queries. In this example, you'll add two new students to the Student table. However, the second INSERT operation is rolled back using a savepoint. Finally, you commit the transaction:

```
INSERT INTO Student (StudentID, Name) VALUES (98, 'Good Student');
SAVEPOINT BeforeAddingBadStudent;
INSERT INTO Student (StudentID, Name) VALUES (99, 'Bad Student');
ROLLBACK TO BeforeAddingBadStudent;
COMMIT;
```

You added two new students to the Student table. However, before adding the second one, you created a savepoint—which was then used as a point to which you rolled back the transaction. Right now, if you type SELECT * FROM Student, in addition to the original rows, you'll see a single new row, containing Good Student.

Oracle Transactions and Exception Handling

Oracle has a robust and powerful exception handling system, which proves to be helpful when dealing with real-world transactions. When an error occurs in your code, the transaction needs to be rolled back, and the other SQL statements shouldn't execute anymore.

You saw in the SQL Server example that one way to deal with this is to check the outcome of each SQL statement and deal with them separately. Oracle has another way of dealing with this.

EXAMPLE: ORACLE TRANSACTIONS AND EXCEPTION HANDLING

In this example, you'll try to insert three rows into the `Student` table. One of the inserts will be rolled back using a savepoint. If any exceptions are raised in the process, you roll back the whole transaction and bring the `Student` table back to its original state.

Here's the piece of code that does this:

```
BEGIN

   INSERT INTO Student (StudentID, Name) VALUES (101, 'Dave');
   INSERT INTO Student (StudentID, Name) VALUES (102, 'Claire');

   SAVEPOINT BeforeAddingAnne;
   INSERT INTO Student (StudentID, Name) VALUES (103, 'Anne');
   ROLLBACK TO BeforeAddingAnne;

   COMMIT;

EXCEPTION
   WHEN OTHERS
      THEN ROLLBACK;
END;
/
```

After typing the batch, you can save it using `SAVE`:

```
SAVE exception
```

Once the procedure is saved to a file, you can call it like this:

```
@exception
```

If any of the SQL queries in the `BEGIN` block generates an error, the execution is passed to the `EXCEPTION` block:

```
EXCEPTION
   WHEN OTHERS
      THEN ROLLBACK;
```

You can handle separately many different kinds of predefined exceptions, and you can also define and raise your own exceptions. Also, if you have nested BEGIN blocks or stored procedures, the unhandled exceptions propagate vertically. In this case, you have used the WHEN OTHERS option, which handles all exceptions. There you use ROLLBACK to roll back the transaction. You can learn more about Oracle exception handling in Chapter 9.

The procedure you wrote would end up adding two rows into the Student table, as long as no errors are generated inside the script. If an error does occur, everything is brought back to the original state. If, for example, you already have a student with a StudentID of 102 before executing the script, an error will be generated when trying to add Claire, and the transaction is rolled back (so not even Dave will end up being in the Student table).

After you've run this code once, each subsequent time you execute @exception, the transaction will be rolled back.

DB2

DB2 supports the COMMIT, ROLLBACK, and SAVEPOINT commands. Like Oracle, DB2 works in automatic-transactions mode by default.

To switch between automatic-transactions mode and autocommit mode, you need to alter the Command Center options, as shown earlier in this chapter. For the purpose of this example, clear the Automatically Commit SQL Statements checkbox. So, let's look at an example.

EXAMPLE: DB2 TRANSACTIONS

In this example, you'll add two new students to the Student table—a good student and a bad student. You reverse the addition of the bad student by using a savepoint:

```
INSERT INTO Student (StudentID, Name) VALUES (98, 'Good Student');
SAVEPOINT BeforeAddingBadStudent ON ROLLBACK RETAIN CURSORS;
INSERT INTO Student (StudentID, Name) VALUES (99, 'Bad Student');
ROLLBACK TO SAVEPOINT BeforeAddingBadStudent;
COMMIT;
```

Notice the way you create the savepoint:

```
SAVEPOINT BeforeAddingBadStudent ON ROLLBACK RETAIN CURSORS;
```

The `ON ROLLBACK RETAIN CURSORS` statement ensures that your code will continue to execute from the point where the rollback was called after the rollback has been performed.

