

Database Integrity and Transactions

Chapter Outline

- Introduction, 354
- Two-Minute Chapter, 355
- Procedural Languages, 355
 - Where Should Code Be Located?*, 356
 - User-Defined Functions*, 357
 - Looking Up Data*, 358
- Programming Tools, 359
- Data Triggers, 360
 - Statement versus Row Triggers*, 361
 - Canceling Data Changes in Triggers*, 362
 - Cascading Triggers*, 363
 - INSTEAD OF Triggers*, 364
 - Trigger Summary*, 365
- Transactions, 366
 - A Transaction Example*, 366
 - Starting and Ending Transactions*, 367
 - SAVEPOINT*, 368
- Multiple Users and Concurrent Access, 369
 - Optimistic Locks*, 370
 - Pessimistic Locks: Serialization*, 373
 - Multuser Databases: Concurrent Access and Deadlock*, 373
- ACID Transactions, 375
- Key Generation, 377
- Database Cursors, 378
 - Cursor Basics*, 379
 - Scrollable Cursors*, 380
 - Changing or Deleting Data with Cursors*, 381
 - Cursors with Parameters*, 383
- Merchandise Inventory at Sally's Pet Store, 384
- Summary, 388
- Key Terms, 389
- Review Questions, 389
- Exercises, 390
- Web Site References, 394
- Additional Reading, 394

What You Will Learn in This Chapter

- Why would you need to use procedural code when SQL is so powerful?
- How are SQL commands integrated into more traditional programming structures?
- What capabilities exist in procedural code?
- How are business rules added to the database?
- How does a DBMS handle multiple transaction events?
- How do you prevent problems arising when two processes change the same data?
- What are the primary rules to ensure integrity of transactions?
- How are key values generated?
- How can procedural code track row-by-row through a query?
- What issues arise when maintaining totals in the database?

A Developer's View

Ariel: Well, is the application finished?

Miranda: No. The basic forms and reports are done. But I'm still running into some problems.

Ariel: I guess there is always more to do. What kinds of problems?

Miranda: Well, the numbers are sometimes wrong. It seems to happen when several people are working on the same data at the same time. And the application seems a little slow sometimes. And...

Ariel: Whoa. I get the picture. But these seem like common problems. Does the database system have any tools to help?

Miranda: I think so. I'm going to start by looking at some programming topics and data triggers. Then, I think indexes will help me with performance.

Getting Started

Procedural code (programming) is used to handle transactions and other operations that must be performed in a specific order. Currently, every DBMS has its own proprietary programming language. Although the features are similar, the syntax varies. So you need to learn how to write some fundamental programs in the DBMS you want to use. Procedural code is needed for tasks such as custom functions, transactions that require multiple changes, handling concurrency issues, and generating key values.

Introduction

Why would you need to use procedural code when SQL is so powerful? Business applications often exhibit several common problems. For example, multiple users might try to change the same data at the same time, or multiple changes need to be made together, or you need to generate new ID numbers for a table. These situations must be handled correctly to ensure the integrity of the data. SQL commands are powerful tools, but in many of these situations, you need the ability to execute multiple statements or to choose which command should be run. Database systems have evolved procedural languages to handle these situations.

Although there are diverse methods to implement procedural languages, it is helpful when the language is embedded into the query system. With this approach, all of the code and conditions remain within the database definition and constraints are enforced automatically for all applications. These conditions are often written as data triggers—code that is executed when some data element is modified.

The issues of transactions, concurrent access, and key generation appear in almost every business application. This chapter explains the issues involved and provides the common solutions. Performance is a tricky issue as databases expand into huge datasets. Complex queries across many large tables could take a long time to run. But, transaction-based applications need to process data quickly. Vendors have invested considerable money and time into improving performance.

One common solution is to create indexes on the tables. You need to understand the basic index technologies to make informed choices to improve your application's performance.

Two-Minute Chapter

SQL is powerful but sometimes it is necessary to use traditional procedural programming languages to accomplish tasks. Procedural code executes one operation at a time and includes loops and conditional statements. It is often used to examine one row of data at a time. The large DBMSs integrate procedural statements with SQL commands. Many also support writing code in external languages (such as C and Java) that can submit SQL statements to store and retrieve data.

Writing procedural code requires several steps. (1) Learning the overall functions and syntax of the commands. (2) Understanding where to place the code so that it is executed at the proper time. (3) Testing and debugging. (4) Learning when to use procedural code.

The challenge with Step 1 is that the SQL standard has begun defining procedural elements but most systems still rely on their proprietary commands. The overall structures are similar but the details are different, which are explained in the workbooks. Primary structures include the ability to define Functions, Conditions, and Loops. SQL commands are integrated with programming code by defining parameters (or variables) within the SQL command that hold values assigned from the code.

Step 2 is critical because most systems today are event-driven and code is executed in response to some defined event. Within a database, you typically attach code to common data triggers including data UPDATE, INSERT, and DELETE events. For example, when a row of data in a table is changed, the DBMS can execute your custom code to check various conditions. So you have to first think in terms of when your code should be executed.

Step 3 is important with any development method. DBMSs rarely provide additional support for testing, so it is critical for programmers to break things into small pieces and thoroughly test all of the pieces during development.

In terms of Step 4, several common business situations require the support of procedural code. Support for transactions is the most important: Several operations that must be performed (or failed) together. The classic example is transferring money from one bank account to another. Handling errors, including issues with concurrent access, is another common situation. Some systems (particularly Oracle) also require support for generating key values. Other situations arise when dealing with creating forms and making them more usable.

Procedural Languages

How are SQL commands integrated into more traditional programming structures? A **procedural language** is a traditional programming language such as C or Java, where you specify the sequence of a set of commands. Common SQL commands are not procedural because you tell the DBMS only what you want done, not how to do it. Although SQL commands are powerful, sometimes you need the more precise control of a procedural language. For example, you might want to specify that a group of commands must be executed in a particular order and all must be completed for the transaction to succeed. Or, you might want to execute some commands only if certain external conditions are true. In

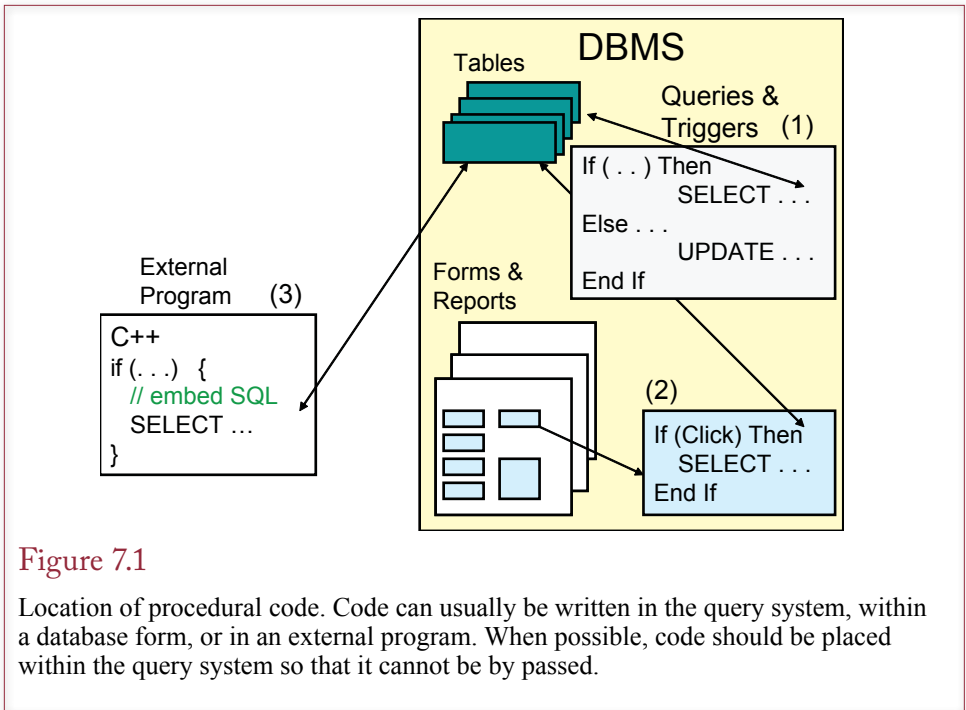


Figure 7.1

Location of procedural code. Code can usually be written in the query system, within a database form, or in an external program. When possible, code should be placed within the query system so that it cannot be bypassed.

more complex cases, you might need to step through each row in a table to perform some difficult computation.

Many varieties of procedural languages exist, but they have elements in common. All of them have variables, conditional statements (if), loops, and subroutines. Each language has its own **syntax**, which includes details such as command and function names, statement terminators, assignment operators, and whether you use parentheses or square brackets for arrays. The syntax is important when you write code, but integrated editors help by prompting for various items and compilers will pinpoint most syntax errors.

This chapter focuses on the logic needed to handle common database operations. The main text is generally language neutral, so you can see how the ideas apply to any database situation. The workbooks provide specific examples using the syntax and structure of individual database systems.

Where Should Code Be Located?

One of the first major questions you face is where the code should be written, stored, and executed. Figure 7.1 shows that procedural code can be placed in three locations: (1) within the DBMS engine as queries or database triggers, (2) within forms and reports, or (3) in external programs. Large, commercial systems, such as Oracle, SQL Server, and DB2 have a procedural language embedded in the DBMS itself. You write code just as you would write any other query and can mix procedural commands with SQL statements. The SQL standard has slowly been adding procedural capabilities. But each vendor supports the concepts using a different syntax.

In general, code that relates directly to the data should be created as a **database trigger** inside the DBMS. Placing the code inside the DBMS means it is written only once and can be called automatically, regardless of how the data is accessed.

The DBMS will ensure that the code is always executed and not bypassed. Think about a security situation where you want to write a note to a log table every time someone changes an employee salary. If you rely on programmers to implement this code in their forms, they might forget to do it or even do it incorrectly. Additionally, someone could create an entirely new form or use a query to change the data directly, without executing the security code. Placing the code within the database provides a mechanism to ensure that it is run anytime the data is changed, regardless of how the modification is generated. In the SQL standard, procedural code stored within the database is called a **persistent stored module (PSM)**, and related procedures and functions can be stored in developer-defined modules. With the release of Office 2010, Microsoft added some rudimentary data macros that can be assigned to tables to handle these types of tasks.

Code within forms should concentrate on handling events or custom problems within the specific form. On the other hand, placing the code into a separate external file is a technique often used in n-tier client/server systems described in Chapter 11. It has the advantage of consolidating the business logic into one location. Separating the business logic from the DBMS makes it easier to replace the DBMS if desired. Database code in external software also arises on Web sites and other situations where data is exchanged with external devices, such as bar code scanners or other sensors.

User-Defined Functions

User-defined functions are a good illustration of procedural code. Occasionally you need a calculation that will be used by several different queries, reports or forms. Even if the computation is relatively simple, placing the code in one location makes it substantially easier to find and change later. You can define your own function name and perform almost any computation you need using procedural code. Figure 7.2 provides an example of a simple function to estimate item costs. In practice, this function would be more complex and include tables and queries, but keeping it simple focuses on the basic elements of a user-defined function.

