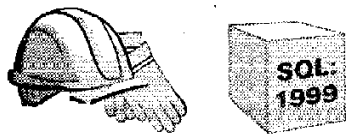


6

DATABASE APPLICATION DEVELOPMENT

- How do application programs connect to a DBMS?
- How can applications manipulate data retrieved from a DBMS?
- How can applications modify data in a DBMS?
- What are cursors?
- What is JDBC and how is it used?
- What is SQLJ and how is it used?
- What are stored procedures?
- **Key concepts:** Embedded SQL, Dynamic SQL, cursors; JDBC, connections, drivers, ResultSets, java.sql, SQLJ; stored procedures, SQL/PSM

He profits most who serves best.

------Ivlotto for Rotary International

In Chapter 5, we looked at a wide range of SQL query constructs, treating SQL as an independent language in its own right. A relational DBMS supports an *interactive SQL* interface, and users can directly enter SQL commands. This simple approach is fine as long as the task at hand can be accomplished entirely with SQL cormnands. In practice, we often encounter situations in which we need the greater flexibility of a general-purpose programming language in addition to the data manipulation facilities provided by SQL. For example, we rnay want to integrate a database application with a nice graphical user interface, or we may want to integrate with other existing applications.

Applications that rely on the DBMS to manage data run as separate processes that connect to the DBlvIS to interact with it. Once a connection is established, SQL commands can be used to insert, delete, and modify data. SQL queries can be used to retrieve desired data. but we need to bridge an important difference in how a database system sees data and how an application program in a language like Java or C sees data: The result of a database query is a set (or multiset) or records, hut Java has no set or multiset data type. This mismatch is resolved through additional SQL constructs that allow applications to obtain a handle on a collection and iterate over the records one at a time.

We introduce Embedded SQL, Dynamic SQL, and cursors in Section 6.1. Embedded SQL allows us to access data using static SQL queries in application code (Section 6.1.1); with Dynamic SQL, we can create the queries at run-time (Section 6.1.3). Cursors bridge the gap between set-valued query answers and programming languages that do not support set-values (Section 6.1.2).

The emergence of Java as a popular application development language, especially for Internet applications, has made accessing a DBMS from Java code a particularly important topic. Section 6.2 covers JDBC, a prograruming interface that allows us to execute SQL queries from a Java program and use the results in the Java program. JDBC provides greater portability than Embedded SQL or Dynamic SQL, and offers the ability to connect to several DBMSs without recompiling the code. Section 6.4 covers SQLJ, which does the same for static SQL queries, but is easier to program in than Java, with JDBC.

Often, it is useful to execute application code at the database server, rather than just retrieve data and execute application logic in a separate process. Section 6.5 covers stored procedures, which enable application logic to be stored and executed at the database server. We conclude the chapter by discussing our B&N case study in Section 6.6.

While writing database applications, we must also keep in mind that typically many application programs run concurrently. The transaction concept, introduced in Chapter 1, is used to encapsulate the effects of an application on the database. An application can select certain transaction properties through SQL cormnands to control the degree to which it is exposed to the changes of other concurrently running applications. We touch on the transaction concept at many points i,n this chapter, and, in particular, cover transaction-related aspects of JDBC. A full discussion of transaction properties and SQL's support for transactions is deferred until Chapter 16.

Examples that appear in this chapter are available online at

http://www.cs.wisc.edu/-dbbook

6.1 ACCESSING DATABASES FROM APPLICATIONS

In this section, we cover how SQL commands can be executed from within a program in a host language such as C or Java. The use of SQL commands within a host language program is called Embedded SQL. Details of Embedded SQL also depend on the host language. Although similar capabilities are supported for a variety of host languages, the syntax sometimes varies.

We first cover the basics of Embedded SQL with static SQL queries in Section 6.1.1. We then introduce cursors in Section 6.1.2. We discuss Dynamic SQL, which allows us to construct SQL queries at runtime (and execute them) in Section 6.1.:3.

6.1.1 Embedded SQL

Conceptually, embedding SQL commands in a host language program is straightforward. SQL statements (i.e., not declarations) can be used wherever a statement in the host language is allowed (with a few restrictions). SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Also, any host language variables used to pass arguments into an SQL command must be declared in SQL. In particular, some special host language variables *must* be declared in SQL (so that, for example, any error conditions arising during SQL execution can be communicated back to the main application program in the host language).

There are, however, two complications to bear in mind. First, the data types recognized by SQL may not be recognized by the host language and vice versa. This mismatch is typically addressed by casting data values appropriately before passing them to or frorn SQL commands. (SQL, like other programming languages, provides an operator to cast values of aIle type into values of another type.) The second complication has to do with SQL being set-oriented, and is addressed using cursors (see Section 6.1.2. Commands operate on and produce tables, which are sets

In our discussion of Embedded SQL, we assume that the host language is C for concreteness, because minor differcnces exist in how SQL statements are embedded in differcnt host languages.

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host-language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC

SQL END DECLARE SECTION. The declarations are similar to how they would look in a C program and, as usual in C. are separated by semicolons. For example. we can declare variables *c_sname*, *c_sid*, *c_rating*, and *c_age* (with the initial *c* used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

The first question that arises is which SQL types correspond to the various C types, since we have just declared a collection of C variables whose values are intended to be read (and possibly set) in an SQL run-time environment when an SQL statement that refers to them is executed. The SQL-92 standard defines such a correspondence between the host language types and SQL types for a number of host languages. In our example, *c_snamc* has the type CHARACTER(20) when referred to in an SQL statement, *c_sid* has the type INTEGER, *c_rating* has the type SMALLINT, and *c_age* has the type REAL.

We also need some way for SQL to report what went wrong if an error condition arises when executing an SQL statement. The SQL-92 standard recognizes two special variables for reporting errors, SQLCODE and SQLSTATE. SQLCODE is the older of the two and is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes. SQLSTATE, introduced in the SQL-92 standard for the first time, associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported. One of these two variables *must* be declared. The appropriate C type for SQLCODE is long and the appropriate C type for SQLSTATE is char[6J , that is, a character string five characters long. (Recall the null-terminator in C strings.) In this chapter, we assume that SQLSTATE is declared.

Embedding SQL Statements

All SQL staternents embedded within a host program must be clearly marked, with the details dependent on the host language; in C, SQL statements must be prefixed by EXEC SQL. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

As a simple example, the following Embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the Sailors relation:

```
EXEC SQL
INSERT INTO Sailors VALUES (:c_sname, :c_sid, :c_rating, :c_age);
```

Observe that a semicolon terminates the command, as per the convention for terminating statements in C.

The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task:

```
EXEC SQL WHENEVER [SQLERROR | NOT FOUND] [ CONTINUE | GOTO st'mt ]
```

The intent is that the value of SQLSTATE should be checked after each Embedded SQL statement is executed. If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to *stmt*, which is presumably responsible for error and exception handling. Control is also transferred to *stmt* if NOT FOUND is specified and the value of SQLSTATE is 02000, which denotes NO DATA.

6.1.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an *impedance mismatch* occurs because SQL operates on *sets* of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation.

This mechanism is called a cursor. We can declare a cursor on any relation or on any SQL query (because every query returns a set of rows). Once a cursor is declared, we can open it (which positions the cursor just before the first row); fetch the next row; move the cursor (to the next row, to the row after the next *n*, to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage

Cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement:

- \Ve usually need to open a cursor if the embedded statement is a SELECT (i.e.) a query). However, we can avoid opening a cursor if the answer contains a single row, as we see shortly.
- INSERT, DELETE, and UPDATE staternents typically require no cursor, although some variants of DELETE and UPDATE use a cursor.

As an example, we can find the name and age of a sailor, specified by assigning a value to the host variable *c_sid*, declared earlier, as follows:

```
EXEC SQL SELECT S.sname, S.age
         INTO    :c_sname, :c_age
         FROM    Sailors S
         WHERE   S.sid = :c_sid;
```

The INTO clause allows us to assign the columns of the single answer row to the host variables *c_sname* and *c_age*. Therefore, we do not need a cursor to embed this query in a host language program. But what about the following query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable *c_minrating*?

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating
```

This query returns a collection of rows, not just one row. 'When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the cOlnmand with EXEC SQL, how can the answers be bound to host language variables? The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating;
```

This code can be included in a C program, and once it is executed, the cursor *sinfo* is defined. Subsequently, we can open the cursor:

```
OPEN sinfo;
```

The value of *c_minrating* in the SQL query associated with the cursor is the value of this variable when we open the cursor. (The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.)

A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row. We can use the FETCH command to read the first row of cursor *sinfo* into host language variables:

```
FETCH sinfo INTO :c_sname, :c_age;
```

When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement (say, in a while-loop in the C program), we can read all the rows computed by the query, one row at a time. Additional parameters to the FETCH command allow us to position a cursor in very flexible ways, but we do not discuss them.

How do we know when we have looked at all the rows associated with the cursor? By looking at the special variables SQLCODE or SQLSTATE, of course. SQLSTATE, for example, is set to the value 02000, which denotes NO DATA, to indicate that there are no more rows if the FETCH statement positions the cursor after the last row.

When we are done with a cursor, we can close it:

```
CLOSE sinfo;
```

It can be opened again if needed, and the value of : *c_minrating* in the SQL query associated with the cursor would be the value of the host variable *c_minrating* at that time.

Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR
        [WITH HOLD]
        FOR some query
        [ ORDER BY order-item-list ]
        [ FOR READ ONLY | FOR UPDATE ]
```

A cursor can be declared to be a read-only cursor (FOR READ ONLY) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (FOR UPDATE). If it is IIpdatable, simple variants of the UPDATE and

DELETE commands allow us to update or delete the row on which the cursor is positioned. For example, if *sinfa* is an updatable cursor and open, we can execute the following statement:

```
UPDATE  Sailors S
SET     S.rating = S.rating - 1
WHERE   CURRENT of sinfo;
```

This Embedded SQL statement modifies the *rating* value of the row currently pointed to by cursor *sinfa;* similarly, we can delete this row by executing the next statement:

```
DELETE  Sailors S
WHERE   CURRENT of sinfo;
```

A cursor is updatable by default unless it is a scrollable or insensitive cursor (see below), in which case it is read-only by default.

If the keyword SCROLL is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed.

If the keyword INSENSITIVE is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. Otherwise, and by default, other actions of some transaction could modify these rows, creating unpredictable behavior. For example, while we are fetching rows using the *sinfa* cursor, we might modify *rating* values in Sailor rows by concurrently executing the command:

```
UPDATE  Sailors S
SET     S.rating = S.rating -
```

Consider a Sailor row such that (1) it has not yet been fetched, and (2) its original *rating* value would have met the condition in the WHERE clause of the query associated with *sinfa,* but the new *rating* value does not. Do we fetch such a Sailor row? If INSENSITIVE is specified, the behavior is as if all answers were computed and stored when *sinfo* was opened; thus, the update command has no effect on the rows fetched by *sinfa* if it is executed after *sinfo* is opened. If INSENSITIVE is not specified, the behavior is implementation dependent in this situation.

A holdable cursor is specified using the WITH HOLD clause, and is not closed when the transaction is conunitted. The motivation for this cornes from long

transactions in which we access (and possibly change) a large number of rows of a table. If the transaction is aborted for any reason, the system potentially has to redo a lot of work when the transaction is restarted. Even if the transaction is not aborted, its locks are held for a long time and reduce the concurrency of the system. The alternative is to break the transaction into several smaller transactions, but remembering our position in the table between transactions (and other similar details) is complicated and error-prone. Allowing the application program to commit the transaction it initiated, while retaining its handle on the active table (i.e., the cursor) solves this problem: The application can commit its transaction and start a new transaction and thereby save the changes it has made thus far.

Finally, in what order do FETCH commands retrieve rows? In general this order is unspecified, but the optional ORDER BY clause can be used to specify a sort order. Note that columns mentioned in the ORDER BY clause cannot be updated through the cursor!

The order-item-list is a list of order-items; an order-item is a column name, optionally followed by one of the keywords ASC or DESC. Every column mentioned in the ORDER BY clause must also appear in the select-list of the query associated with the cursor; otherwise it is not clear what columns we should sort on. The keywords ASC or DESC that follow a column control whether the result should be sorted-with respect to that column-in ascending or descending order; the default is ASC. This clause is applied as the last step in evaluating the query.

Consider the query discussed in Section 5.5.1, and the answer shown in Figure 5.13. Suppose that a cursor is opened on this query, with the clause:

```
ORDER BY minage ASC, rating DESC
```

The answer is sorted first in ascending order by *minage,* and if several rows have the same *minage* value, these rows are sorted further in descending order by *rating.* The cursor would fetch the rows in the order shown in Figure 6.1.

rating	*minage*
8	25.5
3	25.5
7	35.0

Figure 6.1 Order in which Tuples Are Fetched

6.1.3 Dynamic SQL

Consider an application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS. Such an application must accept commands from a user and, based on what the user needs, generate appropriate SQL statements to retrieve the necessary data. In such situations, we may not be able to predict in advance just what SQL statements need to be executed, even though there is (presumably) some algorithm by which the application can construct the necessary SQL statements once a user's command is issued.

SQL provides some facilities to deal with such situations; these are referred to as Dynamic SQL. We illustrate the two main commands, PREPARE and EXECUTE, through a simple example:

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

The first statement declares the C variable *c_sqlstring* and initializes its value to the string representation of an SQL command. The second statement results in this string being parsed and compiled as an SQL command, with the resulting executable bound to the SQL variable *readytogo*. (Since *readytogo* is an SQL variable, just like a cursor name, it is not prefixed by a colon.) The third statement executes the command.

Many situations require the use of Dynamic SQL. However, note that the preparation of a Dynamic SQL command occurs at run-time and is run-time overhead. Interactive and Embedded SQL commands can be prepared once at compile-time and then re-executecl as often as desired. Consequently you should limit the use of Dynamic SQL to situations in which it is essential.

There are many more things to know about Dynamic SQL---how we can pass parameters from the host language program to the SQL statement being pre-parcel, for example--but we do not discuss it further.

6.2 AN INTRODUCTION TO JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. As described in Section 6.1.1, a DBMS-specific preprocessor transforms the Embedded SQL statements into function calls in the host language. The details of this translation vary across DBMSs, and therefore even though the source code can be cOlnpiled to work with different DBMSs, the final executable works only with one specific DBMS.

```
while (true) {
    #sql { FETCH :books INTO :title, :price, };
    if (books.endFetch()) {
        break:
    }

    // process the book
}
```

6.5 STORED PROCEDURES

It is often important to execute some parts of the application logic directly in the process space of the database system. Running application logic directly at the database has the advantage that the amount of data that is transferred between the database server and the client issuing the SQL statement can be minimized, while at the same time utilizing the full power of the database server.

When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application. If we use a cursor to remotely access the results of an SQL statement, the DBMS has resources such as locks and memory tied up while the application is processing the records retrieved through the cursor. In contrast, a stored procedure is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server. The results can be packaged into one big result and returned to the application, or the application logic can be performed directly at the server, without having to transmit the results to the client at all.

Stored procedures are also beneficial for software engineering rea,sons. Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easy. In addition, application programmers do not need to know the the database schema if we encapsulate all database access into stored procedures.

Although they,are called stored *procedur'es,* they do not have to be procedures in a programming language sense; they can be functions.

6.5.1 Creating a Simple Stored Procedure

Let us look at the example stored procedure written in SQL shown in Figure 6.8. We see that stored procedures must have a name; this stored procedure

has the name 'ShowNumberOfOrders.' Otherwise, it just contains an SQL statement that is precompiled and stored at the server.

```
CREATE PROCEDURE ShowNumberOfOrders
SELECT C.cid, C.cname, COUNT(*)
       FROM    Customers C, Orders a
       WHERE   C.cid = O.cid
       GROUP BY C.cid, C.cname
```

Figure 6.8 A Stored Procedure in SQL

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT. IN parameters are arguments to' the stored procedure. OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process. INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values. Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR.

Let us look at an example of a stored procedure with arguments. The stored procedure shown in Figure 6.9 has two arguments: book_isbn and addedQty. It updates the available number of copies of a book with the quantity from a new shipment.

```
CREATE PROCEDURE AddInventory (
                 IN book_isbn CHAR(lO),
                 IN addedQty INTEGER)
UPDATE Books
       SET     qty_in_stock = qtyjn_stock + addedQty
       WHERE   bookjsbn = isbn
```

Figure 6.9 A Stored Procedure with Arguments

Stored procedures do not have to be written in SQL; they can be written in any host language. As an example, the stored procedure shown in Figure 6.10 is a Java function that is dynamically executed by the database server whenever it is called by the dient:

6.5.2 Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

```
CREATE PROCEDURE RallkCustomers(IN number INTEGER)
LANGUAGE Java
EXTERNAL NAME 'file:///c:/storedProcedures/rank.jar'
```

Figure 6.10 A Stored Procedure in Java

```
CALL storedProcedureName(argumentl, argument2, ... , argumentN);
```

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure AddInventory would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char isbn[lO];
long qty;
EXEC SQL END DECLARE SECTION

// set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatment class. CallableStatement is a subclass of PreparedStatement and provides the same functionality. A stored procedure could contain multiple SQL staternents or a series of SQL statements-thus, the result could be many different ResultSet objects. We illustrate the case when the stored procedure result is a single ResultSet.

```
CallableStatement cstmt=
                con.prepareCall("{call ShowNumberOfOrders}");
ResultSet rs = cstmt.executeQueryO
while (rs.next())
```

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
// create the cursor class
#sql !terator CustomerInfo(int cid, String cname, int count);

// create the cursor
```

```
CustomerInfo customerinfo;

// call the stored procedure
#sql customerinfo = {CALL ShowNumberOfOrders};
while (customerinfo.nextO) {
    System.out.println(customerinfo.cid() + "," +
                       customerinfo.count());
}
```

6.5.3 SQLIPSM

All major database systems provide ways for users to write stored procedures in a simple, general purpose language closely aligned with SQL. In this section, we briefly discuss the SQL/PSM standard, which is representative of most vendor-specific languages. In PSM, we define modules, which are collections of stored procedures, temporary relations, and other declarations.

In SQL/PSM, we declare a stored procedure as follows:

```
CREATE PROCEDURE name (parameter1,..., parameterN)
    local variable declarations
    procedure code;
```

We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameterl,..., parameterN)
                RETURNS sqIDataType
    local variable declarations
    function code;
```

Each parameter is a triple consisting of the mode (IN, OUT, or INOUT as discussed in the previous section), the parameter name, and the SQL datatype of the parameter. We can seen very simple SQL/PSM procedures in Section 6.5.1. In this case, the local variable declarations were empty, and the procedure code consisted of an SQL query.

We start out with an example of a SQL/PSM function that illustrates the main SQL/PSM constructs. The function takes as input a customer identified by her *cid* and a year. The function returns the rating of the customer, which is defined as follows: Customers who have bought more than ten books during the year are rated 'two'; customer who have purchased between 5 and 10 books are rated 'one', otherwise the customer is rated 'zero'. The following SQL/PSM code computes the rating for a given customer and year.

```
CREATE PROCEDURE RateCustomer
```

```
                    (IN custId INTEGER, IN year INTEGER)
                    RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numOrders INTEGER;
SET numOrders =
        (SELECT COUNT(*) FROM Orders 0 WHERE O.tid = custId);
IF (numOrders> 10) THEN rating=2;
ELSEIF (numOrders>5) THEN rating=1;
ELSE rating=O;
END IF;
RETURN rating;
```

Let us use this example to give a short overview of some SQL/PSM constructs:

- We can declare local variables using the DECLARE statement. In our example, we declare two local variables: 'rating', and 'numOrders'.
- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.
- SQL/PSM has branches and loops. Branches have the following form:

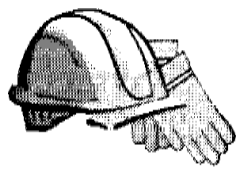
  ```
  IF (condition) THEN statements;
  ELSEIF statements;