MySQL

MySQL does have support for transactions, but only if you know how to create your data tables. Yes, this sounds a bit weird, but it's true.

The MySQL engine supports more internal data storage formats for its data tables. When creating a new data table with `CREATE TABLE`, the default table type is used, which is `MyISAM`. This table type is pretty basic and doesn't support features such as transactions or the capability to enforce referential integrity through foreign keys, but it's the fastest one available for MySQL.

In total, MySQL supports at least five table types: `MyISAM`, `HEAP`, `ISAM`, `BDB`, and `InnoDB`. For detailed information about each table type supported by MySQL, please visit `http://www.mysql.com/doc/en/Table_types.html`.

TIP *MySQL documentation recommends using the default table type,* `MyISAM`*, if transactions aren't required because it works much faster without the overhead of keeping a transaction log.*

The important fact to understand, for the purposes of this chapter, is that `BDB` and `InnoDB` are the only types that support transactions. `InnoDB` fully supports the ACID properties, and as such, we'll be using this table type here. You can learn more about this table type at `http://www.mysql.com/doc/en/InnoDB.html`.

By default, MySQL works in autocommit mode (like SQL Server does). Each update on the database is committed immediately, without the need to call `COMMIT`. To explicitly start a multistatement transaction, you use `BEGIN`, and you finish the transaction with either `COMMIT` or `ROLLBACK`. MySQL doesn't support savepoints.

For transaction-safe tables (`BDB` and `InnoDB`), you can also instruct MySQL to work in non-autocommit mode (just like Oracle works by default) using `SET AUTOCOMMIT=0`.

EXAMPLE: MYSQL TRANSACTIONS

Let's look at a simple example with MySQL transactions. Start a new transaction with `BEGIN`, add two new students to the `Student` table, and then roll back the transaction. Open a new MySQL console, open the `InstantUniversity` database, and type the following:

```
BEGIN;

INSERT INTO Student (StudentID, Name) VALUES (98, 'Anne');
INSERT INTO Student (StudentID, Name) VALUES (99, 'Julian');

ROLLBACK;
```

Now, read the `Student` table, and you'll see that it has no new rows—the transaction was successfully rolled back.

Access

This is really simple: Microsoft Access doesn't support transactions, so there's not a lot to discuss here!

Moving onto Advanced Topics

Having experimented with some simple transactions, we're now going to present further issues involved in writing advanced transaction code. The basics of transactions that you've looked at so far form the foundation for what you'll be looking at here as you consider the wider implications of executing transactions in the real world.

We'll discuss the different *transaction isolation levels* that you can use. These define certain rules that govern the degree of "interaction" between multiple transactions acting on the same data. We'll also discuss the use of database *locks*—one of the mechanisms by which you can control concurrent access to shared resources in the database.

There are several different isolation levels and many different types of lock that can be applied, depending on your specific RDBMS. It's way beyond the scope of this chapter to provide a definitive guide to transactions and locking for each RDBMS. Instead, we aim to provide a good general understanding of the requirements that apply to each level, how you often have to use locks to meet those requirements, and the potential performance consequences of locking database resources.

Concurrency and Transaction Isolation Levels

It's a fact that, most of the time, you can't guarantee that transactions will execute one at a time—on the contrary, in complex databases it's likely that many transactions will run concurrently. This leads to potential *concurrency problems*, which occur when many transactions try to interact with the same database object (access the same data) at the same time. The nature of the interaction depends on the actions each transaction performs: from simply reading data to inserting, updating, or deleting data.

You need to consider these issues because they're directly related to the isolation property of transactions, which dictates that changes made by a transaction shouldn't be visible to other concurrently running transactions. If a transaction modifies information in a data table, should other transactions be able to access that data table? If yes, in what way? What if the transaction only reads from the data table without modifying it?

The answer to these questions depends on the *transaction isolation level*. You can manually set the isolation level for each transaction, and this establishes the way transactions behave when they're trying to access the same piece of data.