A function is just a set of code designed to perform a defined task. Typically this function and task need to be called from multiple locations. Functions are passed values and perform computations on these parameters. A value is returned to the calling routine. You can also create procedures, which are different from

Figure 7.2

User-defined function. Placing the business logic in a central location makes it easy to modify later. The function can be used in code segments or SELECT statements.

```
CREATE FUNCTION EstimateCosts
    (ListPrice Currency, ItemCategory VarChar)
RETURNS Currency
BEGIN
    IF (ItemCategory = 'Clothing') THEN
        RETURN ListPrice * 0.5
    ELSE
        RETURN ListPrice * 0.75
    END IF
END
```

```
CREATE FUNCTION IncreaseSalary
    (EmpID INTEGER, Amt CURRENCY)
RETURNS CURRENCY
BEGIN
    IF (Amt > 50000) THEN
        RETURN -1           -- error flag
    END
    UPDATE Employee SET Salary = Salary + Amt
    WHERE EmployeeID = EmpID;
    RETURN Amt;
END
```

Figure 7.3

Function to update the database. The input parameters are used to specify values in the SQL statement. Additional computations can be performed and the parameters modified if needed.

functions in that they do not return a value. However, in almost all cases, you will want to use functions—if only to return error codes. A key feature is that you can include procedural statements such as “if” conditions to handle complex logic.

Figure 7.3 shows a function that uses input parameters to update the database. Almost all functions and procedures use parameters to pass in values to be used in calculations. You can also create local variables to modify the parameters and then use them in the SQL statement. Functions can be as complex as you need. The procedural language system contains the standard elements of any programming language: variables, conditions, loops, and subroutines.

The specific syntax of the module language and parameters depends on the DBMS. The versions shown here reflect the most recent SQL standard, which is only partially supported by DBMS vendors. Although Microsoft Access does not support the CREATE FUNCTION statement, you can build functions in VBA code modules.

Looking Up Data

Procedures and functions often need to be able to use data from tables or queries. Obtaining data from a single row is straightforward with the SELECT INTO statement. It behaves the same as a standard SELECT statement, but instead of displaying the values, it places them into local variables. However, you have to be careful to ensure that the SELECT statement returns only a single row of data. If you make a mistake in the WHERE condition and return multiple rows, it will generate an error.

Figure 7.4 shows how the SELECT INTO statement is used to retrieve a single value. The statement can be used to retrieve data from multiple columns. Just add another COLUMN INTO VARIABLE on the SELECT line and separate it with a comma from the existing line. Notice the difference between the overall objectives in Figures 7.3 and 7.4: The first hard-codes a maximum value (50000), whereas the new approach looks up the maximum raise in a table. This approach is better than using a fixed value because you can create a form that enables an administrator to change this value quickly. If you leave fixed numbers in your program code, a programmer would have to wade through all of the modules to find the magic number. In addition, anytime someone has to change program code,

```
CREATE FUNCTION IncreaseSalary
    (EmpID INTEGER, Amt CURRENCY)
RETURNS CURRENCY
DECLARE
    CURRENCY MaxAmount;
BEGIN
    SELECT MaxRaise INTO MaxAmount
    FROM CompanyLimits
    WHERE LimitName = 'Raise';

    IF (Amt > MaxAmount) THEN
        RETURN -1 -- error flag
    END
    UPDATE Employee SET Salary = Salary + Amt
    WHERE EmployeeID = EmpID;
    RETURN Amt;
END
```

Figure 7.4

Looking up single data elements. The SELECT INTO statement can be used to return data from exactly one row in a table or query. The result is stored in a local variable (MaxAmount) that you can use in subsequent code or SQL statements.

there is a large risk that additional errors will be introduced. Whenever possible, you should place important values into a table and use the lookup process to get the current value when it is needed.

Programming Tools

What capabilities exist in procedural code? Ideally, you already know how to write program code in a separate language such as Basic, C#, Java, or C++. In most situations, you can use these tools to write any level of code you need and then embed database calls within that program. Typically, the database calls consist of SQL statements to insert or retrieve data. However, sometimes you will have to use the database language built into the DBMS. For instance, when you need to examine large amounts of data, it is usually faster to handle the data solely within the DBMS and return simpler results to other programs. Transferring data—even within the same computer—takes time and processing resources. The DBMS is already optimized for handling data internally.

The main concepts you need to know with any procedural language are: (1) Sequence, (2) Variables, (3) Conditions, (4) Loops, (5) Input and Output, and (6) Procedures and functions (subroutines). These are the building blocks or tools that are available to build programs.

One. The primary difference between SQL queries and programming languages is the concept of sequence. A procedural language executes one command line at a time and then moves to the next one. This process controls the order in which commands or steps are executed. In contrast, note that the SQL SELECT command provides minimal control over sequence. Rows are operated on in any order determined by the query optimizer. You can specify the sorting of the final result, but not the order in which rows are operated on. Hence, it is relatively easy to see situations where a procedural language is necessary, such as when two or more

commands need to be executed in a specific sequence. A simple program might consist of two INSERT commands—where data is added to one table and then referenced by the second INSERT command.

Two. Variables are temporary locations in memory to hold data. They usually have a defined data type. Within a DBMS procedural language, the data types available match those used within tables, such as integer, float, and date. When code is written in more traditional languages such as Basic, C#, and Java, the database connector needs to transfer DBMS data types into local variable data types. This process is complicated when the database can hold Null values. External program code often needs special functions to translate data—watching for problems with Null values. One key to understanding variables is to recognize their scope. **Scope** refers to the context or location where a variable is defined. For instance, variables declared within a function only exist within that function. The values are hidden from code written in other functions.

Three. Conditions. The most common form is IF (condition) THEN ... ELSE ... END IF. Sometimes a CASE or ELSE IF block is available to test multiple values in one setting. The purpose is to define multiple code sections so that only one is executed depending on the value of the condition being tested. The action statements within the conditional element are indented to make them easier to read by separating them from the conditional logic.

Four. Loops. Loops define a block of code that is to be executed multiple times. The number of times can be fixed; determined by the amount of data such as the number of rows in a table; or determined dynamically within the loop. In a database environment, the most common use of loops is to define a SELECT query on a table to retrieve a set of rows—then execute the code for each row of data. This approach is used only when SQL cannot handle the problem. SQL is almost always faster at working with sets of rows, but sometimes, procedural code is needed when computations must be performed in a specific order.

Five. Input and Output. Code within the database typically needs to retrieve or store data in tables. SQL statements are used to handle these operations (SELECT, INSERT, UPDATE, and DELETE). The commands can be modified by adding parameters created from variables defined in the code. Code that is written on forms (or reports) can also access data entered onto the form by users.

Six. Procedures and Functions. These subroutines are used to split the code into manageable pieces that are easier to read and to debug. Procedures and functions contain code that can be called from multiple locations—so any code that needs to be used in more than one location should always be written as a function or procedure. But, even if the code is called only one time, it can be useful to write it as a separate function. Smaller functions are easier to debug and they reduce the complexity of the overall program. For example, perhaps you need to write a procedure that performs five different steps. Each step takes 10 lines of code to create. Instead of writing one procedure consisting of 50 lines of code, it is better to write a main procedure that calls five other procedures—each with the 10 lines of code.

Data Triggers

How are business rules added to the database? Data triggers are procedures that are executed when some event arises within the database. The code is written in the query system and is saved as a procedure or function within the database. By binding the code to the database tables, the DBMS ensures the code is always executed when changes are made to the data. The common events that can host

	INSERT	
BEFORE	DELETE	AFTER
	UPDATE	

Figure 7.5

Data triggers. You can set procedures to execute whenever one of these actions occurs. Row events can be triggered before or after the specified event occurs.

triggers are Update, Insert, and Delete, but some systems enable you to attach code to events related to users or the database instance. To understand the role of triggers, consider a procedure that is run whenever someone changes the Salary column in the Employee table. When the data is changed, your trigger procedure is fired to record the person who made the change. With the log, auditors can go back and see who made changes to this critical data. The salary example is a common use of data triggers, which is to add specific security or auditing features to the database. They can also be used to handle business events, such as monitoring when quantity on hand drops below some level and generating an e-mail message or an EDI order to a supplier.

Figure 7.5 lists the basic SQL commands that support triggers. The main data triggers on the rows and columns each have two attributes: BEFORE and AFTER. For example, you can specify a procedure for BEFORE UPDATE and a different procedure for AFTER UPDATE. The BEFORE UPDATE event is triggered when a user attempts to change data, but before the data is actually written to the database. The AFTER UPDATE trigger is fired once the data has been written. You choose the event based on what you want to do with your application. If you need to check data before it is written to the database, you need to use a BEFORE trigger. For instance, you might want to perform a complicated validation test before saving data. On the other hand, if you want to record when data was changed or need to alter a second piece of data, you can use an AFTER trigger.

Statement versus Row Triggers

The SQL standard defines two levels of triggers: (1) triggers may be assigned to the overall table or (2) they may be assigned to fire for each row of data being modified. Figure 7.6 shows the timing of the various triggers for an UPDATE command. Triggers created to the overall table are fired first (BEFORE UPDATE) or at the very end (AFTER UPDATE). Then individual row triggers are fired before or after each row being examined. For row-level triggers, you can also add conditions that examine the row data to decide if the trigger should be fired or ignored. For instance, you might add a row trigger in the Salary case that fires only for employees in a certain division. Note that this condition is completely separate from the original UPDATE WHERE statement. The trigger condition is used only to decide whether or not to fire the trigger.

Figure 7.7 shows a sample trigger that fires whenever a row is changed in the Employee table. Notice that it is a row-level trigger because of the FOR EACH ROW statement. The example also illustrates that triggers can examine and use the data stored in the target table before it is changed (OLD ROW) and after it has been changed (NEW ROW). In this situation, the original salary and new salary are both recorded to the log table. With this information, security managers and auditors can quickly query the log table to identify major changes to salary and

```
UPDATE Employee
SET Salary = Salary + 10000
WHERE EmployeeID=442
OR EmployeeID=558
```

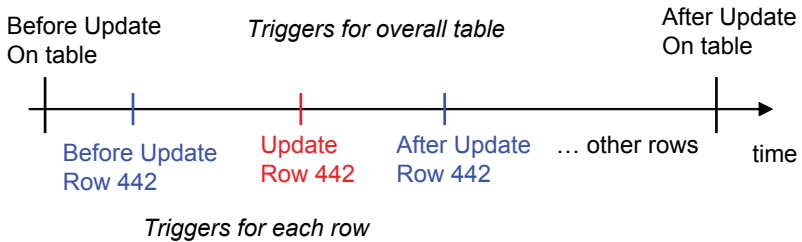


Figure 7.6

Update triggers can be assigned to the overall table and fire once for the entire command, or they can be assigned to fire for each row being updated.

then investigate further to ensure the changes were legitimate. You do have to be careful with the OLD and NEW data. For example, the NEW data has not yet been created in a BEFORE UPDATE trigger, so it cannot be accessed. Also, you cannot alter the OLD data within your trigger code.

Canceling Data Changes in Triggers

One of the uses of triggers is to examine changes in detail before they are written to the database. The BEFORE UPDATE and BEFORE INSERT triggers are often used to validate complex conditions. You also might want to provide more cautious checks before deleting data. In these cases, the structure of the trigger is straightforward. The key element is that you need a way to stop the original SQL statement from executing. The WHEN condition is used to examine the row that is scheduled to be deleted. As shown in Figure 7.8, the SIGNAL statement raises

Figure 7.7

Trigger to log the users who change an employee salary. The trigger fires any time the salary is updated, regardless of the method used to alter the data. It is a useful security tracing technique for sensitive data because it cannot be circumvented, except by the owner of the trigger.

```
CREATE TRIGGER LogSalaryChanges
AFTER UPDATE OF Salary ON Employee
REFERENCING OLD ROW as oldrow
NEW ROW AS newrow
FOR EACH ROW
INSERT INTO SalaryChanges
(EmplID, ChangeDate, User, OldValue, NewValue)
VALUES
(newrow.EmployeeID, CURRENT_TIMESTAMP,
CURRENT_USER, oldrow.Salary, newrow.Salary);
```

```
CREATE TRIGGER TestDeletePresident
BEFORE DELETE ON Employee
REFERENCING OLD ROW AS oldrow
FOR EACH ROW
    WHEN (oldrow.Title = 'President')
        SIGNAL CANNOT_DELETE_PRESIDENT;
```

Figure 7.8

Canceling the underlying SQL command. This trigger examines the data for the employee row being deleted. The company always wants to keep data on any employee with the president title. The WHEN condition evaluates each row. The SIGNAL statement raises an error to prevent the underlying delete from executing.

an error condition that prevents the row from actually being deleted. The actual signal condition (CANNOT_DELETE_PRESIDENT) can be almost anything, but it must be defined as a constant in the overall module. Note that most database system vendors have not yet adopted the SIGNAL keyword, so the actual syntax you need will depend on the system (and version) that you are using. The workbooks give the actual cancel method and syntax needed for each specific DBMS. For instance, Oracle uses the function: Raise_Application_Error, whereas Microsoft SQL Server uses Raiserror.

In general, you should try to avoid using triggers for simple check conditions. Instead, use the standard SQL conditions (e.g., PRIMARY KEY, FOREIGN KEY, and CHECK) because they are more efficient and are less likely to cause additional problems. But sometimes you need to create complex conditions that are difficult to handle with simple conditions.

Cascading Triggers

A serious complication with triggers is that a database can have many triggers on each table. **Cascading triggers** arise when a change that fires a trigger on one table causes a change in a second table, that triggers a change in a third table, and so on. Figure 7.9 shows a common inventory situation. When an item is sold, a new row is added to the SaleItem table that contains the quantity sold. Because the item has been sold, the quantity on hand is updated in the Inventory table. A trigger on the Inventory table then checks to see if the QOH is below the reorder point. If it is, a new order is generated and sent electronically to a supplier, resulting in inserts on the Order and OrderItem tables.

There is nothing inherently wrong with cascading triggers. However, long chains of updates can slow down the system. They also make it difficult to debug the system and find problems. In the example, you might be looking at a problem in the OrderItem table, but it could have been caused by an error in the trigger code for the SaleItem table. The longer the chain, the more challenging it is to identify the source of problems.

A more difficult problem can potentially arise with cascading triggers. What happens when the chain loops on itself? Figure 7.10 shows an example of the problem. A company has embedded several rules about the methods of paying employees. When the salary reaches a certain level, the employee is eligible for bonuses. When the employee has already received substantial bonuses, the bonus amount is limited and the employee is granted additional stock options. If the lev-

Tables	Triggers and Timing
Sale(SaleID, SaleDate, ...) SaleItems(SaleID, ItemID, Quantity, ...)	
	AFTER INSERT ON SaleItems UPDATE Inventory SET QOH = QOH - newrow.Quantity
Inventory(ItemID, QOH, ...)	
	AFTER UPDATE ON Inventory WHEN newrow.QOH < newrow.Reorder INSERT {new Order} INSERT {new OrderItem}
Order(OrderID, OrderDate, ...) OrderItem(OrderID, ItemID, Quantity, ...)	

Figure 7.9

Cascading triggers. With triggers defined on multiple tables, a change in one table (SaleItem) can cascade into changes in other tables. Here, when an item is sold, quantity on hand is updated. If QOH is below the reorder point, a new order is generated and sent.

el of stock options is substantial, the original salary is reduced. But that takes the system back to the beginning, and the salary change could trigger another round of updates. Depending on the computations, this loop could diverge so that the numbers get larger and larger (or increasingly negative), and the computations never end. For this reason, the SQL standard is defined to forbid trigger loops. Systems that follow the standard are supposed to monitor the entire chain of updates, and if it encounters a loop, it should cancel changes and issue a warning. Even if the system is supposed to identify these loops, you should always check the system yourself to make sure that these problems will not arise. Obviously, the system is easier to check if there are only a limited number of triggers. If you can list the triggers in the order shown here, it is fairly easy to see the loop. However, systems rarely provide this option. Instead, you have to look through all of the database triggers and draw your own charts.

INSTEAD OF Triggers

Some database systems support the INSTEAD OF option as an even stronger type of trigger. A standard trigger runs your code in addition to performing the underlying function (DELETE, INSERT, or UPDATE). The INSTEAD OF option completely replaces the underlying command with your code. So, even if the change should be written to the database, you will have to write the additional SQL statements to take the appropriate action. Although this process seems more complicated, it is a useful trick for making queries updateable. Recall that a query that joins multiple tables generally is not updateable; data cannot be added to the query because the system does not always know which table gets the new row. To solve the problem, you can add an INSTEAD OF trigger to the query. Then, changes that are needed can be written to the individual tables with separate SQL statements

	Tables	Triggers and Timing
1	Employee (<u>EID</u> , Salary)	
		AFTER UPDATE IF newrow.Salary > 100000 THEN Add BonusPaid END
2	BonusPaid (<u>EID</u> , <u>BonusDate</u> , Amount)	
		AFTER UPDATE or INSERT IF newrow.Bonus > 50000 THEN Reduce Bonus Add StockOptions END
3	StockOptions (<u>EID</u> , <u>OptionDate</u> , Amount, SalaryAdj)	
		AFTER UPDATE Or INSERT IF newrow.Amount > 100000 THEN Reduce Employee Salary END
4		<i>Return to Step 1</i>

Figure 7.10

Trigger loop. Consider what happens when cascading triggers create a loop, where one trigger returns to alter a table that generated the original change. This loop would set up iterations that might converge or diverge. Even if the loop converges, it will eat up considerable resources.

Trigger Summary

Your first look at database triggers might seem overwhelming. Any table can contain trigger code before and after three different events. You can even write multiple triggers for each event. Do you really need to write database triggers? How do you determine which event to use? The first answer is that you should be conservative in using triggers. Use them to establish critical business rules and monitoring that need to be centralized. Database triggers are convenient and powerful, making it easy to ensure that relatively complex tasks are handled correctly. However, they are difficult to debug and explain to other developers.

The answer to the second question is trickier. You first need to understand the detailed nature of the business rule. Choose the database trigger that provides the most direct application of the rule. For example, if you need a rule related to changing inventory levels, add the trigger to the Items table; not the SaleItems table. When in doubt, write the rule in several locations and test each version. One of the main indicators of success is when your rule fires exactly one time. If the rule does not fire during a test run, it is probably too far away from the desired table. If it fires repeatedly for one business operation, the rule is at too detailed of a level (such as on the SaleItems table instead of the Sale table).

Steps	Savings Balance	Checking Balance
0. Start	5,340.92	1,424.27
1. Subtract 1,000	4,340.92	1,424.27
2. Add 1,000	4,340.92	2,424.27
Problem arises if transaction is not completed		
1. Subtract 1,000	4,340.92	1,424.27
2. Machine crashes		1,000 is gone

Figure 7.11

Transactions involve multiple changes to the database. To transfer money from a savings account to a checking account, the system must subtract money from savings and add it to the checking balance. If the machine crashes after subtracting the money but before adding it to checking, the money will be lost.

Transactions

How does a DBMS handle multiple transaction events? When building applications, it is tempting to believe that components will always work and that problems will never occur. Tempting, but wrong. Even if your code is correct, problems can develop. You might face a power failure, a hardware crash, or perhaps someone accidentally unplugs a cable. You can minimize some of these problems by implementing backup and recovery procedures, storing duplicate data to different drives, and installing an uninterruptible power supply (UPS). Nevertheless, no matter how hard you try, failures happen.

A Transaction Example

An error that occurs at the wrong time can have serious consequences. In particular, many business operations require multiple changes to the database. A **transaction** is defined as a set of changes that must all be made together. Consider the example in Figure 7.11. You are working on a system for a bank. A customer goes to an online banking application and instructs it to transfer \$1,000 from savings to a checking account. This simple transaction requires two steps: (1) subtracting the money from the savings account balance and (2) adding the money to the checking account balance. The code to create this transaction will require two updates to the database. For example, there will be two SQL statements: one UPDATE command to decrease the balance in savings and a second UPDATE command to increase the balance in the checking account.

You have to consider what would happen if a machine crashed in between these two operations. The money has already been subtracted from the savings account, but it will not be added to the checking account. It is lost. You might consider performing the addition to checking first, but then the customer ends up with extra money, and the bank loses. The point is that both changes must be made successfully. The other option is that both operations can fail—leaving the customer and the bank at the starting point. If you have a choice, you want all operations to succeed, but keep in mind that total failure is better than partial success in these cases.

```

CREATE FUNCTION TransferMoney(Amount Currency,
                             AccountFrom Number,AccountTo Number)
    RETURNS NUMBER
curBalance Currency;
BEGIN
    DECLARE HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        Return -2;           -- flag for completion error
    END;
    START TRANSACTION; -- optional
    SELECT CurrentBalance INTO curBalance
    FROM Accounts WHERE (AccountID = AccountFrom);
    IF (curBalance < Amount) THEN
        RETURN -1;       -- flag for insufficient funds
    END IF
    UPDATE Accounts
    SET CurrentBalance = CurrentBalance - Amount
    WHERE AccountID = AccountFrom;
    UPDATE Accounts
    SET CurrentBalance = CurrentBalance + Amount
    WHERE AccountID = AccountTo;
    COMMIT;
    RETURN 0;           -- flag for success
END;

```

Figure 7.12

Transaction to transfer money. If the system crashes before the end of the transactions (Commit), none of the changes are written to the database. On restart, the changes may all be rolled back, or the transaction restarted.

Starting and Ending Transactions

How do you know that both operations are part of the same transaction? It is a business rule—or the definition of a transfer of funds. The real problem is: How does the computer know that both operations must be completed together? As the application developer, you must tell the computer system which operations belong to a transaction. To do that you need to create procedural code and mark the start and the end of all transactions inside your code. When the computer sees the starting mark, it starts writing all the changes to a log file. When it reaches the end mark, it makes the actual changes to the data tables. If something goes wrong before the changes are complete, when the DBMS restarts, it examines the log file and completes any transactions that were incomplete. From a developer's perspective, the nice part is that the DBMS handles the problem automatically. All you have to do is mark the start and the end of the transaction.

Transactions illustrate the need for procedural languages. As shown in Figure 7.12, the multiple UPDATE statements need to be stored in a module function or procedure. In this example, the two UPDATE statements must be completed together or fail together. The START TRANSACTION statement is optional (in the SQL standard) but highlights the beginning of the transaction. If both updates complete successfully, the COMMIT statement executes, which tells the DBMS

```

START TRANSACTION;
SELECT ...
UPDATE ...
SAVEPOINT StartOptional;
UPDATE ...
UPDATE ...
If error THEN
    ROLLBACK TO SAVEPOINT StartOptional;
END IF
COMMIT;

```

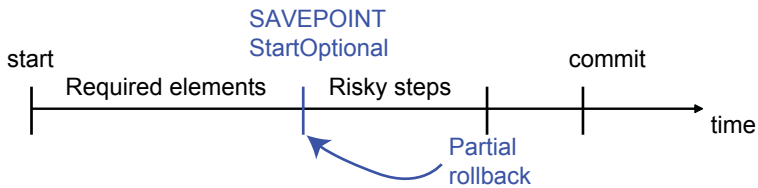


Figure 7.13

SAVEPOINT. A SAVEPOINT enables you to rollback to an intermediate point in the procedure. You can set multiple SAVEPOINTS and choose how far back you want to rollback the changes.

to save all of the changes. If an unexpected error arises, the ROLLBACK statement executes so none of the changes are saved. Most systems handle the transaction requirement by writing all changes to an intermediate log file. If something goes wrong with the transaction, the system can recover the log file and rollback or complete the transaction.