Transactions ask for ownership of a particular piece of data by placing *locks* on it. There are many different types of locks, and they differ in the way they limit access to database resources. For example, a row can be locked in such a way that other transactions can't access it in any way or can read it but not modify it. Also, depending on the lock granularity, the resource they apply to can be an entire database, a data table, a row or a number of rows, and so on.

It's important to understand the difference between locks and the transaction isolation level. Locks limit access to database objects, and the transaction isolation level specifies how the active transaction places locks on the resources with which it works.

SQL-99 specifies four transaction isolation levels. With each level, a different balance is struck between the level of data integrity protection offered and the performance penalties imposed.

The four transaction isolation levels, listed from the one that offers the best performance to the one that offers the best protection, are as follows:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

The SQL-99 standard also categorizes transactions into *read-only* transactions and *read-write* transactions. Read-only transactions are a special kind of transaction, and you'll learn about them in a moment.

NOTE *The isolation levels mentioned previously can only be applied to read-write transactions.*

The SQL-99 command for setting the transaction type is `SET TRANSACTION`. The complete syntax is as follows:

```
SET [LOCAL] TRANSACTION { { READ ONLY | READ WRITE } [,...]
| ISOLATION LEVEL
   { READ COMMITTED
   | READ UNCOMMITTED
   | REPEATABLE READ
   | SERIALIZABLE } [,...]
| DIAGNOSTIC SIZE INT };
```

Therefore, to choose between read-only and read-write transactions, the syntax is as follows:

```
SET TRANSACTION [READ ONLY|READ WRITE]
```

To set a transaction level, you can use a command such as the following. Note that setting the transaction isolation level automatically assumes a read-write transaction:

```
SET TRANSACTION ISOLATION LEVEL <isolation level name>
```

The default transactional mode in all databases covered in this book is `READ COMMITTED`.

Read-Only Transactions

Read-only transactions are so named because they can't contain any data modification statements—only plain `SELECT` statements are allowed.

The default state for read-write transactions is *statement-level consistency*. In other words, if, within the scope of a transaction, the same record is read twice, different values may be retrieved each time if the record was modified between readings.

Read-only transaction mode solves this problem by establishing *transaction-level read consistency*. In a read-only transaction, all queries can only see the changes committed before the transaction began.

To set a transaction as being read-only, you type the following command:

```
SET TRANSACTION READ ONLY;
```

Read-only transactions aren't supported by SQL Server, DB2, or MySQL. They are supported, however, by Oracle.

READ UNCOMMITTED Isolation Level

This is the first and most dangerous isolation level, but it's also the one that offers the best performance. The following command sets the `READ UNCOMMITTED` isolation level for the next transaction:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

The `READ UNCOMMITTED` isolation level isn't supported by Oracle but is supported by SQL Server, DB2, and MySQL.

When the transaction is in `READ UNCOMMITTED` mode, its isolation property isn't enforced in any way, and the transaction is able to read uncommitted changes from other concurrently running transactions (effectively breaking the isolation property of the ACID rules).

With `READ UNCOMMITTED` mode, the transaction is susceptible to *dirty reads* and many other existing kinds of consistency problems.

Data Consistency Problem: Dirty Reads

Dirty reads happen when a transaction reads data that was modified by another transaction but that hasn't yet been committed. If the other transaction rolls back, the first transaction ends up reading values that, theoretically, never existed in the database.

To understand dirty reads, imagine two concurrent transactions that work with the `Student` table. One transaction calculates the total number of students from that table, and the second removes or adds students to the table, as shown in Table 10-1.

Table 10-1. Sequence of Actions That Demonstrate Dirty Reads

Time	Transaction 1	Transaction 2
T1	The transaction starts.	
T2	Removes or adds records from/to the `Student` table.	Transaction starts.
T3		Calculates the total number of students based on the uncommitted results of Transaction 1.
T4	Transaction rolls back.	

If Transaction 2 works with the `READ UNCOMMITTED` isolation level, it can read the uncommitted changes from Transaction 1. In this example, because Transaction 1 finally rolls back, Transaction 2 ends up calculating an erroneous number of students. This is a dirty read.

Another scenario of a dirty read would be if Transaction 1 changed the data of a student (say, the name), then Transaction 2 read that uncommitted data, and finally Transaction 1 rolled back.