Notice that the START TRANSACTION line comes before the initial SELECT statement. This might seem unnecessary, since it appears that only the UPDATE commands need to be within the transaction. There is a syntax reason for placing this statement first: Any SELECT statement automatically initiates a new transaction. However, as will be explained in the section on concurrency, there is a good reason for starting the transaction before this SELECT statement. Think about things that can go wrong if another process tries to modify the data retrieved by the SELECT statement, before this transaction is finished.

SAVEPOINT

Sometimes, you need intermediate points in a transaction. Some steps are more critical than others. You might have some optional changes that would be useful to save, but if they fail, you still need to ensure that the critical updates are committed. The SAVEPOINT technique divides transaction procedures into multiple pieces. You can roll back a transaction to the beginning, or to a specific SAVEPOINT. Figure 7.13 illustrates the process and shows the syntax to set a SAVEPOINT and rollback to it. As indicated, it can be used to mark a set of risky steps that you would like to include in the update but are not required to use. Consequently, if the updates fail for the risky section, you can discard those changes and still keep the required elements that were defined at the beginning of the transaction. Generally, you could accomplish the same thing by using multiple COMMIT statements, but sometimes the optional code might include a calculation that you want to include in the final result. Without the SAVEPOINT option, you might have to write the final value more than once.

Receive Payment	Balance	Place New Order
1. Read balance 800	800	
2. Subtract Pmt. -200		
		3. Read balance 800
4. Save balance 600	600	
		5. Add order 150
	950	6. Write balance 950

Figure 7.14

Concurrent access. If two processes try to change the same data at the same time, the result will be wrong. In this example the changes made when the payment is received are overwritten when a new order is placed at the same time.

Multiple Users and Concurrent Access

How do you prevent problems arising when two processes change the same data? One of the most important features of a database is the ability to share data with many users or different processes. This concept is crucial in any modern business application: Many people need to use the application at the same time. However, it does create a potential problem with database integrity: What happens when two people try to change the same data at the same time? This situation is known as **concurrent access**. Consider the example of an Internet order system shown in Figure 7.14. The company records basic customer data and tracks charges and receipts from customers. Customers can have an outstanding balance, which is money they currently owe. In the example, Jones owes the company \$800. When Jones makes a payment, a clerk receives the payment and checks for the current balance (\$800). The clerk enters the amount paid (\$200), and the computer subtracts to find the new balance due (\$600). This new value is written to the customer table, replacing the old value. So far, no problem. A similar process occurs if Jones makes a new purchase. As long as these two events take place at different times, there is no problem.

However, what happens if the two transactions do occur together? Consider the following intermingling: (1) The payments clerk receives the payment, and the computer retrieves the current amount owed by Jones (\$800). (2) The clerk enters the \$200 payment. Before the transaction can be completed, Jones places a new order on the Internet for \$150 of new merchandise. (3) The Web server also reads the current balance owed (\$800) and adds the new purchases. Now, before this transaction can be completed, the first one finishes. (4) The payments clerk's computer determines that Jones now owes \$600 and saves the balance due. (5) Finally, the Web server adds the new purchases to the balance due. (6) The order computer saves the new amount due (\$950). Customer Jones is going to be justifiably upset when the next bill is sent. What happened to the \$200 payment? The answer is that it was overwritten (and lost) when the new order change was mixed in with the receipt of the payment.

Receive Payment	Balance	Place New Order
1. Read balance 800	800	
2. Subtract Pmt. -200		
		3. Read balance 800
4. Save balance 600	600	Error: Blocked
		3. Read balance 600
		4. Add order 150
	950	5. Write balance 750

Figure 7.15

Serialization. The first process locks the data so that the second process cannot even read it. Concurrent changes are prevented by forcing each process to wait for the earlier ones to be completed.

Optimistic Locks

Two common methods exist to solve the problem of concurrent changes (optimistic and pessimistic). Today, with fast computer speeds the DBMS can process transactions quickly so there is a lower probability of concurrency problems. An **optimistic lock** begins with the assumption that collisions are rare and unlikely to arise. If they do arise, it is easier to handle the situation at that time. Handling problems is straightforward and takes less DBMS overhead. Particularly in distributed database environments, it is often easier and faster to use optimistic locking.

The key to understanding optimistic locks is to realize that they are not really locks; the DBMS lets your program read any piece of data needed. When your program attempts to change the data, the DBMS rereads the database and compares the currently stored value to the one it gave you earlier. If there is a difference between the two values, it signifies a concurrency problem because someone else changed the data before you were able to finish your task. The DBMS then raises an error and expects your program to deal with it. In summary, optimistic locking can improve performance, but it requires you to deal with potential collisions. Figure 7.15 outlines the basic process. The key to the process lies in modifying the UPDATE command by adding a WHERE clause similar to: WHERE Amount = oldAmount. The “oldAmount” value is the original value stored in a variable when the transaction begins.

The preferred solution to collisions using optimistic locks is to rollback any changes you have already made, and restart your code to read the current value from the database, re-compute your changes, and write the new value to the database. Consider the example of the orders in Figure 7.16. The function first reads the current value of the balance into memory. After completing some other tasks (slow code), it attempts the UPDATE command, with one twist. It specifies that the UPDATE command only applies to the row with the given Account Number **and with the original Amount value**. If the value was changed by a second transaction, this UPDATE command will not alter any rows. The error test following the UPDATE command will recognize if the changes were successful or not. If successful, the routine is done and it exits. If the changes failed, you have

```

CREATE FUNCTION ReceivePayment (
    AccountID NUMBER, Amount Currency) RETURNS NUMBER
BEGIN
    DECLARE HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        RETURN -2;
    END
    SET TRANSACTION SERIALIZABLE, READ WRITE;
    UPDATE Accounts
    SET AccountBalance = AccountBalance - Amount
    WHERE AccountNumber = AccountID;
    COMMIT;
    RETURN 0;
END

```

Figure 7.16

Transaction to transfer money. If the system crashes before the end of the transactions (Commit), none of the changes are written to the database. On restart, the changes may all be rolled back, or the transaction restarted.

complete control over what to do. In this case, it makes sense to go back and pick up the newly revised Amount and try again. To be safe, you should add a counter to the number of retries. If the count reaches too large of a number, this routine should simply give up and produce an error code indicating that it is not possible to update the data at this time.

One catch with the UPDATE command is that you have to be careful with Null values. Recall from queries that a condition of the form Amount = Null will not work correctly. Instead, you have to write Amount IS Null. Consequently, if the original value might be missing, the comparison test is more complicated:

((Amount = oldAmount) OR (Amount IS Null AND oldAmount IS Null))

One of the strengths of the optimistic approach is that it works with any DBMS, even if multiple distributed databases are involved in the transactions. However, it does require that programmers write and validate the proper code for every single update. Consequently, it makes sense to create a code library that contains a generic version of the UPDATE command that can be called for almost any transaction.

The other powerful feature of this approach is that the program code can contain relatively sophisticated analysis to automatically handle common update problems. The other option of a pessimistic lock usually just blocks or delays a transaction which forces users to slow down or solve problems themselves. On the other hand, the optimistic lock realized that it simply had to get the new balance and use it to compute the final amount. No intervention and almost no delay were involved.

Today it is possible to reduce the collisions and concurrent access issues. Focus on using the DBMS to handle all updates. Avoid computing values in code or on forms. Consider Web-based forms which are notoriously slow. The form shows customer account data to a clerk. The clerk enters a value for a payment receipt. If this value is added to the current balance on the form or on the Web server, it runs the risk of a collision when the total is written back to the DBMS. This col-

Process 1	Data A	Data B	Process 2
1. Lock Data A			
	Locked By 1		2. Lock Data B
3. Wait for Data B		Locked By 2	
			4. Wait for Data A

Figure 7.17

Deadlock. Process 1 has locked Data A and is waiting for Data B. Process 2 has locked Data B and is waiting for Data A. To solve the problem, one of the processes has to back down and release its lock.

lision can be avoided by computing the total within the DBMS using the update statement:

```
UPDATE Customer
SET Balance = Balance + NewValue
WHERE CustomerID=@CustomerID;
```

The DBMS simply adds the new value to whatever total currently exists in the table. Your code does not need to test for concurrency issues. Of course, a DBMS running parallel processors (and multithreading) would have to internally monitor concurrency issues when running multiple update commands at the same time. But that work is handled by the DBMS vendor.

The other way to minimize concurrency issues is to avoid storing any totals. Transaction changes are simply written to a table along with time stamps. Totals are computed from this log table whenever they are needed. However, in some

Figure 7.18

Lock manager. A global lock manager tracks all locked resources and associated processes. If it detects a cycle, then a deadlock exists, and the lock manager instructs processes to release locks until the problem is solved.

	Resource A	Resource B	Resource C	Resource D	Resource E
Process 1		Lock		Wait	
Process 2	Wait			Lock	
Process 3			Lock		
Process 4	Lock				Wait
Process 5				Wait	
Process 6		Wait			Lock
Process 7			Wait	Wait	

situations you still want to monitor concurrency. For instance, you do not want two people to buy the last seat on an airplane.

Pessimistic Locks: Serialization

A second solution to the problem of concurrent access is to prevent collisions by forcing transactions to be completely isolated. As shown in Figure 7.17, the **serialization** process forces transactions to run separately so that a second process cannot even read the data being modified by the first process. The first process requests a lock on the balance. Any process that attempts to read that data before the lock is released will receive an error message. A key feature in this approach is the ability of the DBMS to set row-level locks to minimize interference with other processes. Some early systems used table-level locks, so no one could read the data while one balance was being updated!

The method of invoking this type of lock mechanism depends heavily on the DBMS. SQL 99 defined a standard method of specifying the transaction lock, but it has not been widely implemented yet. Figure 7.18 shows the basic logic, but keep in mind that the syntax will be different for each DBMS. The main step is to specify the **isolation level** to SERIALIZABLE in the SET TRANSACTION statement. The DBMS then knows to lock each data element you will be using so that other transactions will be prevented from reading the data until the first changes have been committed. However, it is important that all of the transaction procedures contain error-handling code. Otherwise, when the second transaction (RecordPurchase is almost identical to this one) runs, it will crash and display a cryptic error message when it tries to update or read the data.

The concept of serialization is logical, and it emphasizes the importance of forcing each transaction to complete separately. However, it is based on the technique of a **pessimistic lock**—where each transaction assumes that concurrent interference will always occur. Every time the transaction runs, it places locks on all of the resources that will be needed. This technique slows down the processing and can result in another serious problem described in the following section.

Multiuser Databases: Concurrent Access and Deadlock

Concurrent access is a problem that arises when two processes attempt to alter the same data at the same time. When the two processes intermingle, generally one of the transactions is lost and the data becomes incorrect. For most database operations the DBMS handles the problem automatically. For example, if two users open forms and try to modify the same data, the DBMS will provide appropriate warnings and prevent the second user from making changes until the first one is

Figure 7.19

Optimistic locking process. The steps assume that concurrency problems will not arise. If another transaction does change the data before this transaction finishes, the code receives an error message and must restart.

1. Read the balance.
2. Add the new order value.
3. Write the new balance.
4. **Check for errors.**
5. If errors exist, return to step 1.

```

CREATE FUNCTION ReceivePayment (
    AccountID NUMBER, Amount Currency) RETURNS NUMBER
oldAmount Currency;
testEnd Boolean = FALSE;
BEGIN
    DO UNTIL testEnd = TRUE
    BEGIN
        SELECT Amount INTO oldAmount
        WHERE AccountNumber = AccountID;
        ...
        UPDATE Accounts
        SET AccountBalance = AccountBalance - Amount
        WHERE AccountNumber = AccountID
        AND Amount = oldAmount;
        COMMIT;
        IF SQLCODE = 0 And nrows > 0 THEN
            testEnd = TRUE;
            RETURN 0;
        END IF
        -- keep a counter to avoid infinite loops
    END
END

```

Figure 7.20

Optimistic concurrency with SQL. Keep the starting value within memory and then only do the update if that value is unchanged. If another transaction changed the data before this one completes, go back and get the new value and start over.

finished. Similarly, two SQL operations (e.g., UPDATE) will not be allowed to change the same data at the same time.

Even if you write program code, the DBMS will not allow two processes to change the same data at the same time. However, your code has to understand that sometimes a change to the data will not be allowed. This condition is often handled as an error.

The solution to the concurrency problem is to force changes to each piece of data to occur one at a time. If two processes attempt to make a change, the second one is stopped and must wait until the first process finishes. The catch is that this forced delay can cause a second problem: **deadlock**. **Deadlock** arises when two (or more) processes have placed locks on data and are waiting for the other's data. An example is presented in Figure 7.19. Process 1 has locked data item A. Process 2 has locked item B. Unfortunately, Process 1 is waiting for B to become free, and Process 2 is waiting for A to be released. Unless something changes, it could be a long wait.

Two common solutions exist for the deadlock problem. First, when a process receives a message that it must wait for a resource, the process should wait for a random length of time, try again, release all existing locks, and start over if it still cannot obtain the resource. This method works because of the random wait. Of the two deadlocked processes, one of them will try first, give up, and release all locks with a ROLLBACK statement. The release clears the way for the other process to complete its tasks. This solution is popular because it is relatively easy to program. However, it has the drawback of causing the computer to spend a lot

of time waiting—particularly when there are many active processes, leading to many collisions.

A better solution is for the DBMS to establish a global lock manager as shown in Figure 7.20. A lock manager monitors every lock and request for a lock (wait). If the lock manager detects a potential deadlock, it will tell some of the processes to release their locks, allow the other processes to proceed, and then restart the other processes. It is a more efficient solution, because processes do not spend any time waiting. On the other hand, this solution can be implemented only within the DBMS itself. The lock manager must be able to monitor every process and its locks.

For typical database operations with forms and queries, the DBMS handles concurrent access and deadlock resolution automatically. When you write code to change data, the DBMS still tries to handle the situation automatically. However, the DBMS may rely on you to back out your transaction. Some systems may simply generate an error when the second process attempts to access the data, and it is your responsibility to catch the error and handle the problem.

This section focuses on terms used in computer science and the SQL standards. They are not critical for beginning students.

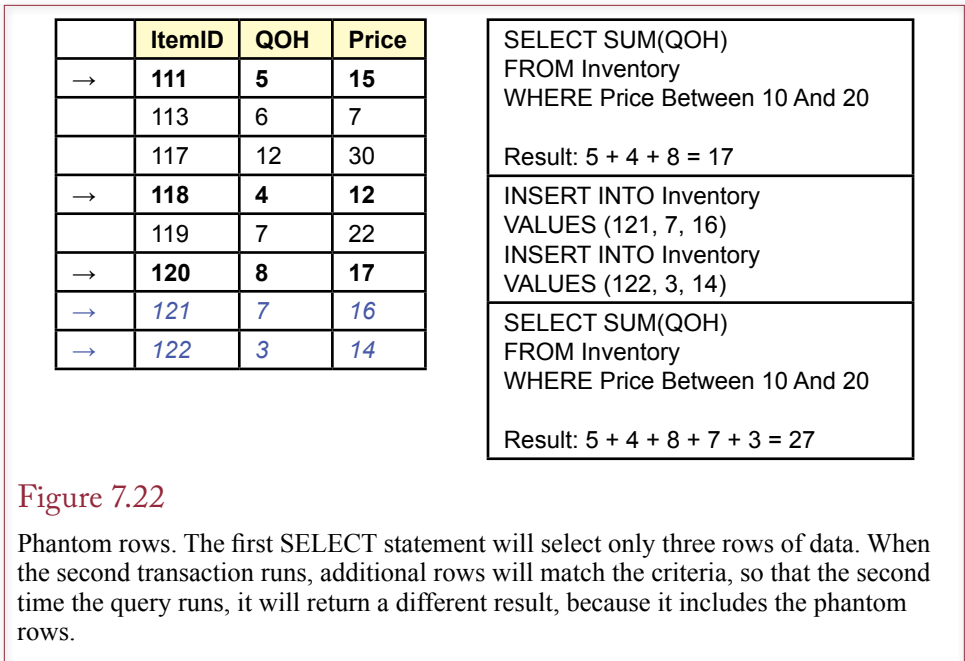
ACID Transactions

What are the primary rules to ensure integrity of transactions? The concept of integrity is fundamental to databases. One of the strengths of the database approach is that the DBMS has tools to handle the common problems. In terms of transactions, many of these concepts can be summarized in the acronym ACID. Figure 7.21 shows the meaning of the term. **Atomicity** represents the central issue that all parts of a transaction must succeed or fail together. **Consistency** means that all data in the database ultimately must be consistent. Even though there might be temporary inconsistencies while a transaction is being processed, in the end, the database must be returned to a consistent state. This status should be able to be tested with application-defined code. For example, referential integrity must be maintained after a transaction is completed. **Isolation** means that concurrent access problems are prevented. Changes by one transaction do not result in errors in other transactions. Note that transactions are rarely completely isolated:

Figure 7.21

ACID transactions. The acronym highlights four of the main integrity features required of transactions.

- **Atomicity:** All changes succeed or fail together.
- **Consistency:** All data remain internally consistent (when committed) and can be validated by application checks.
- **Isolation:** The system gives each transaction the perception that it is running in isolation. There are no concurrent access issues.
- **Durability:** When a transaction is committed, all changes are permanently saved even if there is a hardware or system failure.



they might encounter pessimistic or optimistic locking messages that need to be handled. **Durability** indicates that committed transactions are lasting. Once the transaction commits a change, it stays changed. This concept is critical in the face of hardware and software failures and is more difficult to maintain in a distributed database environment. Most systems ensure durability by writing changes to a log file. Then, even if a hardware failure interrupts an update, the changes will be finished when the system is restarted. Importantly, once the COMMIT statement is accepted, the DBMS cannot rollback the changes.

With SQL 99, the START TRANSACTION and SET TRANSACTION commands can be used to set the isolation level. In increasing isolation order, the four choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. These levels are supposed to be used to prevent different types of concurrency problems, but rarely is there a need for the intermediate levels, so many systems provide only the first and last.

The READ UNCOMMITTED level provides almost no isolation. It enables your routine to read data that another transaction has altered but not yet committed. This problem is sometimes called dirty read because the value you receive might be rolled back and the value ultimately may be inaccurate. If you select this level, SQL will not allow your transaction to update any data, because it might spread a false number throughout the database. The READ COMMITTED level is similar to optimistic concurrency. It will prevent your transaction from reading uncommitted data, but the data might still be changed or deleted by another transaction before the first transaction completes.

The REPEATABLE READ level prevents specific data you are using from being changed or deleted, but does not resolve the problem of phantom data. As shown in Figure 7.22, consider a transaction that computes the sum of quantity on hand if the price of an item falls within a specified range. Now, a second transaction is started before the first one completes. This command inserts rows of data

(or alters the prices). These new rows are phantom rows that are not included in the first query because they did not exist when the query began. After the first two queries have finished, if you repeat the first query, the phantom rows will be committed and you will see new results.

Are phantom rows bad? In many ways, no; they simply arise because a database has constantly changing data. You (and managers) must always remember that the results of a query are accurate only at a specific point in time. On the other hand, if you are writing procedural code, you might be surprised by the results when your queries do not finish in the order you expected—particularly if the DBMS is running on a multiprocessor system. In these situations, you might have to add semaphores or repeat queries to ensue your code follows a specific sequence. Alternatively, you can specify a higher level of isolation.

The SERIALIZABLE isolation level prevents the phantom row problem by ensuring that all transactions behave as if they were run in sequence. However, keep in mind that this result is usually accomplished through the use of locks, so it requires database resources, and it does not guarantee that your transaction will be able to finish on the first try. You still need error handling to catch and resolve the problem when your transaction is blocked by another one.

Key Generation

How are key values generated? As you know by now, the relational database relies heavily on primary keys, which must be unique. It can be difficult in business to guarantee that these keys are always created correctly. Hence, most relational databases have a mechanism to generate numeric keys that are unique. Although these methods work reasonably well for simple projects, you will eventually learn that generated key values present some challenges that must be handled with programming. Also, bear in mind that each DBMS uses a different mechanism to generate keys.

The main problem you encounter with generated keys is when you want to add a row to one table and then insert the matching key value into a second table. For example, when you add a new Customer, the system generates a CustomerID, which you need to insert into the Order table. Figure 7.23 shows the basic problem: the CustomerID key generated to create the new customer must be kept by the transaction procedure so that the key can be inserted into the Order table. The diverse ways of handling the number creation make the problem more difficult.

Logically, generated keys could be created through two primary methods: (1) by an automatic method when a new row is added to a table, and (2) by a separate

Figure 7.23

Generated keys. Creating an order for a new customer requires generating a CustomerID key that is used in the Customer table and must be stored so it can be used in the Order table.

1. Generate key for CustomerID.	Customer Table
2. INSERT row into Customer.	CustomerID, Name, ...
3. Generate key for OrderID.	Order Table
4. INSERT row into Order, using new OrderID and CustomerID.	OrderID, CustomerID, ...

1. INSERT row into Customer.
2. Get the key value that was generated.
3. Verify the key value is correct.
4. INSERT row into Order.

Figure 7.24

Auto-generated keys. The process seems relatively easy when the DBMS automatically generates keys. However, what happens at step 2 if two transactions generate a new key value on the same table at almost the same time?

key generation routine. The advantage of the first method is that the process of adding a row to the initial (Customer) table is relatively simple. The drawback is that it is tricky to make sure you get the correct generated key to use in a second table. The second method solves the second problem, but makes it more difficult to create keys and requires programmers to ensure that the process is followed for every table and insertion operation.

As shown in Figure 7.24, if the DBMS automatically generates key values for each table, the code seems relatively simple. Microsoft Access and SQL Server use this approach. The complication is that problems arise when two transactions generate new key values on the same table at almost the same time. Or, when one transaction triggers inserts into multiple tables. You need to be careful that your code retrieves the correct key value. With some systems, it is difficult to verify the value is correct. You might have to use a `SELECT INTO` statement to retrieve the customer data and double-check the name and phone number.

Because of the difficulties in obtaining an auto-generated key value, the second approach of calling a key generation routine has some benefits. This approach is primarily used by Oracle. Figure 7.25 shows the basic steps needed to create an order for a new customer. Notice that there is no uncertainty about the key value generated. The generation routine ensures that values are unique—even if two transactions request values at the same time. The drawback to this approach is that it is not automatic. However, it is straightforward to write trigger code for the main table (Customer) to generate a new ID for use whenever an `INSERT` is performed on the table.