NOTE *Oracle doesn't support the* `READ UNCOMMITTED` *isolation level. Oracle always reads the last-committed values, even if the data is being changed by other ongoing transactions.*

You can avoid dirty reads by forbidding data that's being modified by other transactions from being read.

However, there are certain situations in which you might want to allow dirty reads, the most common of which is when you want to get quick reports or statistics from your database, where it isn't critical to get very accurate data. In such situations, setting an isolation mode of `READ UNCOMMITTED` can help you because it locks fewer resources and results in better performance than the other isolation levels.

The default transaction isolation level, `READ COMMITTED`, doesn't permit dirty reads, so you explicitly need to set the isolation level to `READ UNCOMMITTED` in situations where you want to allow dirty reads. The other isolation levels (`REPEATABLE`

READ and SERIALIZABLE) are even more stringent about enforcing data consistency, and they don't permit dirty reads either.

READ COMMITTED Isolation Level

The second isolation level in terms of enforcing transaction isolation is READ COMMITTED, which is the default mode for most database systems.

When in READ COMMITTED mode, all resources modified by the transaction are locked until the transaction is completed—in other words, any updated, inserted, or deleted records won't be visible to other transactions (or will be only visible to their last-committed values), until (and unless) you commit the transaction. Transactions that run in READ UNCOMMITTED mode are the exception to this rule because they can uncommitted changes from other transactions.

Also, other transactions aren't allowed to modify the rows that are being modified by your transaction but are able to modify the rows that are simply read by your transaction.

When in READ COMMITTED mode, no dirty reads will happen in your current transaction because you can't see or modify data that is being updated by other transactions. However, the READ COMMITTED mode doesn't guard against *unrepeatable reads*.

Data Consistency Problem: Unrepeatable Reads

An unrepeatable read happens when you read some data twice in a transaction and you get different values because it has been modified in the meantime by another transaction.

When in READ COMMITTED mode, other transactions are allowed to modify data that has only been accessed for reading by your transaction. So, if you start a transaction, read a record, do some other things, and then read the same record again, you might read a different value.

If this is acceptable and no serious problems can occur because of unrepeatable reads, it's best to stick with the READ COMMITTED isolation mode. Additionally, in some cases, you can avoid unrepeatable reads by saving pieces of information in variables or temporary tables and using the saved data instead of querying the database again.

Before moving on to the next transaction isolation level (which prevents unrepeatable reads), you'll see an example demonstrating unrepeatable reads (see Table 10-2). In this example, Transaction 1 tries to calculate the average mark for all students by summing up all their marks and then dividing by the number of students.

For the purposes of this example, assume the mark of each student is stored in the Student table (so working with marks implies working with Student).

Table 10-2. Sequence of Actions That Demonstrate Unrepeatable Reads

Time	Transaction 1	Transaction 2
T1	The transaction starts. (`READ COMMITTED`)	
T2	Sums up the students' marks.	Transaction starts.
T3	(Waits for Transaction 2 to release locks.)	Deletes one student, locking the `Student` table.
T4	(Waits for Transaction 2 to release locks.)	Transaction commits, releases locks.
T5	Retrieves number of students.	
T6	Calculates average mark by dividing the two numbers calculated earlier.	

Transaction 1 ends up calculating a wrong average mark because the students' data changes while the transaction is running. The students' data can be modified by other transactions because Transaction 1 is only reading it and doesn't place any locks on it.

In fact, after Transaction 2 updates the `Student` table (placing locks on it), Transaction 1 needs to wait until Transaction 2 finishes in order to calculate the number of students.

NOTE *Oracle would behave as before, reading the last-committed values from the* `Student` *table without blocking Transaction 1 until Transaction 2 finishes executing.*

If you don't find any solutions to avoid unrepeatable reads, SQL-99 has an isolation level that does the work for you, guarding against unrepeatable reads and dirty reads: the `REPEATABLE READ` isolation level.

REPEATABLE READ Isolation Level

This transaction isolation mode provides an extra level of concurrency protection by preventing not only dirty reads but also unrepeatable reads.

The way most databases (again, not Oracle) enforce repeatable reads is to place *shared read locks* on rows that are being read by the transaction, not just on the ones that are being updated (as the `READ COMMITTED` isolation mode does). This way, as soon as one record is read, you can guarantee you'll get the same value if you read it again during the same transaction.

Having a shared lock on a record permits other transactions to read the record but not to modify it. In the previous example, setting the first transaction to `REPEATABLE READ` would prevent the second transaction from removing the student, ensuring the calculated average mark is correct. Table 10-3 shows the same transactions running, but this time the first transaction runs in `REPEATABLE READ` mode.

Table 10-3. Sequence of Actions That Explain the `REPEATABLE READ` *Isolation Mode*

Time	Transaction 1	Transaction 2
T1	The transaction starts in `REPEATABLE READ` mode.	
T2	Sums up the students' marks (placing shared locks on the rows in the `Student` table).	Transaction starts.
T3	Retrieves number of students.	Tries to delete one student but finds the table locked.
T4	Calculates average mark by dividing the two numbers calculated earlier.	(Waits.)
T5	Transaction completes executing, releases locks.	(Waits.)
T6		Deletes student.

You get the right answer this time, but because other transactions need to wait for your transaction to finish processing before getting to the requested data, this can result in important performance penalties for your applications. The longer your transaction lasts, the longer the locks will be held, practically not allowing other transactions to perform any modifications on them.

With the `REPEATABLE READ` isolation level, you can be sure unrepeatable reads will be avoided. However, there's still one more concurrency-related problem that can happen: *phantoms*.

Data Consistency Problem: Phantoms

Phantoms are similar to repeatable reads, except they also take into account the case where new records are being introduced to a data table by another transaction while the current transaction is working with the table.

Let's imagine another scenario, similar to the previous example, but this time Transaction 2 inserts a new student rather than removing an existing one. Table 10-4 shows the actions performed by the two transactions.

Table 10-4. Sequence of Actions That Demonstrate Phantoms

Time	Transaction 1	Transaction 2
T1	Transaction starts in `REPEATABLE READ` mode.	
T2	Sums up the students' marks (placing shared locks on the rows in the `Student` table).	Transaction starts.
T3	Tries to count the number of students, but Transaction 2 has a lock on the newly created student.	Inserts one student (placing a lock on the new created record).
T4	(Waits.)	Transaction commits, releases locks.
T5	Calculates the total number of students.	
T6	Calculates average mark by dividing the two numbers calculated earlier.	

Transaction 2 is allowed to insert new records to the `Student` table because it doesn't affect any of the existing rows in `Student` (on which Transaction 1 has set shared locks). Phantoms refer to the situation when other transactions insert new records that meet one of the `WHERE` clauses of any previous statement in the current transaction. The `SERIALIZABLE` isolation level guards your transaction against phantoms, as well as the previously presented concurrency-related problems.

SERIALIZABLE Isolation Level

The `SERIALIZABLE` isolation level guarantees the transactions will run as if they were serialized—with other words, they're guaranteed not to interfere, and they execute as if they were run in sequence.

When you set the transaction isolation level to `SERIALIZABLE`, you're guaranteed that other transactions cannot modify (with `UPDATE`, `INSERT`, or `DELETE`) any data that meets the `WHERE` clause of any statement in your transaction.

The `SERIALIZABLE` transaction isolation level is a dangerous one when it comes to performance. It does provide the highest level of consistency—indeed, transactions works the same as if they were executed one at a time.

However, while increasing consistency, you get much lower concurrency because other transactions are restricted in the actions they can do with database objects already used in other transactions—they need to wait one after the other to get access to shared data.

If Transaction 1 was set to SERIALIZABLE in the example presented in Table 10-4, Transaction 2 couldn't have inserted a new record to the Student table before Transaction 1 finished executing.

RDBMS-Specific Transaction Support

So far we've presented the SQL-99 features regarding a transaction's isolation level. Now you'll take a closer look at how they're supported by SQL Server, Oracle, MySQL, and DB2.

SQL Server

SQL Server supports the four specified isolation levels, but it doesn't support read-only transactions. After setting the transaction isolation level, it remains set as such for that connection, unless explicitly changed. The default isolation level is READ COMMITTED.