Database Cursors

How can procedural code track row-by-row through a query? To this point, all of the procedures and functions have dealt with either DML statements or single-row `SELECT` statements. These statements either do not return values or they return only one row of data. This restriction simplifies the program logic and

Figure 7.25

Key-generation routine. The steps are not difficult, but programmers must add them for every table and every routine that inserts data.

1. Generate a key for CustomerID
2. INSERT row into Customer
3. Generate a key for OrderID
4. INSERT row into Order

```
DECLARE cursor1 CURSOR FOR
    SELECT AccountBalance
    FROM Customer;
sumAccount, balance Currency;
SQLSTATE Char(5);
BEGIN
    sumAccount = 0;
    OPEN cursor1;
    WHILE (SQLSTATE = '00000')
    BEGIN
        FETCH cursor1 INTO balance;
        IF (SQLSTATE = '00000') THEN
            sumAccount = sumAccount + balance;
        END IF
    END
    CLOSE cursor1;
    -- display the sumAccount or do a calculation
END
```

Figure 7.26

SQL cursor structure. DECLARE, OPEN, FETCH, and CLOSE are the main statements in the SQL standard.

makes it easier to learn the foundations of SQL procedures. However, some applications will require more sophisticated queries: SELECT statements that return multiple rows of data.

Remember that SQL commands operate on sets of data—multiple rows at one time. What if you want more precise control? Perhaps you need to examine one row at a time to perform a complex calculation, compare some data from an external device, or display the row to the user and get a response. Or perhaps you need to compare one row of data to a second row. For example, you might want to subtract values across two rows. It is difficult to accomplish these tasks with standard SQL commands. As noted in Chapter 9, newer versions of SQL are adding features to perform even these tasks with straight SQL commands. However, you will still find times where you want to track through query results one row at a time.

Cursor Basics

SQL has a process that enables you to track through a set of data one row at a time. You create a **database cursor** that defines a SELECT statement and then points to one row at a time. A loop statement enables you to move the cursor to the next row and repeat your code to examine each row returned by the query. You can also move the cursor back to previous rows, but this process requires more overhead and is rarely needed.

Figure 7.26 shows the basic structure of a procedure to create a cursor and loop through the Customer table to calculate the total amount of money owed. Of course, this particular calculation can be done easier and faster with a simple SELECT statement. The goal here is to show the main structure of the code needed to implement a database cursor. The DECLARE CURSOR statement defines the SELECT statement that retrieves the rows to be examined. Although the example

```

DECLARE cursor2 SCROLL CURSOR FOR
SELECT ...
OPEN cursor2;
FETCH LAST FROM cursor2 INTO ...
Loop...
    FETCH PRIOR FROM cursor2 INTO ...
End loop
CLOSE cursor2;

```

Figure 7.27

FETCH options. A scrollable cursor can move in either direction. This code moves to the last row and then moves backward through the table. Other FETCH options include FIRST, ABSOLUTE, and RELATIVE.

uses only one column, you can use any common SELECT statement including multiple columns, WHERE conditions, and ORDER BY lines. You must OPEN the cursor to use it, and eventually should CLOSE the cursor to free up database resources. When a cursor is first opened, it points to a location immediately before the first row of data. The FETCH statement retrieves one row of data and places the columns of data for that row into program variables. A loop is necessary to track through each row that matches the selection conditions.

Scrollable Cursors

By default, the FETCH command picks up the next row. If the FETCH command pushes the cursor past the end of the dataset, an error condition is created. You can use the WHENEVER statement to catch the specific error, or you can examine the SQLSTATE variable to see if an error was generated with the last SQL statement. A string value of five zeros indicates that the last command was successful.

Several options are available for the FETCH command to move the cursor to a different row. The common options are NEXT, PRIOR, FIRST, and LAST. These retrieve the indicated row. Figure 7.27 outlines the cursor procedure that begins at the last row and moves up to the first row. Note that you must declare the cursor as scrollable with the SCROLL keyword. Of course, it would be more efficient to simply sort the data in reverse order and then move forward; but the objective is

Figure 7.28

Transaction concurrency in cursor code. Your cursor code has tracked down through the data to Carl. It then tries to go back to the prior row with FETCH PRIOR. But, if another transaction has inserted a new row (Bob) in the meantime, your code will pick up that one instead of the original (Alice).

Original Data	Cursor	Modified Data	Insert
<u>Name</u> <u>Sales</u>		<u>Name</u> <u>Sales</u>	
Alice 444,321	1. Read Alice	Alice 444,321	
Carl 254,998	2. Read Carl	Bob 333,229	3. Bob inserted by second process
Donna 652,004		Carl 254,998	
Ed 411,736	4. Move Prior but get Bob instead of Alice	Donna 652,004	
		Ed 411,736	

Year	Sales	Gain
2000	151,039	
2001	179,332	
2002	195,453	
2003	221,883	
2004	223,748	

Figure 7.29

Sales analysis table. A standard SELECT query can compute and save the sales total by year. You now need to write a cursor-based procedure to compute the sales gain from the prior year.

to show that you can move in either direction. Additional FETCH scroll options include the ability to move to the first row (FETCH FIRST) and to jump to a specific row in the dataset. For example, FETCH ABSOLUTE 5 will retrieve the fifth row in the dataset. Since you rarely know the exact row number to retrieve, the relative scroll option is more useful. For instance, FETCH RELATIVE -3 skips back three rows from the current position.

The ability to move backward in the list of rows highlights another transaction concurrency issue. What happens if you work your way down a set of rows and issue the FETCH PRIOR command? Most of the time, you would simply retrieve the row before the current one. But what happens if another transaction inserts a new row immediately before the FETCH PRIOR command is executed? Figure 7.28 shows the problem. Your code has tracked down to Carl, but a second process has inserted Bob into your list. The FETCH PRIOR command will return data for Bob instead of the data for Alice that you expected to see. The SQL standard solution to this problem is to make the dataset insensitive to other changes. You simply add a keyword to the cursor declaration (DECLARE cursor3 INSENSITIVE CURSOR FOR ...). Effectively, the DBMS copies the results of the query into a temporary table that is not affected by other commands. Although this approach will work, it can be an expensive use of database resources. Instead, be sure to ask yourself why you need to move backward. In most cases, you will find that it is unnecessary. For example, if you want to calculate differences by subtracting the value on the current row from the value on the prior row, simply store the “prior” value in memory, then fetch the next row and perform the subtraction. There is no need to move backwards and risk getting the wrong value.

You might notice that there is no procedure to find a row within the retrieved dataset and move the cursor to that row (such as a SEEK command). Although some systems provide this feature, it is rarely needed. Instead, you should create the WHERE condition to only retrieve exactly the rows you want.

Changing or Deleting Data with Cursors

A common situation that a cursor-based application encounters is the need to change or delete the data at the current row. For example, Figure 7.29 shows a table created to hold sales data for analysis. A standard SELECT command with a GROUP BY clause can compute the sales totals by year. You need to write a cursor-based procedure to compute the increase (or decrease) in sales for each year.

```

DECLARE cursor1 CURSOR FOR
SELECT Year, Sales, Gain
FROM SalesTotal
ORDER BY Year
FOR UPDATE OF Gain;
priorSales, curYear, curSales, curGain
BEGIN
    priorSales = 0;
    OPEN cursor1;
    Loop:
        FETCH cursor1 INTO curYear, curSales, curGain
        UPDATE SalesTotal
        SET Gain = Sales – priorSales
        WHERE CURRENT OF cursor1;
        priorSales = curSales;
    Until end of rows
    CLOSE cursor1;
    COMMIT;
END

```

Figure 7.30

Cursor code for update. The FOR UPDATE option in the declaration enables the Gain column to be changed. The WHERE CURRENT OF statement specifies the row pointed to by the cursor.

Figure 7.31

Parameterized cursor query. Your code sets the value of maxPrice through user input or calculation or another query. When this cursor is opened, the value is applied to the SELECT statement and only the matching rows are returned.

```

DECLARE cursor2 CURSOR FOR
SELECT ItemID, Description, Price
FROM Inventory
WHERE Price < :maxPrice;
maxPrice Currency;
BEGIN
    maxPrice = ...    -- from user or other query
    OPEN cursor2;    -- runs query with current value
    Loop:
        -- Do something with the rows retrieved
    Until end of rows
    CLOSE cursor2;
END

```

The catch is that you need to store this computed value back into the table. To do that, you need to specify that the cursor is updateable, and then write an UPDATE statement that stores the calculation in the row currently pointed to by the cursor. Figure 7.30 shows the main code needed to perform the calculations.

Notice that the cursor declaration states that only the Gain column is updateable. This option protects the database slightly. If you make a mistake or someone else modifies your code later, the DBMS will allow only the Gain column to be changed. An attempt to change the Year or Sales column will generate an error. The other important element is the *WHERE CURRENT OF cursor1* statement. This condition states that the row currently fetched, or pointed to by the cursor, is the one to be changed. The UPDATE statement will apply only to this row. An almost identical statement can be used to delete the current row (DELETE FROM SalesTable WHERE CURRENT OF cursor1).

Cursors with Parameters

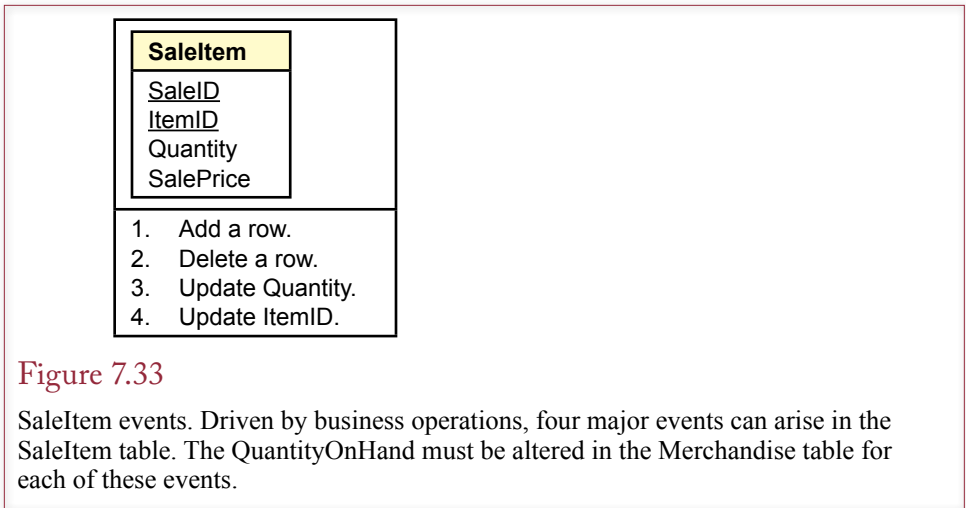
Occasionally, you need a more dynamic query, where you want to pick the specific rows based on some variable within your procedure. For example, a user might enter a price, or your program compute a price based on some other query. Then, you want to retrieve only the rows that are less than the specified price and perform some computation on those rows. You can enter local variables as parameters in the cursor query. Figure 7.31 shows the basic elements of the parameterized cursor. You enter the name of a variable within the cursor's SELECT statement. Within the procedure, you assign a value to this variable. The value might be computed from other variables, input by the user, or even retrieved from a different cursor or query. When the parameterized cursor is opened, the current value is substituted into the query, so that it returns only the rows that match the request. Parameterized queries in the cursor provide powerful tools to dynamically evaluate data automatically in response to other changes.

Be aware that each DBMS uses a different notation to indicate parameters. The standard uses a colon in front of the variable name (:MyVar). SQL Server uses an "at" sign (@MyVar). Oracle does not use any characters in front of the parameter variable, but requires a colon in assignment statements (MyVar := 100). Microsoft Access does not require any special notation. The benefit to marking parameters is that it makes them easier to spot when reading code written by others. When you work in systems without the notation, you might want to adopt a policy of naming parameters and variables to make them easier to recognize (such as v_MyVar).