Here's the syntax:

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

Oracle

Oracle supports read-only transactions, but it doesn't support the READ UNCOMMITTED and REPEATABLE READ isolation levels. The default isolation level is READ COMMITTED.

With Oracle, SET TRANSACTION affects only the current transaction—not other users, connections or other transactions. Here's its syntax:

```
SET TRANSACTION
{ { READ ONLY | READ WRITE }
  | ISOLATION LEVEL
    { READ COMMITTED
    | SERIALIZABLE } };
```

To change the default transaction isolation level for all the transactions on the current session, you use the ALTER SESSION command. Here's an example:

```
ALTER SESSION SET ISOLATION_LEVEL SERIALIZABLE;
```

MySQL

MySQL supports transaction isolation levels with InnoDB tables. Here's the syntax for setting the transaction isolation level:

```
SET [GLOBAL | SESSION]
TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED | READ COMMITTED |
  REPEATABLE READ | SERIALIZABLE }
```

By default, in MySQL, `SET TRANSACTION` affects only the next (not yet started) transaction.

The `SESSION` optional parameter changes the default transaction level for all the transactions on the current session.

The `GLOBAL` optional parameter changes the default transaction isolation level for all new connections created from that point on.

DB2

DB2 supports transaction isolation levels, but the way that the isolation levels are implemented is quite different in DB2 compared to the other RDBMS implementations. In DB2, you can set the isolation level for a transaction on each statement that you use, for example:

```
UPDATE table name
SET assignment clause
WHERE search condition
WITH isolation level
```

where ***isolation level*** can be one of the following:

- `RR`: `REPEATABLE READ`
- `RS`: Read Stability (similar to `REPEATABLE READ`—locks all rows being read and modified but doesn't completely isolate the application process, leaving it vulnerable to phantoms)
- `CS`: Cursor Stability (similar to `READ COMMITTED`)
- `UR`: Uncommitted Read

The default isolation level of any statement relates to the isolation level of the package containing the statement.

For example, referring back to an example you saw in Chapter 3, "Modifying Data" (where you added some new professors to the university but without titles), you could use the following:

```
UPDATE Professor
SET Name = 'Prof. ' || Name
   WHERE ProfessorID > 6;
   WITH RS
```

This method of applying an isolation level also applies to other SQL statements, for example, `SELECT INTO`, `INSERT INTO`. For more information, please refer to the DB2 documentation.

Playing Concurrency

Let's do a short exercise now and test how the database enforces consistency, depending on how you set the transaction isolation level.

For this you'll need to open two connections to the same database. If you're running DB2, please first uncheck the Automatically Commit SQL Statements checkbox in the Command Center ➤ Options ➤ Execution window of Command Center. This way you'll start multistatement transactions just like with Oracle (which works in automatic-transactions mode).

For the purpose of this exercise you'll use the default isolation level, `READ COMMITTED`, for both transactions. This means that you don't need `SET TRANSACTION` statements to explicitly set the transaction isolation level.

First, you need to start new transactions on the two connections. If you're using Oracle or DB2, no additional statements are required. Use `BEGIN TRANSACTION` for SQL Server or `BEGIN` for MySQL.

Then, add a new record to the `Student` table on the first connection:

```
INSERT INTO Student (StudentID, Name) VALUES (115, 'Cristian');
```

After executing this command, while still in the first connection, test that the row was indeed successfully added:

```
SELECT * FROM Student
```

The results show the newly added student:

```
StudentID   Name

----------- ---------------------------------------
1            John Jones
2            Gary Burton
3            Emily Scarlett
...          ...
12           Isabelle Jonsson
115          Cristian
```

Now, while the first transaction is still active, switch to the second connection and read the `Student` table:

```
SELECT * FROM Student
```

Because the first transaction is in `READ COMMITTED` mode, it doesn't allow other transactions to `READ UNCOMMITTED` changes in order to prevent dirty reads. However, the databases implement this protection differently.