Figure 7.32

Processing inventory changes. When an item is sold, the quantity sold is entered into the SaleItem table. This value has to be subtracted from the QuantityOnHand in the Merchandise table.

SaleItem Table	Event Code	Merchandise
<u>SaleID</u>	1. Item is sold by adding row to SaleItem.	<u>ItemID</u>
<u>ItemID</u>		Description
Quantity	2. Quantity is subtracted from QuantityOnHand.	QuantityOnHand
SalePrice		ListPrice
		Category



Merchandise Inventory at Sally's Pet Store

What issues arise when maintaining totals in the database? ? To understand the value of procedural code, it helps to look at an example. Handling inventory updates is often a tricky procedure in business database applications. In many situations, employees need to know the quantity on hand for a particular item. An employee may be looking at items to reorder, or a manager might want to know which items are overstocked and have not been selling fast enough. Two basic methods exist to determine the quantity on hand in a database system. First, you could write a procedure that computes the current total on hand whenever it is needed. The routine would add every purchase and subtract every sale of the item to reach the current inventory level. In a large application, this process might be slow. The second approach is to keep a running total of the quantity on hand in the inventory table. This value must then be updated whenever an item is purchased or sold. This second process provides the total very quickly, but faces the drawback of some slightly complicated programming. Keep in mind that both methods also need an adjustment mechanism for “inventory shrink,” to use the accountant’s euphemistic term for inventory items that have disappeared.

Looking at the Merchandise table from Sally’s Pet Store, shown in Figure 7.32, you will notice that it contains a column for QuantityOnHand, so the plan is to use the second inventory approach and keep an updated total for each item. Ultimately, you will need three sets of procedures: One to handle item purchases, one for item sales, and one to adjust for inventory shrinkage identified from physically counting the stock. The adjustment procedure is straightforward, but you have to work on the user interface to make it easy to use. The purchase and sale processes are similar to each other, so the discussion here will examine only the sale of an item.

Whenever something changes in the SaleItem table, the total in the Merchandise table has to be adjusted. Figure 7.33 shows the four basic changes that can arise in the SaleItem table. For instance, when an item is sold, a new row is added to the SaleItem table keyed by the SaleID and ItemID. The row includes the quantity of the item being purchased, such as 10 cans of dog food. This quantity is used to adjust the QuantityOnHand in the Merchandise table. These events might


```
CREATE TRIGGER NewSaleItem
AFTER INSERT ON SaleItem
REFERENCING NEW ROW AS newrow
FOR EACH ROW
    UPDATE Merchandise
    SET QuantityOnHand = QuantityOnHand - newrow.Quantity
    WHERE ItemID = newrow.ItemID;
```

Figure 7.34

New Sale trigger. Inserting a new row triggers the event to subtract the newly entered quantity sold from the quantity on hand.

not be immediately obvious, so consider the following business actions that drive them.

1. A new sale results in adding a row to the SaleItem table, so QuantityOnHand must be decreased by the quantity sold.
2. A clerical error or a customer changing his or her mind could result in the cancellation of a sale or of an item, so a row is removed from the SaleItem table. Any quantity that was already subtracted from the QuantityOnHand must be restored to the total.
3. An item could be returned, or the clerk might change the Quantity because of an error. The quantity adjustment must be applied to the QuantityOnHand total.
4. An item might have been entered incorrectly, so the clerk changes the ItemID. The QuantityOnHand for the original ItemID has to be restored, and the QuantityOnHand for the new ItemID has to be reduced.

You can use database triggers to make the process easier by writing code for each specific event. If you are working with a DBMS without database triggers, the corresponding code has to be written into the forms; this process is similar, but you need to validate each form to make sure it has the necessary code.

The first situation of adding a new row is straightforward. Figure 7.34 shows the logic needed for the database trigger. Only one UPDATE statement is needed: subtract the newly entered Quantity from the QuantityOnHand in the Merchandise table. If you are responsible for reviewing or fixing code in an existing application, you should find that this event is usually handled correctly. The problem is that many developers forget about the other events.

Figure 7.35

Delete Row trigger. This trigger reverses the original subtraction by adding the Quantity back in.

```
CREATE TRIGGER DeleteSaleItem
AFTER DELETE ON SaleItem
REFERENCING OLD ROW AS oldrow
FOR EACH ROW
    UPDATE Merchandise
    SET QuantityOnHand = QuantityOnHand + oldrow.Quantity
    WHERE ItemID = oldrow.ItemID;
```

SaleItem	Clerk	Event Code	Merchandise
<u>SaleID</u> 101 <u>ItemID</u> 15 Quantity 10	1. Enter new sale item, enter Quantity of 10.		<u>ItemID</u> 15 QOH 50
Quantity 8	3. Change Quantity to 8.	2. Subtract Quantity 10 from QOH.	QOH 40
		4. Subtract Quantity 8 from QOH.	QOH 32
Solution that Corrects for Change			
<u>SaleID</u> 101 <u>ItemID</u> 15 Quantity 10	1. Enter new sale item, enter Quantity of 10.		<u>ItemID</u> 15 QOH 50
Quantity 8	3. Change Quantity to 8.	2. Subtract Quantity 10 from QOH.	QOH 40
		4. Add original Quantity 10 back and subtract Quantity 8 from QOH.	QOH 42

Figure 7.36

Errors arise if you do not handle changes in quantity. If Quantity is changed, you must add back the old value and then subtract the new value. The top steps show the error in QOH if you do not handle changes.

The second event of handling deleted rows is no more difficult than the code for inserting a row. Figure 7.35 shows the new trigger that is needed. Deleting a row from SaleItem indicates that the item was not really sold. Consequently, the trigger reverses the effect of the sale by adding the Quantity back to the QuantityOnHand.

As shown in Figure 7.36, the situation for changing data is more complex. You need to think about what it means when the Quantity value is changed. Say that the QuantityOnHand for the specified item begins at 50 units. Then, a SaleItem row was inserted with a Quantity of 10. The insert trigger fired and subtracted those 10 units, leaving the QuantityOnHand at 40 units. The clerk now changes the Quantity from 10 to 8. Since 2 fewer units were sold, the QuantityOnHand needs to be adjusted.

Figure 7.37

Update Quantity trigger. If Quantity is changed, you must add back the old value and then subtract the new value.

```
CREATE TRIGGER UpdateSaleItem
AFTER UPDATE ON SaleItem
REFERENCING OLD ROW AS oldrow
NEW ROW AS newrow
FOR EACH ROW
UPDATE Merchandise
SET QuantityOnHand = QuantityOnHand
+ oldrow.Quantity - newrow.Quantity
WHERE ItemID = oldrow.ItemID;
```

```
CREATE TRIGGER UpdateSaleItem
AFTER UPDATE ON SaleItem
REFERENCING OLD ROW AS oldrow
              NEW ROW AS newrow
FOR EACH ROW
BEGIN
    UPDATE Merchandise
    SET QuantityOnHand = QuantityOnHand + oldRow.Quantity
    WHERE ItemID = oldrow.ItemID;

    UPDATE Merchandise
    SET QuantityOnHand = QuantityOnHand - newRow.Quantity
    WHERE ItemID = newRow.ItemID;
    COMMIT;
END
```

Figure 7.38

Final update trigger. If the ItemID is changed, you must restore the total for the original item and subtract the new quantity from the new ItemID.

As shown in Figure 7.37, the easiest way to understand the adjustment code is to think of it as adding the original 10 units back and then subtracting the new Quantity of 8 units. The net result will leave QuantityOnHand at 42 units. Notice that you need access to the old row value (10). All trigger-based systems have a way to obtain this value. If you have to build the inventory code on a form, it is slightly more complicated to obtain this value; but it can be done.

The fourth change to the code is more difficult to portray. What happens if a clerk changes the ItemID value? Ultimately, you have to restore the QuantityOnHand for the original ItemID, then subtract it for the new ItemID. The first complication is that database triggers might not have separate events for each column being changed. So you have to integrate the changes due to the ItemID into the previous code written to handle Quantity changes. Again, you need to think about the individual steps. Start with a QuantityOnHand of 50 for ItemID 1, then enter a sale of 10 items. The Insert trigger reduces QuantityOnHand to 40 units. Now the clerk changes the ItemID from 1 to 11. That means that no units of ItemID 1 were actually sold, so the 10 units have to be added back to its QuantityOnHand. Additionally, the 10 units have to be subtracted from the QuantityOnHand for ItemID 11. As shown in Figure 7.38, this trigger requires two separate UPDATE statements. Notice that the WHERE clause in the first statement uses the oldrow.ItemID and the second one uses the newRow.ItemID. Also, look more closely at the two SET statements. The first one adds the oldRow.Quantity, the second one subtracts the newRow.Quantity. Why is this difference important? First, it is possible that the clerk changed the Quantity along with the ItemID, and you need to make sure the old Quantity is used for the old ItemID. Second, and more importantly, this trigger also handles the simple change in Quantity, even if the ItemID is not changed. Assume the ItemID is set at 1 and is not changed. Start with a QuantityOnHand of 50 units, and an initial Quantity sold of 10, leaving a current QuantityOnHand of 40 units. Read through the code to see how it works if only

the Quantity is changed from 10 to 8 units. First, the old Quantity (10) is added back to the QuantityOnHand. Second, the new Quantity (8) is subtracted, leaving 42 units on hand. This process is the same as that shown in Figure 7.37, but it is accomplished in two steps instead of one.

The same code must be written for the purchase table (OrderItem) with the same logic. However, for business reasons, you might want to wait to update the QuantityOnHand until the items actually arrive. If you do decide to wait, your primary initial trigger is not on the OrderItem INSERT event, but on the UPDATE event on the MerchandiseOrder table. Have the trigger look for an entry in the ReceiveDate column, and then do the QuantityOnHand updates.

Summary

Although SQL commands are powerful, you sometimes need a procedural language to gain detailed control over updates or to connect to other devices or applications. Depending on the DBMS, procedural code can exist within modules, within forms, or in external applications. Database triggers are an important application of procedural code. These procedures are triggered or executed in response to some database event, such as inserting, updating, or deleting data. Triggers can be used to enforce complex conditions or to execute business rules. For instance, a trigger might be attached to QuantityOnHand within an Inventory table to automatically notify a supplier when the value falls below a certain level. Cascading triggers arise when a change in one table fires a trigger that causes changes in additional tables, that might trigger even more events. Long cascades can be difficult to debug and use substantial server resources.

Transactions are critical applications in most business operations. They represent a collection of changes that must succeed or fail together. Setting start and ending points for transactions is an important step in application development to protect the integrity of the data. Concurrent access where multiple users attempt to modify the same data at the same time is another substantial threat to database integrity. Pessimistic locks have often been used to protect data through serialization so that only one transaction can see data at a time. However, multiple locks eat up resources and can lead to deadlock issues. Optimistic locks assume that collisions are unlikely, but code must be added to handle the situations when they do arise. The ACID acronym (atomicity, consistency, isolation, and durability) is a useful way to remember the main features desired of a DBMS to protect transaction integrity.

Generating keys is an important step in many relational databases, since it is difficult to trust humans to create unique identifiers. Two common methods are used to generate keys: (1) automatically create them when a row is added to a table, or (2) provide a separate function that generates keys on demand. Both methods create complications. The automatically generated keys are difficult to obtain and use in secondary tables. The generation functions require programmers to write code for every table and every insertion procedure.