With Oracle and MySQL, the second transaction simply ignores the changes made by the first one (which is still running because it wasn't committed or rolled back yet). The list of students will be returned:

```
StudentID   Name

----------- ---------------------------------------
1            John Jones
2            Gary Burton
3            Emily Scarlett
...          ...
12           Isabelle Jonsson
```

DB2 and SQL Server, on the other hand, have a different approach: They don't allow the second transaction to read the entire `Student` table until the first

one decides to commit or roll back. The SELECT in the second transaction will be blocked until the first transaction (which keeps the Student table locked) finishes.

Note that a SELECT statement in the second transaction that refers strictly to rows that weren't added, modified, or deleted by the first transaction isn't affected in any way by that transaction (for example, it isn't blocked until the first transaction finishes, in SQL Server and DB2). An example of such a statement is as follows:

```
SELECT * FROM Student WHERE StudentID=10
```

Or even:

```
SELECT * FROM Student WHERE StudentID<100
```

Note that with SQL Server and MySQL, on the second connection you can set the transaction isolation level to READ UNCOMMITTED. This way, when you read the Student table, you'll see even the row that was inserted by the first transaction, even though that transaction hasn't been committed:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT StudentID, Name FROM Student WHERE StudentID>1200;
```

Now let's roll back the first transaction to get the database to its original form:

```
ROLLBACK;
```

Transaction Best Practices and Avoiding Deadlocks

Transactions are, in a way, a necessary evil in database programming. They provide the functionality you need to ensure data consistency, but they reduce concurrency. The more consistency a transaction isolation level provides, the less concurrency you have and hence the greater performance penalties for concurrently running transactions.

The higher the isolation level you set, the more the users accessing the database at the same time are affected. For this reason, it's important to always use the lowest possible transaction isolation level:

- The lowest isolation level, `READ UNCOMMITTED`, doesn't provide any concurrency protection, and it should be avoided except for the times when you don't require the data you read to be very accurate.

- The default isolation level, `READ COMMITTED`, is usually the best choice, because it protects you from dirty reads, which are the most common concurrency problem.

- The higher isolation protection levels, `REPEATABLE READ` and `SERIALIZABLE`, can and should usually be avoided. Apart from hurting performance, they also increase the probability of *deadlocks*.

A deadlock is a situation when two or more transactions started processing, locked some resources, and they both end up waiting for each other to release the locks in order to complete execution. When this happens, there can be only one that can win the battle and finish processing. The database server you use will choose one of the transactions (named a *deadlock victim*) and roll it back, so the other one can finish execution.

Deadlocks can rarely be entirely eliminated from a complex database system, but there are certain steps you can make to lower the probability of their happening. Of course, most of the times the rules depend on your particular system, but here are a few general rules to keep in mind:

- Use the lowest possible transaction isolation level.

- Keep the transactions as short as possible.

- Inside transactions, access database objects in the same order.

- Don't keep transactions open while waiting for user input (okay, this is common sense, but it had to be mentioned).

Distributed Transactions and the Two-Phase Commit

It's not unusual these days for companies to have, say, SQL Server, Oracle, and MySQL installations on their servers. This leads to the need of conducting transactions that spread over more databases (say, a transaction that needs to update information on three different database servers).

This situation is a bit problematic because all you've studied so far is about transactions that apply to a single database server only. Distributed transactions are possible through specific protocols (depending on the platform) that use a two-phase commit protocol system. The database originating the distributed transaction is called a *Commit Coordinator*, and it coordinates the transaction.

The first phase in the two phase-commit is when the Commit Coordinator instructs all participating databases to perform the required actions. The participating databases start individual transactions, and when they're ready to commit, they send a "ready to commit" signal to the Commit Coordinator.

After all participating databases send a "ready to commit" signal, the Commit Coordinator instructs all of them to commit the operations, and the distributed transaction is committed. If any of the participating databases report a failure, the Commit Coordinator instructs all the other databases to roll back and cancels the transaction.

Summary

We've covered some many new concepts in this chapter. You learned what transactions are, what the ACID properties are, and how they're enforced by the databases.

You could see that, unfortunately, each database product has its own view about how transactions should be implemented, and you learned about the most important features provided by some of the database systems. You then experimented with a few transaction examples that demonstrated transactions.

The chapter ended by previewing a few advanced topics that enhanced your understanding about how database transactions work. In the next chapter, you'll learn about users and security.