Database cursors provide a method for procedural code to retrieve multiple rows of data from a query to step through the rows one at a time. The cursor points to one current row that can be examined, modified, or deleted by your code. Scrollable cursors move forward or backward through the rows, but whenever possible, you should try to move only in one direction. With updateable cursors, code can change or delete the data in the current row. With a parameterized query, code can dynamically choose the rows to be retrieved in response to other conditions.

A Developer's View

Miranda learned that even a good DBMS often requires programming to handle some complex issues. In developing your application, you should examine all of the business processes and identify transaction elements. Also, be sure that your UPDATE and DELETE procedures can handle concurrency issues. Remember that a professional application anticipates and handles errors gracefully. Write data triggers or module code to automate basic processes and perform all needed calculations. Write additional cursor-based code if needed to perform advanced calculations.

Key Terms

atomicity	optimistic lock
cascading triggers	persistent stored module (PSM)
concurrent access	pessimistic lock
consistency	procedural language
database cursor	scope
deadlock	serialization
durability	syntax
isolation	transaction
isolation level	trigger

Review Questions

1. Why would you need a procedural language when SQL is available?
2. What is the purpose of data triggers?
- ✓ 3. What is the purpose of form events?
4. What is a transaction and why do they have to be defined by developers?
5. How do you start and finish a transaction?
6. How is pessimistic locking different from optimistic locks?
7. What code do you need to add to handle conflicts with optimistic locks?
- ✓ 8. What is an ACID transaction?
9. What are the most common methods used to generate keys?
10. How do you obtain the most recently generated key in the DBMS you are using?
- ✓ 11. What is a database cursor and why is it important?
12. What is the program logic to using a database cursor to alter data?

Exercises



1. Create a small database with tables for Customers and Employees. In addition to name and phone number, each table should hold a date column for when the person first started (as either a customer or hire date). Write a function that returns a percentage discount that uses a phone number to decide if the buyer is a customer or employee. Customers for less than one year get no discount, 1-3 years (2%), 4-7 years (4%), 8 or more years (5%). Employees for less than one year get no discount, 1-2 years (5%), 3-5 years (7%), 6 or more years (10%).
2. Create a database table of Employees that includes the maximum number of vacation days and number of sick days allowed each year.

```
Employees(EmployeeID, LastName, FirstName, Phone,
VacationDays, SickDays, DateHired, Dateborn)
```

Create a second table with keys for EmployeeID and Year that has values for number of vacation days and sick days taken that year.

```
EmployeeDays(EmployeeID, EYear, NVacation, NSick)
```

Write a function that has input parameters for Year, EmployeeID, number of days off, and whether they should be recorded as sick or vacation days. If the employee exceeds the number of allotted sick days, assign the days as vacation time instead. Excess vacation days do not get counted as sick days.

3. Using the same two tables as the prior exercise (Employees and EmployeeDays), write a database trigger that prevents anyone from entering a value for vacation days taken that exceeds the maximum allowed.
4. Create a table that lists item category and the level of tax on that category. For example, food (0 percent), clothing (3 percent), entertainment (10 percent). Write a function with category and price as parameters. Compute and return the appropriate tax. Normally, you would use an SQL statement for this computation, but if the tax table is provided on a separate system, you might need to write code.
5. Create a data trigger that writes a row in a new table whenever employee salary is changed. Store the date changed, the employee, the old salary and the new value.
6. Create a data trigger that will prevent anyone from increasing an employee salary by more than 75 percent.
7. Create a data trigger (or form code if triggers are not available) that adjusts inventory quantity on hand whenever an item is sold. You need a SaleItem and Item table.
8. A Web site sells custom components for cell phones. The site often offers daily deals which consist of “packages” of related items for a specific phone. Table: PackageItems(PackageID, ItemID, SalePrice) For example, one deal might contain a case, screen protector, and color-matched earphones. Each item is listed separately in the Items table of the database which includes



the Quantity On Hand value. Table: Items(ItemID, Category, Description, ListPrice, QOH) Write a transaction function to safely handle the sale of one package that updates all of the QOH values as part of a transaction.

9. Using the basic Items table that contains a QOH column, create a form that lets users edit the data directly. (Normally, you would use a Sale form but keep it simple for now.) Using default settings, determine what happens if two people change the same data at the same time. Adjust the settings to check for optimistic and pessimistic locking if they are available. Hint: You might want to create two separate forms connected to the same table for testing purposes.
10. Assume you are building a database for a Web-based form where a manager loads and displays all of the employee data for editing. At the end of the session, the changes made to the data are sent back to the database. Write the SQL command to safely update the table using optimistic concurrency. Assume you have an array that holds (a) the original values read from the database and (b) the new/changed values.
11. Create a table for LoanPayments(LoanID, PaymentNo, DateDue, Amount). Write a function that is called whenever a new loan is created, to load the payments table with the scheduled payments and amount due.
12. Given the following table, write a cursor-based procedure to loop through the table and compute the percent change from the prior month and store that value in the current row.

SalesMonth	Sales	PercentChange
01	25,123	
02	24,331	
03	32,992	
04	37,102	
05	42,474	
06	46,551	

13. Using the table in the previous exercise, write a cursor-based procedure to compute the average monthly sales (without using the SQL AVG statement).



Sally's Pet Store

14. Where would you put the code (which Event) in each of the following situations? Note if you are using Access or SQL. You do not have to create the code for this exercise.
 - A. Notify a purchasing manager whenever inventory drops below a specified amount.
 - B. Compute the Sales Tax owed on a Sale.
 - C. Notify a supplier when an order is received.
 - D. Notify adoption groups of the total amount of donations they received for the day.
 - E. Validate a new employee's Taxpayer ID with an online company.




15. Write a function to compute the average purchase cost of an item over the prior year and provide a warning if the ListPrice of the merchandise is lower than that value.
16. Write a function to insert a new Customer and return the generated key value. Inputs to the function include the LastName, FirstName, and Phone number.
17. Create a table to hold totals of merchandise sales by month and a percentage increase in sales from the prior month. Write a (SQL) function to compute the monthly totals and transfer them into the table. Add code to compute the percentage changes.
18. Write the code to increase quantity on hand when an item is purchased—specifically when the receive date is set. Be sure to handle it as a transaction, since quantity on hand can also be affected by sales.
19. The Pet Store is thinking about purchasing scanners to use at checkout. These scanners will pick up the ItemID of each merchandise item scanned. Assume that this data will trigger an event when an item is scanned. Write a function that can be called by this event. This function should create a new sale, and store the data for the items sold. You can emulate the scanner trigger by creating a form with a control to select an ItemID and a button to fire the trigger.



Rolling Thunder Bicycles




20. Create a function to compute the great circle route (shortest) distance between two geographic locations.
21. Where would you put the code (which Event) in each of the following situations? Specify if you are using Access or SQL. You do not have to create the code for this exercise.
 - A. Send an e-mail message to a customer when a bicycle is shipped.
 - B. Send an e-mail message to a supplier to order more components when quantity on hand drops below a preset level.
 - C. Notify a manager when an employee is involved with purchases of more than \$50,000 in a month.
 - D. Notify (e-mail) a manager if the daily sales value of bicycles exceeds a preset level (both high and low) in terms of percentage change from the prior year.
 - E. Notify a purchasing manager of all items that were ordered within the last month but not yet received.
22. Create a table to log changes to Employee salaries (SalaryChange(ChangeID, ChangeDate, EmployeeID, OldSalary, NewSalary, User)). Write trigger code on the Employee table to record any changes to the salary into the log table.
23. Create a function that estimates the time to build a new bicycle. It should use the average number of days for the same model type but adjust the days by the number of orders of all bikes made in the past 14 days.

24. Create a form or a function that lets the finance manager safely record payments to manufacturers.
25. Write a function to update the BalanceDue column in the Customer table while avoiding concurrency issues. The function needs input parameters for CustomerID and ChangeAmount which can be positive or negative.
26. Create a query to compute sales by month for each model type. Create a temporary table to hold that data and to hold the percentage change. Write a program that executes the query, placing the data into the table. Then cursor-based code computes the percentage change in sales. The function should return the new balance value.
-  27. Write a procedure to add an interest charge to customer accounts with a balance due. Make sure to handle concurrency/locking problems.
28. Write a program to automatically generate a new purchase order when quantity on hand falls below a specified level. Add the ReorderPoint column to the Component table and enter sample data.



Corner Med

29. Where would you put the code (which Event) in each of the following situations? Specify if you are using Access or SQL. You do not have to create the code for this exercise.
 - A. Two physicians sign up for vacation on the same days.
 - B. E-mail notices sent to the director physician whenever a patient is diagnosed with a set list of codes/diseases (particularly some contagious diseases).
 - C. A warning message sent to the physician and business manager whenever the AmountCharged for a Visit Procedure is below 50 percent of the base cost.
 - D. An e-mail sent to the business manager whenever the amount paid by the insurance company plus the amount paid by the patient differs from the total amount charged for a visit.
 - E. A warning notice sent to the physician when the Systolic pressure for a visit is greater than 140 and the patient is prescribed a drug from a certain list.
30. Write a function that reduces the amount charged for a procedure for a specific patient (VisitProcedureID) and reduces the patient amount owed/paid.
-  31. Create a table to hold revenue earned per week, using a date format of yyyy-ww. Include a column to hold percentage change from the prior week. Write a query to compute the totals and a routine to compute and store the percentage change.
32. To facilitate loading data from the company's older system, write a function that creates a new patient record given LastName, FirstName, Gender, DateOfBirth as input parameters, and creates a new visit record for that patient for a VisitDate parameter. The function should return the newly generated VisitID.

33. Write a database trigger to record the date, user, and patient name any time a patient row is deleted.
34. Change the tables so that patients can make multiple payments. Include the date, amount of payment, and visit. Write a function to return the total amount paid by a patient for a given VisitID. Briefly explain why this method is better than the current tables.

Web Site References

http://www.sigplan.org/	Association for Computing Machinery—Special Interest Group on Programming Languages (advanced).
http://support.microsoft.com/kb/115986 http://speckyboy.com/2012/05/13/six-common-web-programming-mistakes-and-how-to-avoid-them/	Avoiding common database programming mistakes.

Additional Reading

- Baralis, E. and J. Widom, An Algebraic Approach to Static Analysis of Active Database Rules, *ACM Transactions on Database Systems (TODS)*, 25(3) September 2000, 269-332. [Issues in database triggers and sequencing, but plenty of algebra.]
- Ben-Gan, I., L. Kollar, and D. Sarka, *Inside Microsoft SQL Server 2005: T-SQL Querying*, Microsoft Press: 2006. [Discussion and examples of advanced topics for SQL Server.]
- Gray, Jim and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, San Francisco: Morgan Kaufmann Publishers, 1993. [A classic reference on all aspects of transaction processing.]
- ISO/IEC 14834:1996, *Information Technology—Distributed Transaction Processing—The XA Specification*, 1996. [A discussion of the common method of handling transactions across multiple systems.]
- Sanders, R. and J. Perna, *DB2 Universal Database SQL Developer's Guide*, Burr Ridge, IL: McGraw-Hill, 1999. [Using embedded SQL with IBM's DB2 database.]
- Urman, S., R. Hardman, and M. McLaughlin, *Oracle Database 10g PL/SQL Programming*, Oracle Press: 2005. [One of many references providing an introduction to SQL Server programming.]
- Vossen, G., G. Weikum and J. Gray, *Fundamentals of Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery*, San Mateo, CA: Morgan Kaufmann, 2001. [Detailed programmer's perspective of transaction details.]