  ELSEIF statements;
  ELSE statements; END IF
  ```

  Loops are of the form

  ```
  LOOP
          staternents:
  END LOOP
  ```

- Queries can be used as part of expressions in branches; queries that return a single value can be assigned to variables as in our example above.
- 'We can use the same cursor statements as in Embedded SQL (OPEN, FETCH, CLOSE), but we do not need the EXEC SQL constructs, and variables do not have to be prefixed by a colon ':'.

We only gave a very short overview of SQL/PSM; the references at the end of the chapter provide more information.



20

PHYSICAL DATABASE DESIGN AND TUNING

- What is physical database design?
- What is a query workload?
- How do we choose indexes? What tools are available?
- What is co-clustering and how is it used?
- What are the choices in tuning a database?
- How do we tune queries and view?
- What is the impact of concurrency on perforrnance?
- How can we reduce lock contention and hotspots?
- What are popular database benchnlarks and how are they used?
- Key concepts: Physical database design, database tuning, workload, co-clustering, index tuning, tuning wizard, index configuration, hot spot, lock contention, database benchmark, transactions per second

Advice to a client who cornplained al)out rain leaking through the roof onto the dining table: "Move the table."

—Architect Frank Lloyd Wright

The perfonnance of a DBMS on cornrnonly asked queries and typical update operations is the ultirnate Ineasure of a database desigIl. A I}BA can irnprove perforrnance by identifying perforrnance bottlenecks and adjusting sorne DBIVlS pararneters (e.g., the size of the buffer pool or the frequency of checkpointing) or adding hardware to elirninate such bottlenecks. rIhe first step in achieving

good perforlnancc, however, is to Inake good database design choices, which is the focus of this chapter.

After we design the *conceptual* and *external* schernas, that is, create a collection of relations and views along 'with a set of integrity constraints, we .Il1ust address pe1'forlna11c8 goals through physical database design, in which we design the *physical* sche111a. As user requirernents evolve, it is usually necessary to tune, or adjust, all aspects of a database design for good perforrnance.

This chapter is organized as follows. We give an overview of physical database design and tuning in Section 20.1. The 1nost irnportant physical design decisions concern the choice of indexes. We present guidelines for deciding which indexes to create in Section 20.2. These guidelines are illustrated through several exalnples and developed further in Sections 20.3. In Section 20.4, we look closely at the irnportant issue of clustering; we discuss how to choose clustered indexes and whether to store tuples fron1 different relations near each other (an option supported by sorne DBMSs). In Section 20.5, we emphasize how well-chosen indexes can enable some queries to be answered without ever looking at the actual data records. Section 20.6 discusses tools that can help the DBA to autornatically select indexes.

In Section 20.7, we survey the lnain issues of database tuning. In addition to tuning indexes, we 111ay have to tune the conceptual schema as well as frequently used query and view definitions. We discuss how to refine the conceptual schelna in Section 20.8 and how to refine queries and view definitions in Section 20.9. We briefly discuss the perforrnance irnpact of concurrent access in Section 20.10. We illustrate tuning on our Internet shop exarnple in Section 20.11. We conclude the chapter with a short discussion of DBMS benchrnarks in Section 20.12; benchrnarks help evaluate the perfOI'lnanCe of alternative DBl\1S products.

20.1 INTRODUCTION TO PHYSICAL DATABASE DESIGN

Like all other aspects of database design, physical design rnust be guided by the nature of the data a,nd its intended use. In particular, it is irnportant tonnderstand the typical workload that the database lI1Ust support; the workload consists of a rnix of queries and updates. Users also have certain requirements about how fast certain queries or updates 11111st run or how rnan.y tran.sactions rnust be processed per second. The \vorkload description and users' performance reqllirernents are the basis on \vhich a nurnber of decisions have to be rnade during pl1ysical database design.

Identifying Performance Bottlenecks: All commercial systems provide a suite of tools for monitoring a wide range of system parameters. These tools, used properly, can help identify performance bottlenecks and suggest aspects of the database design and application code that need to be tuned for performance. For example, we can ask the DBMS to monitor the execution of the database for a certain period of time and report on the number of clustered scans, open cursors, lock requests, checkpoints, buffer scans, average wait time for locks, and many such statistics that give detailed insight into a *snapshot* of the live system. In Oracle, a report containing this information can be generated by running a script called UTLBSTAT.SQL to initiate monitoring and a script UTLBSTAT.SQL to terminate monitoring. The system catalog contains details about the sizes of tables, the distribution of values in index keys, and the like. The plan generated by the DBMS for a given query can be viewed in a graphical display that shows the estimated cost for each plan operator. While the details are specific to each vendor, all InajaI' DBMS products on the market today provide a suite of such tools.

To create a good physical database design and tune the system for performance in response to evolving user requirements, a designer must understand the workings of a DBMS, especially the indexing and query processing techniques supported by the DBMS. If the database is expected to be accessed concurrently by many users, or is a *distributed database*, the task becomes more complicated and other features of a DBMS come into play. We discuss the impact of concurrency on database design in Section 20.10 and distributed databases in Chapter 22.

20.1.1 Database Workloads

The key to good physical design is arriving at an accurate description of the expected workload. A workload description includes the following:

1. A list of queries (with their frequency, as a ratio of all queries / updates).
2. A list of updates and their frequencies.
3. Performance goals for each type of query and update.

For each query in the workload, we must identify

- Which relations are accessed.
- Which attributes are retained (in the SELECT clause).

- \Vhich attributes have selection Of join conditions expressed on thern (in the WHERE clause) and how selective these conditions are likely to be.

SiInilarly, for each update in the \vorkloacl, we Blust identify

- Which attributes have selection or join conditions expressed on therll (in the WHERE clause) and how selective these conditions are likely to be.
- The type of update (INSERT, DELETE, or UPDATE) and the updated relation.
- For UPDATE cOHnuands, the fields that are rnodified by the update.

R.elllleluber that queries and updates typically have parameters, for exarnple, a debit or credit operation involves a particular account nUlnber. rrhe values of these paralneters deterlnine selectivity of selection and join conditions.

Updates have a query cornponent that is used to find the target tuples. This cOlllponent can benefit froIn a good physical design and the presence of indexes. On the other hand, updates typically require additional work to ITlaintain indexes on the attributes that they 111odify. Thus, while queries can only benefit froill the presence of an index, an index rnay either speed up or slow down a given update. Designers should keep this trade-off in rnind when creating indexes.

20.1.2 Physical Design and Tuning Decisions

Irnportant decisions rnade during physical database design and database tuning include the follovving:

1. Choice of indexes to create:
   - Which relations to index and which field or cornbination of fields to choose as index search keys.
   - For each index, should it be clustered or ul1clustered?
2. *Tuning the conceptual schema:*
   - *Alternative normalized schemas:* We usually have rnore than one way to decompose a schclua into a desired IlOl'Inal fOlTn (BCNF or 3NF). A choice can be rnade on the basis of perforrnance criteria.
   - *Denormalization:* We might want to reconsider scherna decolnposibons ca.rried Olll t for norrnalization. during the conceptual schern.a design process to irnprove the perforrnance of queries that involve attributes fr0111 several previously decornposed relations.

- *Vertical partitioning:* Under certain circurnstances we rnight 'want to further decornpose relations to ilnprove the perfornlance of queries that involve only a few attributes.
- *Views:* We luight 'want to add sorne views to nlask the changes in the conceptual scherna fr0111 users.

3. *Query and transaction tuning:* Frequently executed queries and transactions ulight be rewritten to run faster.

In parallel or distributed databases, which we discuss in Chapter 22, there are additional choices to consider, such as whether to partition a relation across different sites or whether to store copies of a relation at multiple sites.

20.1.3 Need for Database Thning

Accurate, detailed workload infonnation 111Cly be hard to corne by while doing the initial design of the systen1. Consequently, tuning a database after it has been designed and deployed is ilnportant---we HlllSt refine the initial design in the light of actual usage patterns to obtain the best possible perfonnance.

The distinction bet\veen database design and database tuning is soruewhat arbitrary. We could consider the design process to be over once an initial conceptual sche1na is designed and a set of indexing and clustering decisions is nlade. Any subsequent changes to the conceptual scherna or the indexes, say, would then be regarded as tuning. Alternatively, we could consider sorne refinernent of the conceptual scheula (and physical design decisions afl'ected by this refinernent) to be part of the physical design process.

Where we draw the line between design and tuning is not very irnpoItant, and we sirnply discuss the issues of index selection and database tuning without regard to when the tuning is carrier} out.

20.2 GUIDELINES FOR INDEX SELECTION

In considering which indexes to create, we begin with the list of queries (including queries tha,t a.ppear as part of update operations). ()bviously, only relations accessed by some query need to be considered as candidates for indexing, and the choice of attributes to index is guided by the conditions that appear in the WHERE clauses of the queries in the \vorkload. The presence of suitable indexes can significantly irnprove the evaluation plan for (1, query, as we saw in Chapters 8 and 12.

One approach to index selection is to consider the Ulost irnportant queries in turn, and, for each, deterrnine \vhich plan the optimizer would choose given the indexes currently on our list of (to be created) indexes. Then\ve consider whether we can arrive at a substantially better plan by adding more indexes; if so, these additional indexes are candidates for inclusion in our list of indexes. In general, range retrievals benefit froIn a B+ tree index, and exact-IHatch retrievals benefit frorn a hash index. Clustering benefits range queries, and it benefits exact-rnatch queries if several data entries contain the saIne key value.

Before adding an index to the list, however, we lnust consider the impact of having this index on the upda,tes in our workload. As we noted earlier, although an index can speed up the query cornponent of an update, all indexes on an updated attribute---{)n *any* attribute, in the case of inserts and deleteslnust be updated whenever the value of the attribute is changed. Therefore, we must sOlnetirnes consider the trade-off of slo\ving sorne update operations in the workload in order to speed up some queries.

Clearly, choosing a good set of indexes for a given workload requires an understanding of the available indexing techniques, and of the workings of the query optiruizer. The following guidelines for index selection sunnnarize our discussion:

Whether to Index (Guideline 1): The obviollS points are often the lnost important. Do not build an index unless sorne query- including the query cOlnponents of updates benefits frolu it. Whenever possible, choose indexes that speed up rIlore than one query.

Choice of Search Key (Guideline 2): Attributes rnentioned m a, WHERE clause are ca,ndidates for indexing.

- An exact-match selection condition suggests that we consider an irldex on the selected attributes, ideally, a hash index.
- *j*\ range selection condition suggests that we consider a 13+- tree (Of ISAM) index on the selected attrilnltes. *j*\ B+ tree index is usually preferable to an ISAM index. A.n ISAM irlclex rnay be worth considering if the relation is infrequently updated, but we assume that a B+ tree index is always chosen over an ISAM index, for sirnplicity.

Multi-Attribute Search :Keys (Guideline 3): Indexes with multiple-attribute search keys slH)uld be considered in the follc)\ving two situ<ltion.s:

- *j*\ WHERE clause includes conditinns 011 Inore t.han one attribute of a relation.

- They enable index-only evaluation strategies (i.e., accessing the relation can be avoided) for important queries. (This situation Gould lead to attributes being in the search key even if they do not appear in WHERE clauses.)

When creating indexes on search keys with rnultiple attributes, if range queries are expected, be careful to order the attributes in the search key to match the quenes.

Whether to Cluster (Guideline 4): At lllost one index on a given relation can be clustered, and clustering affects perfonnance greatly; so the choice of clustered index is iInportant.

- As a rule of tlnunb, range queries are likely to benefit the 1110St froIll clustering. If several ra,nge queries are posed on a relation, involving different sets of attributes, consider the selectivity of the queries and their relative frequency in the workload when deciding which index should be clustered.
- If an index enables an index-only evaluation strategy for the query it is intended to speed up, the index need not be clustered. (Clustering Inatters only when the index is used to retrieve tuples fr(nll the underlying relation.)

Hash versus Tree Index (Guideline 5): A B+ tree index is usually preferable because it supports range queries as well as equality queries. A hash index is better in the following situations:

- The index is intended to support index nested loops join; the indexed relation is the inner relation, and the search key includes the join colurllns. In this case, the slight ilnproveIllent of a hash index over a B+ tree for equality selections is rnagnified, because an equality selection is generated for each tuple in the outer relation..
- rrllcre is a very important equality query, and no range queries, involving the sea.rch key attributes.

Balancing the Cost of Index Maintenance (Guideline 6): After drawing up a 'wishlist' of indexes to create, consider the irnpact of each index on the updates in the workload.

- If maintaining an index slows down frequent update operations, consider dropping the index.
- Keep ill mind, however, that adding an index Illay 'well speed up a given update operation. For exanlplc, an index on employee IDs could speed up the operation of increasing the salary of a given ernployee (specified by ID).

20.3 BASIC EXAMPLES OF INDEX SELECTION

The follawing examples illustrate how to choose indexes during database design, continuing the discussion froln Chapter 8, where we focused on index selection for single-table queries. The schernas used in the exarnples are not described in detail; in general, they contain the attributes nalned in the queries. Additional inforlnation is presented when necessary.

Let us begin with a silnple query:

```
SELECT  E.enaIne, D.rugI'
FROM    Enlployees E, Departments D
WHERE   D.dname='Toy' AND E.dno=D.dno
```

The relations rnentioned in the query are Enlployees and Departnlents, and both conditions in the WHERE clause involve equalities. Our guidelines suggest that we should build hash indexes on the attributes involved. It seeIns clear that we should build a hash index on the *dnaTne* attribute of Departments. But consider the equality *E.dno=D.dno*. Should we build an index (hash, of course) on the *dno* attribute of Departrnents or Ernployees (or both)? Intuitively, we want to retrieve Departments tuples using the index on *dnarne* because few tuples are likely to satisfy the equality selection *.D.dnaTne='Toy'*.[1] For each qualifying Departrnents tuple, we then find lnatching EInployees tuples by using an index on the *dno* attribute of Ernployees. So, we should build an index on the *dno* field of En1ployees. (Note that nothing is gained by building an additional index on the *dno* field of Departrnents because Departnlents tuples are retrieved using the *dna:rne* index.)

Our choice of indexes was guided by the query evaluation plan we wanted to utilize. This consideration of a, potential evaluation plan is connnon while rnaking physical design decisions. U·nderstanding query optirnization is very useful for physical design. We show the desired plan for this query in Figure 20.1.

As a variant of this query, suppose that the WHERE clause is rnodified to be WHERE *J).dnarne='Toy'* AND E'.*dno=D.dno* AND *E.age=25*. Let us consider alternative evaluation plans. ()ne good plan is to retrieve Departrnents tuples that satisfy the selection on *dnarne* and retrieve rnatching Ernployees tuples by using an index on the *dno* field; the selection on *age* is then applied on-the-fly. However, unlike the previous variant of this query, we do not really need to have an index on the *dna* field of Ernployees if we have an index. on *age*. In this

[1] This is only a heuristic. If *dname* is not the key, and we have no statistics to verify this cla.inl. it is possible that several tuples satisfy this condition.

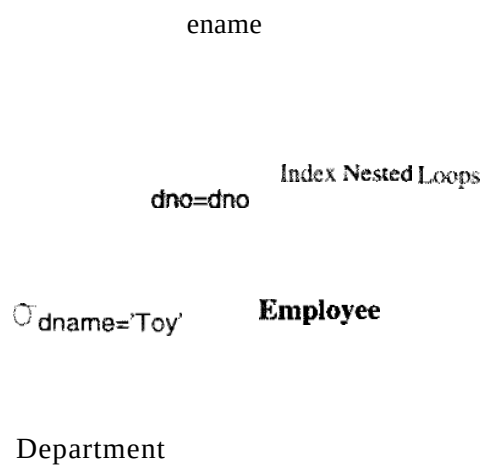


Figure 20.1 A Desirable Query Evaluation Plan

case we can retrieve Departrnents tuples that satisfy the selection on *dnarrte* (by using the index on *dname,* as before), retrieve Ernployees tuples that satisfy the selection on *age* by using the index on *age,* and join these sets of tuples. Since the sets of tuples we join are srnall, they fit in 111emory and the join Inethod is unirnportant. This plan is likely to be sornewhat poorer than using an index on *dno,* but it is a reasonable alternative. rrherefore, if we have an index on *age* already (prolnpted by sorne other query in the workload), this variant of the sarnple query does not justify creating an index on the *dno* field of Ernployees.

Our next query involves a range selection:

```
SELECT  E.enan1e, I),dnarne
FROM    Elnployees E, Departrnents D
WHERE   B.sal BETWEEN 10000 AND 20000
        AND E.hobby='Starups' AND E.dno=D.dno
```

This query illustrates the use of the BETWEEN operator for expreSSIng range selections. It is equivalent to the condition:

$$10000 \leq \text{E.sal AND E.sal} \leq 20000$$

The use of BETWEEN to express rarlge conditions is reconunended; it lnakes it easier for both the user and the optilnizer to recognize both parts of the range selection.

Returning to the exarnple query, both (nonjoin) selections are on the Ernployees relation. Therefore, it is clear that a plan in which Eluployees is the outer relation and I)epartrnents is the inner relation is the best, as in the previous query, and we should build a hash index on the *dno* attribute of Departlnents. But which index should we build on Ernployees? A B+ tree index on. the *sal* attribute would help with the range selection, especially if it is clustered. A

hash index on the *hobby* attribute would help ·with the equality selection. If one of these indexes is available, we could retrieve Ernployees tuples using this index, retrieve rnatching Departruents tuples using the index on *dno*, aJld apply all rernaining selections and projections on-the-fly. If both indexes are available, the optirnizer would choose the mo1'e selective index for the given query; that is, it \volllld consider which selection (the range condition on *salary* or the equality on *hobby)* has fe\ver qualifying tuples. In general, which index is rnore selective depends on the data. If there are very few people with salaries in the given range and rnany people collect starnps, the B-t- tree index is best. Otherwise, the hash index on *hobby* is best.

If the query constants are known (as in our exarnple), the selectivities can be estiInated if statistics on the data are available. Otherwise, as a rule of thurnb, an equality selection is likely to be rnore selective, and a reasonable decision would be to create a hash index on *hobby.* Sornethnes, the query constants are not known—we rnight obtain a query by expanding a query on a view at rUIl-tirrle, or we rnight have a query in Dynalnic SQL, which allows constants to be specified as *wild-card variables* (e.g., *%X)* and instantiated at run-tinle (see Sections 6.1.3 and 6.2). In this case, if the query is very important, we lllight choose to create a B+ tree index on *sal* and a hash index on *hobby* and leave the choice to be rnade by the optirnizer at run-tirrle.

20.4 CLUSrrERING AND INDEXING

Clustered indexes can be especially iInportant while accessing the inner relation in an index nested loops join. To understand the relationship between clustered hldexes and joins, let us revisit our first exarnple:

```
SELECT  E.enanle, D.rngr
FROM    Employees E, I)epartrnentsD
WHERE   D.dname='Toy' AND E.dno=D.dno
```

We concluded that a good evaluation plan is to use an index on *dname* to retrieve Departments tuples satisfying the condition on *dnarne* and to find. rnatching Ernployees tuples using an index on *dna.* Should these indexes be clustered? G·iven our asslunption that the number of tuples satisfying 1). *dname='Toy'* is likely to be small, we should build an unclustered index on *dname.* ()n the other hand, Employees is the inner relation in an index nested loops join and *dna* is not a candidate key. This situation is a strong argument that the index on the *dno* field of Ernployees 8ho111(1 be clustered. In fact, because the join consists of repeatedly posing equality selections on the dnofield of the inner relation, this type of quer,Y is a stronger justification for rnaking the index on *dno* clustered than a sirnple selection query such as the previous selection on

hobby. (Of courso, factors such as selectivities and frequency of queries have to be taken into account as well.)

'rhe following oxaluplc, very sirnilar to the previous one, illustrates how clustered indexes can be used for sort-rnerge joins:

```
SELECT  E.enarne,D.rngr
FROM    Ernployees E, DepartlTlents D
WHERE   E.hobby='Starnps' AND E.dno=D.dno
```

This query differs froIll the previous query in that the condition *E.hobby='Starnps'* replaces *D.dname='Toy'*. Based on the assumption that there are few ernployees in the Toy departrnent, we chose indexes that would facilitate an indexed nested loops join with DepartlTlents as the outer relation. Now, let us suppose that rllany ernployees collect stamps. In this case, a block nested loops or sort-rnerge join Blight be rnore efficient. A sort-rnerge join can take advantage of a clustered B+ tree index on the *dno* attribute in Departrnents to retrieve tuples and thereby avoid sorting Departrnents. Note that an unelustered index is not useful----since all tuples are retrieved, performing one I/O per tuple is likely to be prohibitively expensive. If there is no index on the *dno* field of Ernployees, we could retrieve Ernployees tuples (possibly using an index on *hobby*, especially if the index is clustered), apply the selection *E.hobby='Starnps'* on-the-fly, and sort the qualifying tuples on *dno*.

As our discllssion has indicated, when we retrieve tuples using an index, the irnpact of clustering depends on the rnunber of retrieved tuples, that is, the nuruber of tuples that satisfy the selection conditions that rnatch the index. An unclustered index is just as good as a clustered index for a selection that retrieves a single tuple (e.g., an equality selection on a candidate key). As the llurnber of retrieved tuples increases, the unclustered index quickly becoHlcs rnore expensive than e'ven a sequential scan of the entire relation. Although the sequential scan retrieves all tuples, each page is retrieved exactly once, whereas a page rIlay be retrieved as often as the rnunber of tuples it contains if an unclustered index is usee!' If blocked l/C) is performed (as is COl1nnon), the relative advantage of sequential scan versus an llnclustered index increases further. (Blocked T/C) also speeds up access using a clustered index, of course.)

We illustrate the relationship between the number of retrieved tuples, viewed as a percentage of the total nurnber of tuples in the relation, and the cost of various access rnethods in .Figure 20.2. We assume that the query is a selection on a single relation, for sirnplicity. (Note that this figure reflects the C()st of writing out the result: otherwise, the line for seqnential scan weHlld be flat.)

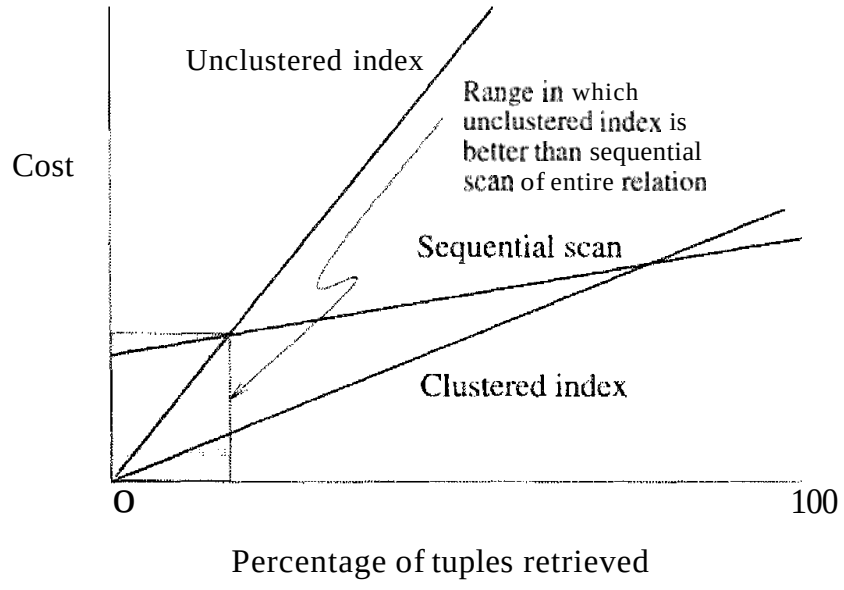


Figure 20.2 The Impact of Clustering

20.4.1 Co-clustering Two Relations

In our description of a typical database systern architecture in Chapter g, we explained how a relation is stored as a file of records. Although a file usually contains only the records of SOIn8 one relation, SCHue systeIlls allow records frorn Inore than one relation to be stored in a single file. rrhe database user can request that the records froIll two relations be interleaved physically in this 111anne1'. This data layout is sornetiInes referred to as co-clustering the two relations. We now discuss when co-clustering can be beneficial.

As an exarnple, consider two relations with the following schernas:

Parts(pid: integer, *pnarne:* string, *cost:* integer, *8upplierid:* integer)
Assembly(*partid:* integer, *componentid:* integer, *quantity:* integer)

In this scherna the *componentid* field of Assernbly is intended to be the *pid* of sorne part that is used as a cornponent in assernbling the part with *pid* equal to *partid.* Therefore, the Assernbly table represents a l:N relationship between parts and their subparts; a part can have rnany s11bparts, but each part is the subpart of at rnost one part. In, the Parts table, *pid* is the key. For cOlnposite parts (those assernbled frorIl other parts, as indicated by the contents of Assclnb1y), the *cost* field is taken to be the cost of assembling the part frorn its subparts.

Suppose that a frequent query is to find the (inllnediate) subparts of all parts supplied by a given supplier:

```
SELECT  P.piel, .A.componentid
FROM    Parts 'J'), Assembly A
```

```
WHERE   P.piel = A.partid AND P.supplierid = 'Acme'
```

A good evaluation plan is to apply the selection condition on Parts and then retrieve rnatching Asselnbly tuples throngh an index on the *partid* field. Ideally, the index on *partid* should be clustered. rrhis plan is reasonably good. :However, if such selections are COHU110n and we want to optirnize thorn further, we can *co-cluster* the two tables. In this approach, we store records of the two tables together, \vith each Parts record P follc)\ved by all the Assernbly records A such that $P.pid = A.partid$. This approach improves on storing the two relations separately and having a, clustered index on *paTtid* because it does not need an index lookup to find the A.ssernbly records that rnatch a given Parts record. Thus, for each selection query, we save a few (typically two or three) index page l/Os.

If we are interested in finding the imrnediate subparts of *all* parts (i.e., the preceding query with no selection on *supplierid)*, creating a clustered index on *partid* and doing an index nested loops join with Assembly as the inner relation offers good perfonnance. An even better strategy is to create a clustered index on the *paTtid* field of Assernbly and the *pid* field of Parts, then do a sort-rnerge join, using the indexes to retrieve tuples in sorted order. This strategy is comparable to doing the join using a co-clustered organization, which involves just one scan of the set of tuples (of Parts and Asselnbly, which are stored together in interleaved fashion).

The real benefit of co-clustering is illustrated by the following query:

```
SELECT  P.pid,A.componentid
FROM    Parts P, Assernbly A
WHERE   P.pid = A.partid AND P.cost=10
```

Suppose that rnany parts have *cost* = 10. This query essentially a,rnonnts to a collection of queries in which we are given a Parts record and want to find rnatching Assernbly records. If we have an index on the *cost* field of Parts, we can retrieve qualifying Parts tuples. For each such tuple, we have to use the index on Assernbly to locate records with the given *pid*. rrhe index access for A.ssernbly is avoided if we have a co-clustered organization. (()f courS8, we still require all index on the *cost* attribute of Parts tuples.)

Such an optirnization is especially irnportant if we want to traverse several levels of the part-subpart hierarchy. .For example, a COnll110Il query is to find the totaJ cost of a part, which requires us to repeatedly carry out joins of Pa,rts and Asscrnbly. Incidentally, if we do not know the nurnber of levels in the hierarchy in advance, the nUlnber of joins varies and the query cannot be expressed in SQL. The query can be answered by ernbedcling an SQL statement

for the join insicie an iterative host language prograrll. How to express the query is orthogonal to our lnain point here, which is that co-clustering is especially beneficial when the join in question is carried out very frequently (either because it arises repeatedly in an important query such as finding total cost, or because the join query itself is asked frequently).

To sUIlunarize co-clustering:

- It can speed up joins, in particular key·foreign key joins corresponding to l:N relationships.
- A sequential scan of either relation becornes slower. (In our exalnple, since several Assenlbly tuples are stored in between consecutive Parts tuples, a scan of all Parts tuples becornes slower than if Parts tuples were stored separately. SiInilarly, a sequential scan of all Assernbly tuples is also slower.)
- All inserts, deletes, and updates that alter record lengths becorne slower, thanks to the overheads involved in ruaintaining the clustering. (We do not discuss the irnplernentation issues involved in co-clustering.)

20.5 INDEXES THAT ENABLE INDEX-ONLY PLANS

This section considers a nU111ber of queries for which we can find efficient plans that avoid retrieving tuples froln one of the referenced relations; instead, these plans scan an associated index (which is likely to be lnuch srnaller). An index that is used (only) for index-only scans does *not* have to be clustered because tuples fronl the indexed relation are not retrieved.

This query retrieves the lnanagers of depal'truents with at least one ernployee:

```
SELECTD.rugr
FROM    Departrnents I), Employees E
WHERE   I).dno=E.dno
```

()bserve that no attributes of Ernployees are retained. If we have an index on the *dno* field of Employees, tl1e optirnization of doirlg an index nested loops join using an index-onl:y searl for the inner relation is applicable. Given this variant of the query, the correct decision is to build an uncIustered index on the *dna* field of Employees, rather thaIl a clustered index.

rrhe next query takes this idea a step further:

```
SELECT  ]).rngr, E.eid
FROM    ])epartrnents I), Employees E
WHERE   D.dno=E.dno
```

If we have an index on the *dno* field of Employees, we can use it to retrieve Employees tuples during the join (\vith Departrnents as the outer relation), but unless the index is clustered, this approach is not be efficient. ()n the other hand, suppose that we have a B+- tree index on *(dna, e'id).Now* all the inforrnation we need about an Employees tuple is contained in the data entry for this tuple in the index. We can use the index to find the first data entry \vith a given *elno;* all data entries 'with the same *dno* are stored together in the index. (Note that a hash index on the cOlnposite key *(dna, eid)* cannot be used to locate an entry with just a given *dno!)* We can therefore evaluate this query using an index nested loops join with Departlnents as the outer relation and an index-only scan of the inner relation.

20.6 TOOLS TO ASSIST IN INDEX SELEC"fION

The rUllnber of possible indexes to consider building is potentially very large: For each relation, we can potentially consider all possible subsets of attributes as an index key; we have to decide on the ordering of the attributes in the index; and we also have to decide which indexes should be clustered and which unclustered. Many large applications---for exalnple enterprise resource planning systems—create tens of thousands of different relations, and rnanual tuning of such a large schelna is a daunting endeavor.

The difficulty and irnportance of the index selection task rnotivated the development of tools that help database adrninistrators select appropriate indexes for a given workload. The first generation of such index tuning wizards, or index advisors, were separate tools outside the database engine; they suggested indexes to build, given a workload of SQL queries. rfhe rnain drawback of these systerns was that they had to replicate the database query optirnizer's cost rnodel in the tuning tool to rnake sure that the optimizer would choose the sanlC query evaluation plans as the design tool. Since query optirnizers cha,nge froIn release to release of a conunercial database systern, considerable effort was needed to keep the tuning tool and the database optirnizer synchronized. The rnost recent generation of tuning tools are integrated \vith the database engine and use the database query optiluizer to estimate the cost of a workload given a set of indexes, avoiding duplication of the query optirnizer's cost rnodel into an external tool.

20.6.1 Automatic Index Selection

\{Ve call a set of indexes for a given database scherna. an index configuration. We assume that a, query workload is a set of queries over a database scherna where each query has a frequency of occurrence assigned to it. Given a database schelna and a, workload, the cost of an index configuration is the expected

cost of running the queries in the workload given the index configuration taking the different frequencies of queries in the workload into account. Given a database schema and a query workload, we can now define the problel11 of automatic index selection as finding an index configuration with nlinirnal cost. A.s in query optinlization, in practice our goaJ is to find a *good* index configuration rather than the true optirnal configuration.

Why is automatic index selection a hard problern? Let us calculate the nUlnber of different indexes with c attributes, assurning that the table has n attributes. For the first attribute in the index, there are n choices, for the second attribute $n-1$, and thus for a c attribute index, there are overall $n \cdot (n-1) \ldots (n-c+1) = \frac{n!}{(n-c)!}$ different indexes possible. The total nurnber of different indexes with up to c attributes is

$$\sum_{i=1}^{c} \frac{n!}{(n-i)!}$$

For a table with 10 attributes, there are 10 different one-attribute indexes, 90 different two-attribute indexes, and 30240 different five-attribute indexes. For a cornplex workload involving hundreds of tables, the nurnber of possible index configurations is clearly very large.

The efficiency of autornatic index selection tools can be separated into two components: (1) the nurnber of candidate index configurations considered, and (2) the nurnber of optimizer calls necessary to evaluate the cost for a configuration. Note that reducing the search space of candidate indexes is analogous to restricting the search space of the query optiInizer to left-deep plans. In lnany cases, the optirnal plan is not left-deep, but alllong all left-deep plans there is usually a plan whose cost is close to the optirnal plan.

We can easily reduce the time taken for autornatic index selection by reducing the nUlnber of candidate index configurations, but the srnaller the space of index. configurations considered, the farther away the final index configuration is [1'0111 the optirnal index configllration. rrherefore, different index tuning \vizards prune the search space differently, for exarnple, by considering only one- or two-attribute indexes.

20.6.2 How Do Index Thning Wizards Work?

All index tuning \vizards search a set of candidate indexes for an index configuration with lowest cost. Tools differ in the space of candidate index configurations they consider and how they search this space. We describe one representative algoritlun; existing tools implement 'variants of this algorithrn, hut their irnplernentations have the sanIe basic structure.

The DB2 Index Advisor. The DB2 Index Advisor is a tool for automatic index recommendation given a workload. The workload is stored in the database systell1 in a tabl'e called ADVISE_WORKLOAD. It is populated either (1) by SQL statcrnents 1'10111 the DB2 dynanlic SQL statelnent cache, a cache for recently executed SQL statenlents, (2) with SQL staternents frol11 packages·..--groups of statically cornpiled SQL statenlents, or (3) with SQL statelnents frolll an online monitor called the Query Patroller. The DB2 Advisor allows the user to specify the lnaximuill arnount of disk space for new indexes and a rnaxirrnul1 time for the cornputation of the recomrnended index configuration.

The DB2 Index Advisor consists of a prograrrl that intelligently searches a subset of index configurations. Given a candidate configuration, it calles the query optirnizer for each query in the ADVISE_WORKLOAD table first in the RECOMMEND_INDEXES rnode, where the opthnizer recommends a set of indexes and stores thern in the ADVISE_INDEXES table. In the EVALUATE_INDEXES mode, the optimizer evaluates the benefit of the index configuration for each query in the ADVISE-WORKLOAD table. The output of the index tuning step is are SQL DDL statenlents whose execution creates the recomrnended indexes.

The Microsoft SQL Server 2000 Index Tuning Wizard. Microsoft pioneered the irnplell1entation of a tuning wizard integrated with the database query optiInizer. The Microsoft Tuning vVizard has three tuning rnodes tha.t perrnit the user to trade off running time of the analysis and nurnber of candidate index configurations exarnined: *fast*, *medium*, and *thorough*, with *fast* having the lo\vest running tirne aJld *thoTo'ugh* exalnining the largest nUlnber of configurations. rro further reduce the running time, the tool has a salnpling Inode in which the tuning wizard randoruly salllpics queries fronl the input workload to speed up analysis. Other parameters include the lnaxirnurn space allowed for the reeornmended indexes, the. maximum nurnber of attributes per index considered, and the tables on which Indexes can. be generated. The Microsoft Index Tuning WIzard also, perunts *table scaling*, where the user can specify an anticipated nurnber of records for the tables involved in the workload. This allows users to plan for future growth of the tables.

Before we describe the index tuning algoriUlIn, let us consider the problell1 of estiInating the cost of a configuration. Note that it is not feasible to actually create the set of indexes in a candidate configuration and then optirnize the query workload given the physical index configuration. Creation of even a single candidate configuration with several indexes lIlight take hours for large databases and put considerable load on the database systerIl itself. Since we want to exauline a large nUlnber of possible candidate configurations, this approach is not feasible.

Therefore index tuning algorithrIls usually *simulate* the effect of indexes in a candidate configuration (unless such indexes already exist). Such what-if indexes look to the query optilIlizer like any other index and are taken into account when calculating the cost of the workload for a given configuration, but the creation of what-if indexes does not incur the overhead of actual index creation. Commercial database systelIls that support index tuning wizards using the database query optirnizer have been extended with a module that permits the creation and deletion of what-if indexes with the necessary statistics about the indexes (that are used when estirnating the cost of a query plan).

We now describe a representative index tuning algorithm. The algorithm proceeds in two steps, *candidate index selection* and *configuration enumeration.* In the first step, we select a set of candidate indexes to consider during the second step as building blocks for index configurations. Let us discuss these two steps in Inore detail.

Candidate Index Selection

We saw in the previous section that it is iInpossible to consider every possible index, due to the huge nUluber of candidate indexes available for larger database schernas. ()ne heuristic to prune the large space of possible indexes is to first tune each query in the workload independently and then select the union of the indexes selected in this first step as input to the second step.

:F'or a query, let us introduce the notion of an indexable attribute, which is an attribute whose appearance in an index could change the cost of the query. An indexable attribute is an attribute on \vhich the WHERE-part of the query has a condition (e.g., an equality predicate) or the attribute appears in a GROUP BY or ORDER BY clause of the SQL query. An admissible index for a query is an index that contains only indexable attributes in the query.

How do we select candidate indexes for an individual query? ()ne approach is a basic enumeration of all indexes with up to k attributes. \Ve start \ivith all indexable attributes as single attribute candidate indexes, then add all com-

binations of two indexable attributes as candidate indexes, and repeat this procedure until a user-defined size threshold *k.* This procedure is obviously very expensive as we add overall $n + n \cdot (n - 1) + \ldots + n \cdot (n - 1) \ldots (n - k + 1)$ candidate indexes, but it guarantees that the best index with up to *k* attributes is aII10ng the candidate indexes. The references at the end of this chapter COII-tain pointers to faster (but less exhaustive) heuristieal search algorithrns.

Enumerating Index Configurations

In the second phase, we use the candidate indexes to enUInerate index configurations. As in the first phase, we can exhaustively enurnerate all index configurations up to size *k,* this time cornbining candidate indexes. As in the previous phase, more sophisticated search strategies are possible that cut down the number of configurations considered while still generating a final configuration of high quality (i.e., low execution cost for the final workload).

20.7 OVERVIEW OF DATABASE TUNING

After the initial phase of database design, actual use of the database provides a valuable source of detailed information that can be used to refine the initial design. Many of the original assumptions about the expected workload can be replaced by observed usage patterns; in general, some of the initial workload specification is validated, and some of it turns out to be wrong. Initial guesses about the size of data can be replaced with actual statistics frorn the systern catalogs (although this inforrnation keeps changing as the systern evolves). Carefulrnonitoring of queries can reveal unexpected problerlls; for example, the optimizer lllay not be using SOIne indexes as intended to produce good plans.

Continued database tuning is irnportant to get the best possible perforrnance. In this section, we introduce three kinds of tuning: *tuning indexes, tuning the conceptual scherna,* and *tuning queries.* OUf discussion of index selection also applies to index tuning decisions. Conceptual schcrna and query tuning are discussed further in Sections 20.8 and 20.9.

20.7.1 Thning Indexes

The initial choice of indexes rnay be refined for one of several reasons. The sirnplest reason is that the observed \vorkload reveals that scnne queries and updates considered irnportant in the initial\vorkload specification are not very frequent. The observed workload rnay a1so identify SCHne new queries and updates that *are* inlportant.The initial choice of indexes has to be reviewed in light of this new inforrnation. Scnne of the original in,dexes rnay be dropped and

new ones added. The reasoning involved is siInilar to that used in the initial design.

It Inay also be discovered that the optimizer in a given systenl is not finding some of the plans that it was expected to. For exaluple, consider the following query, which we discussed earlier:

```
SELECT  D.Ingr
FROM    Ernployees E, Departulents D
WHERE   D.dnamc='Toy' AND E.dno=D.dno
```

A good plan here would be to use an index on *dnarne* to retrieve Departnlents tuples with *dnarne= 'Toy'* and to use an index on the *dno* field of Employees as the inner relation, using an index-only scan. Anticipating that the optirnizer would find such a plan, we rnight have created an unclustered index on the *dno* field of Ernployees.

Now suppose queries of this fonn take an unexpectedly long time to execute. We can ask to see the plan produced by the optiInizer. (Most commercial systerIls provide a simple cOillrnand to do this.) If the plan indicates that an index-only scan is not being used, but that Employees tuples are being retrieved, we have to rethink our initial choice of index, given this revelation about our system's (unfortunate) lhnitations. An alternative to consider here would be to drop the unclustered index on the *dno* field of EUlployees and replace it with a clustered index.

SOUle other COllllnon lirnitations of optiInizers are that they do not handle selections involving string expressions, arithrnetic, or *null* values effectively. We discuss these points further when we consider query tuning in Section 20.9.

In addition to re-exarnining our choice of indexes, it pays to periodically reorganize S(Hne indexes. For example, a static index, such as an ISAM index, Illay have developed long overflow chains. Dropping the index and rebuilding it—if feasible, given the interrupted access to the indexed relation—can substantially irnprove access tiTHes through this index. Even for a dynarnic structure such as a 13+ tree, if the implernentation does not rnerge pages on deletes, space occupancy can decrease considerably in SaIne situations. This in turn rnakes the size of the index (in pages) larger than necessary, and could increase the height and therefore the access tilne. Ilebuilding the index should be considered.Extensive updates to a clustered index rnight also lead to overflow pages being allocated, thereby decreasing the degree of clustering. Again, rehuilding the index Inay be vvorthwhile.

Finally, note that the query optinlizer relies on statistics rnaintained in the SystCIll catalogs. These statistics are updated only when a special utility progranl is run; be sure to run the utility frequently enough to keep the statistics reasonably current.

20.7.2 Thning the Conceptual Schema

In the course of database design, we rnay realize that our current choice of relation SChelllaS does not enable us rneet our perforrnance objectives for the given workload with any (feasible) set of physical design choices. If so, we 11lay have to redesign our conceptual scherna (and re-exarnine physical design decisions affected by the changes we rnake).

We rnay realize that a redesign is necessary during the initial design process or later, after the systern has been in use for a while. Once a database has been designed and populated with tuples, changing the conceptual scherna requires a significant effort in tenrlS of rnapping the contents of the relations affected. Nonetheless, it rnay be necessary to revise the conceptual scherna in light of experience with the systern. (Such changes to the schema of an operational systerll are sometirnes referred to as schema evolution.) We now consider the issues involved in conceptual scherna (re)design frorn the point of vie\v of perforrnance.

rrhe rnain point to understand is that *our choice of conceptual 8cherna should be guided by a cons'ideration of the queries and 'updates in our 'workload"* in addition to the issues of redundancy that rllotivate nonnalization (which we discussed in Chapter 19). Several options rnust be considered while tuning the conceptual scherna:

- We lIlay decide to settle for a 3NF design instead of a BCN'F design.
- If there are two ways to decornpose a given schelna into 3NF or BCNF, our choice should be guided by the workload.
- Sometimes we rnight decide to further decornpose a relation that is *already* rn BCNF.
- In other situations, we rnight *denormalize.* rrhat is, we rnight choose to replace a collection of relations obtained by a dec(nnposition frorn a larger relation with the original (larger) relation, even though it suffers frorn 80rne redundancy problerl1s. Alternatively, we rnight choose to add sorne fields to certain relations to speed up SCHne irnportant queries, even if this leads to a redundant storage of 80rne infonnation (anei, consequently, a scherna that is in neither 3NF nor BCNF).

- This discussion of nonnalization has concentrated on the technique of *decomposition*, 'which arnounts to vertical partitioning of a relation. Another technique to consider is *horizontal partitioning* of a relation, which \vould lead to having two relations with identical schernas. Note that we are not talking about physically partitioning the tuples of a single relation; rather, we want to create two distinct relations (possibly with different constraints and indexes on each).

Incidentally, when \ve redesign the conceptual scherna, especially if we are tuning an existing database sche1na, it is \vorth considering whether we should create views to rnask these changes fronl users for whOIII the original schcrlla is]nore natural. We discuss the choices involved in tuning the conceptual scherna in Section 20.8.

20.7.3 Thning Queries and Views

If we notice that a query is running rnuch slower than we expected, we have to exarnine the query carefully to find the problern. SaIne rewriting of the query, perhaps in conjunction with SCHne index tuning, can often fix the problern. Sirnilar tuning rnay be called for if queries on SaIne view run slower than expected. We do not discuss view tuning separately; just think of queries on views as queries in their own right (after all, queries on views are expanded to account for the view definition before being optirnized) and consider how to tune thern.

When tuning a query, the first thing to verify is that the systern uses the plan you expect it to use. Perhaps the systelll is not finding the best plan for a variety of reasons. Sorne COIllrllon situations not handled efficiently by rnany optinlizers follow:

- A selection condition involving *null* values.
- Selection conditions involving aritlunetic or string expressions or conclitions using the OR connective. For exarnple, if we have a conclitionE.*age* = 2*]).*age* in the WHERE clause, the optirnizer rnay correctly utilize an available index on *E.age* but fail to utilize an available index on 1).*age*. R,eplacing the condition by *E.age/2* = 1).*age* \vould reverse the situation.
- Inability to recognize a sophisticated plan such as an index-only scan for an aggregation query involving a GROUP BY clause. ()f course, virtually no optirnizer looks for plans outside the plan space described in Chapters 12 and 15, such as nonleft-deep join trees. So a good understanding of ·what an optirnizer typically does is irnportant. In addition, the rnore a:ware you are of a given systeur's strengths arrd lirnitations, the better off you arc.

If the optirnizer is not smart enough to find the best plan (using access Inethods and evaluation strategies supported by the DB.wIS), SOHIe systern.s allow users to guide the choice of a plan by providing hints to the opthnizer; for example, users rnight be able to force the use of a particular index or choose the join order and join rnethod. A user who wishes to guide optirnization in this Inanner should have a thorough understanding of both optirnizatioll and the capabilities of the given DBMS. We discuss query tuning further in Section 20.9.

20.8 CHOICES IN TUNING THE CONCEPTUAL SCHEMA

We now illustrate the choices involved in tuning the conceptual scheIua through several exarnples using the following schelnas:

Contracts(*cid:* integer, *s'Upplierid:* integer, *projectid:* integer,
deptid: integer, *partid:* integer, *qty:* integer, *value:* real)
Departments(*did:* integer, *budget:* real, *annualreport:* varchar)
Parts(pid: integer, *cost:* integer)
Projects(jid: integer, *rngr:* char(20))
Suppliers(*sid:* integer, *address:* char(50))

For brevity, we often use the cornrnon convention of denoting attributes by a single character and denoting relation schernas by a sequence of characters. Consider the scherna for the relation Contracts, whic.h we denote as CSJDPQV, with each letter denoting an attribute. The Ineaning of a tuple in this relation is that the contract with *cid* C is an agreernent that supplier S (with *sid* equal to *supplierid)* will supply Q iterns of part P (with *pid* equal to *partid)* to project J (with *jid* equal to *projectid*) associated with departrnent D (with *deptid* equal to *did),* and that the value V of this contract is equal to *value.*[2]

There are two known integrity constraints with respect to Contracts. A project purchases a given part using a single contract; thus, there cannnot be two distinct contracts in which the saIne project buys the saIne part. This constraint is represented using th.e FI) $.II) \rightarrow C$. Also, a departrnent purchases at rnost one part frolll any given supplier. This constraint is represented llsing the FD $SD \rightarrow P$. In addition, of course, the contract ID C is a key. The rneaning of the other relations should be obvious, and we do not describe thcrn further because we focus on the Contra.cts rela.tion.

2If this schema seems cornplicated, note that real-life situations often call for considerably more cornplex schemas!

20.8.1 Settling for a Weaker Normal Form

Consider the Contracts relation. Should we deconlpose it into sHlaller relations? Let us see what norrnal fann it is in. The candidate keys for this relation are C and JP. (C is given to be a key, and tIP functionally deterrnines C.) The only nonkey dependency is 3D $\rightarrow P$, and P is a *prime* attribute because it is part of candidate key JP. rrhus, the relation is not in BCNF—because there is a nonkey dependency-.:-but it is in 3NF.

By using the dependency $SD \rightarrow P$ to guide the decornposition, we get the two schemas SDP and CSJDQV. This decornposition is lossless, but it is not dependency-preserving. However, by adding the relation schelne CJP, we obtain a lossiess-join, dependency-preserving decoruposition into BCNF. Using the guideline that such a decorllposition into BCNF is good, we might decide to replace Contracts by three relations with schernas CJP, SDP, and CSJDQV.

However, suppose that the following query is very frequently asked: Find the nurnber of copies Q of part P ordered in contract C. 'rhis query requires a join of the decornposed relations CJP and CSJDQV (or SDP and CSJDQV), whereas it can be answered directly using the relation Contracts. The added cost for this query could persuade us to settle for a 3NF design and not decompose Contracts further.

20.8.2 Denormalization

The reasons rTIotivating us to settle for a weaker norrnal forIn may lead us to take an even rnore extrerne step: deliberately introduce SOlllC redundancy. As an exarnple, consider the Contracts relation, 'which is in 3NF. Now, suppose that a frequent query is to check that the value of a contract is less than the budget of the contracting departruent. We lllight decide to add a budget field B to Contracts. Since *did* is a key for Departrnents, we now have the dependency $D \rightarrow B$ in Contracts, \vhich InCt1IIS Contracts is not in 3NF any r11orc. Nonetheless, we rnight choose to stay with this design if the rnotivating query is sufficiently irnportant. Such a decision is clearly subjective and CaInes at the cost of significant redundancy.

20.8.3 Choice of Decomposition

Consider the Contracts relation again. Several choices are possible for dealing with the redundancy in this relation:

- We can leave Contracts as it is and accept the redundancy associated \\"ith its being in :3N:F rather than .BCNF.

- We Inight decide that we want to avoid the anolIlalies resulting froIn this redundancy by deeornposing Contracts into BCNF using one of the following ruethods:
  - We have a lossless-join decornposition into PartInfo with attributes SDP and Contractlnfo \vith attributes CSJDQV. As noted previously, this decornposition is not dependency-preserving, and to rnake it so would require us to add a third relation CJP, \vhose sale purpose is to allow us to cheek the dependency $JP \rightarrow C$.
  - We could choose to replace Contracts by just Partlnfo and Contract-Info even though this decornposition is not dependency-preserving.

R,eplacing Contracts by just Partlnfo and Contractlnfo does not prevent us frorll enforcing the constraint $JP \rightarrow C$; it only makes this n10re expensive. We could create an assertion in SQL-92 to check this constraint:

```
CREATE ASSERTION checkDep
CHECK      ( NOT EXISTS
             (SELECT *
             FROM      Partlnfo PI, Contractlnfo Cl
             WHERE     PI.supplierid=CI.supplierid
                   AND PI.deptid==CI.deptid
           GROUP BY  C1.projectid, PI.partid
           HAVING    COUNT (cid) > 1 ) )
```

This assertion is expensive to evaluate because it involves a join followed by a sort (to do the grouping). In cornparison, the systerll can check that JP is a prirnary key for table CJP by rnaintaining an index on JP. This difference in integrity-checking cost is the rl1otivation for dependency-preservation. On the other hand, if updates are infrequent, this increased cost IIlay be acceptable; therefore, we rnight choose not to rnaintain the table C.JP (and quite likely, an index all it).

As another exarnple illustrating decornposition choices, consider the Contracts relation again, and suppose that we also have the integrity constraint that a departrnent uses a given supplier for at rnost one of its projects: $SPQ \rightarrow V$. Proceeding as before, we have a lossless-join decornposition of Contracts into SDP and CSJDQV. Alternatively, we could begin by using the dependency $SPQ \rightarrow V$ to guide our decomposition, and replace Contracts with SPQV and CSJDPQ. We can then dec(Hnpose CSJDPQ, guided by $51D \rightarrow P$, to obtain SDP and CSJDQ.

We now have two alternative lossless-join decornpositions of Contracts into BCNF, neither of which is dependency-preserving. The first alternative is to

replace Contracts with the relations SDP and CSJDQV. The second alternative is to replace it \vith SPQV, SDP, and CSJDQ. The addition of CJP makes the second deC0111positioll (but not the first) dependency-preserving. Again, the cost of lnaintaining the three relations CJP, SPQV, and CSJDQ (versus just CSJDQV) Illay lead us to choose the first alternative. In this case, enforcing the given FDs becornes Inore expensive. We Illight consider Ilot enforcing thern, but we then risk a violation of the integrity of our data.

20.8.4 Vertical Partitioning of BCNF Relations

Suppose that we have decided to decornpose Contracts into SDP and CSJDQV. These scheruas are in BCNF, and there is no reason to decornpose thern further from a nonl1alization standpoint. However, suppose that the following queries are very frequent:

- Find the contracts held by supplier S.
- Find the contracts placed by departrnent D.

These queries rnight lead us to decompose CSJDQV into CS, CD, and CJQV. The decornposition is lossless, of course, and the two il11portant queries can be answered by exarnining 111uch slualler relations. Another reason to consider such a dec()l11position is concurrency control *hot spots*. If these queries are COIllIllon, and the rnost COIlunon updates involve changing the quantity of products (and the value) involved in contracts, the decoulposition inlproves perforrnance by reducing lock contention. Exclusive locks are now set rnostly on the CJQV table, and reads on CS and CD do not conflict with these locks.

Whenever we decornpose a relation, we have to consider which queries the decolnposition rnight adversely affect, especially if the only rnotivation for the decoInposition is iUlproved perforrnance. For exaruplc, if another illlportant query is to find the total value of contracts held by a supplier, it would involve a join of the decornposed relations CS and CJQV. In this situation, we rnight decide against the decolnposition.

20.8.5 Horizontal Decomposition

Thus far, we have essentially considered how to replace a relation with a collection of vertical decorupositions. Sornetilnes, it is worth considering whether to replace a relation with two relations that have the sa.Dle attributes as the original relation, each containin.g a subset of the tuples in tb.e original. Intuitively, this technique is useful \vhen different subsets of tuples are queried in very distinct ways.

For example, different rules lnay govern large contracts, \vhich are defined as contracts with values greater than 10,000. (Perhaps, such contracts have to be awarded through a bidding process.) This constraint could lead to a nUlnber of queries in which Contracts tuples are selected using a condition of the forIII $value > 10,000$. On8 way to approach this situation is to build a clustered B+ tree index on the *value* field of Contracts. Alternatively, we could replace Contracts with two relations called LargeContracts and SrnallContracts, with the obvious 11leaning. If this query is the only lllotivation for the index, horizontal decornposition offers all the benefits of the index without the overhead of index maintenance. This alternative is especially attractive if other irnportant queries on Contracts also require clustered indexes (on fields other than *val'ue).*

If we replace Contracts by two relations LargeContracts and SrnallContracts, we could r1l8sk this change by defining a view called Contracts:

```
CREATE VIEW Contracts(cid, supplierid, projectid, deptid, partid, qty, value)
        AS ((SELECT *
            FROM    LargeContracts)
            UNION
            (SELECT *
            FROM    SmallContracts))
```

However, any query that deals solely with LargeContracts should be expressed directly on LargeContracts and not on the view. Expressing the query on the view Contracts with the selection condition $value > 10,000$ is equivalent to expressing the query on LargeContracts but less efficient. This point is quite general: Although we can mask changes to the conceptual scherlla by adding view definitions, users concerned about perforrnance have to be aware of the change.

As another exanlple, if Contracts had an additional field *year* and queries typically dealt with the contracts in sorne one year, we rnight choose to pa,rtition Contracts by year. ()f course, queries that involved contracts fron1 rnore than one year rnight require us to pose queries against each of the decolllposed relations.

20.9 CHOICES IN TUNING QUERIES AND VIEWS

T'he first step in tuning a query is to understand the plan used by the DBMS to evaluate the query. 8ysten1s usually provide sorne facility for identifying the plan used to evaluate a query. Once we understand the plan selected by the systelIl, we can consider how to irnprove performance. We can consider a different choice of ilHlexes or perhaps co-clustering two relations for join queries,

guided by our understanding of the old plan and a better plan that we want theDBlVIS to use. The details are sinlilar to the initial design process.

One point'worth rnaking is that before creating new indexes we should consider \vhether rewriting the query achieves acceptable results with existing indexes. For example, consider the foll()\ving query with an OR connective:

```
SELECT  E.dno
FROM    Ernployees E
WHERE   E.hobby='Stalnps' OR E.age==10
```

If \ve have indexes on both *hobby* and *age,* we can use these indexes to retrieve the necessary tuples, but an optiInizer ruight fail to recognize this opportunity. The optinlizer rnight view the conditions in the WHERE clause as a whole as not rnatching either index, do a sequential scan of Ernployees, and apply the selections on-the-fly. Suppose we rewrite the query as the union of two queries, one with the clause *WHEREE.hobby= 'Starnps"* and the other with the clause WHERE *E.agc==10.* Now each query is answered efficiently with the aid of the indexes on *hobby* and *age.*

We should also consider rewriting the query to avoid sorne expensive operations. For exalnple, including DISTINCT in the SELECT clause leads to duplicate elirn-ination, which can be costly. rrhus, we should ornit DISTINCT whenever pos-sible. For exalnple, for a query on a single relation, we can ornit DISTINCT whenever either of the following conditions holds:

- We do not care about the presence of duplicates.
- rrhe attributes lllentioned in the SELECT clause include a candidate key for the relation.

SOlnetirnes a query with GROUP BY and HAVING can be replaced by a query without these clauses, thereby eliminating a sort operation. For example, coll-sider:

```
SELECT    MIN (E.age)
FROM      Employees E
GROUP BY  E.dno
HAVING    E.(lno=102
```

This quer:y is equivalent to

```
SELECT    MIN (E.age)
FROM      Erl1ployees E
WHERE     E.dno=102
```

Cornplex queries are often written in steps, using a temporary relation. We can usually rewrite such queries without the tClnporary relation to rnake thcrn run faster. Consider the following query for cornputi.ng the average salary of departrnents rnanaged by Robinson:

```
SELECT   *
INTO     Ternp
FROM     ErnployeesE, Depa:rtruents D
WHERE    E.dno=D.dno AND D.rngrnanle='Robinson'

SELECT   T.dno, AVG (T.sal)
FROM     T'clnp T
GROUP BY T.dno
```

This query can be rewritten as

```
SELECT   E.dno, AVG (E.sal)
FROM     Elnployees E, Departments D
WHERE    E.dno==D.dno AND D.rngrnarne=='llobinson'
GROUP BY E.dno
```

The rewritten query does not 111aterialize the interrnediate relation ,Ternp and is therefore likely to be faster. In fact, the optimizer may even find a very efficient index-only plan that never retrieves Ernployees tuples if there is a cornposite B+· tree index on *(d'no, sal).* This exanlple illustrates a general observation: *By rewriting queries to avoid unnecessary temporaries, we not only avoid creating the ternporary relations, we also open up rnore optimization possibilities for the optim,izer to el;plore.*

In SCHne situations, ho\vever, if the optirnizer is unable to find a good plan for a cornplex query (typically a nested query with correlation), it rnay be worthwhile to rewrite the query using tenlporary relations to guide the optirnizer toward a good plan.

In fact, nested queries are a conunon source of inefficiency because luany optirnizers deal poorly with theIn, as discussed in Section 15.5.v'Vllenever possible, it is better to rewrite a nested query \vithout nesting and a correlated query without correlation. As already noted, a good reforrIlulation of the query rnay require us to introduce new, ternporary relations, and techniques to do so systenlatically (ideally, to be done by the optirnizer) have been \videly studied. ()ften tllough, it is possible to re\vrite nested queries ,vithout nesting or the use of ternpora,ry relations, as illustrated in Section 15.5.

20.10 IMPACT OF CONCURRENCY

In a system with IIlaIlY concurrent users, several additional points IllUSt be considered. Transactions obtain *locks* on the pages they access, and other transactions Ina)' be blocked waiting for locks on objects they wish to access.

We observed in Section 16.5 that blocking delays 111ust be IniniInized for good perforrnance and identified two specific ways to reduce blocking:

- Reducing the time that transactions hold locks.
- R,edllcing hot spots.

We now discuss techniques for achieving these goals.

20.10.1 Reducing Lock Durations

Delay Lock Requests: Tune transactions by writing to local prograrn variables and deferring changes to the database until the end of the transaction. This delays the acquisition of the corresponding locks and reduces the time the locks are held.

Make Transactions Faster: The sooner a transaction c01npletes, the sooner its locks are released. We have already discussed several ways to speed up queries and updates (e.g., tUllillg indexes, rewriting queries). In addition, a careful partitioning of the tuples in a relation and its associated indexes across a collection of disks can significantly irnprove concurrent access. For exarnple, if we have the relation on one disk and an index on another, accesses to the index can proceed without interfering with accesses to the relation, at least at the level of disk reads.

Replace Long Transactions by Short Ones: SometiInes, just too ruuch work is done within a transaction, and it takes a long tirne and holds locks a long tirne. Consider rewriting the transaction as two or Inore smaller transactions; holdable cursors (see Section 6.1.2) can be helpful in doing this. The advantage is that each new transaction cornpletes quicker and releases locks sooner. The disadvantage is that the original list of operations is no longer executed atomically, and the application code Illust deal with situations in which one or rnore of the new transactions fail.

Build a Warehouse: CC)lnplex queries can hold shared locks for a long tirne. ()ften, however, these queries involve statistical analysis of business trends and it is a,cceptable to run theln on a copy of the data that is a little out of date. rrhis led to the popularity of *data warehouses*, which are databases that cornplcnlcnt

the operational database by rnaintaining a copy of data used in cornplex queries (Chapter 25). Running these queries against the warehouse relieves the burden of long-running queries froln the operational database.

Consider a Lower Isolation Level: In rnany situations, such as queries generating aggregate infonnation or statistical sununaries, we can use a lower SQL isolation level such as REPEATABLE READ or READ COMMITTED (Section 16.6). Lo\ver isolation levels incur lower locking overheads, a,nel the application progrannner rllust make good design trade-offs.

20.10.2 Reducing Hot Spots

Delay Operations on Hot Spots: We already discussed the value of delaying lock requests. Obviously, this is especially irnportant for requests involving frequently used objects.

Optimize Access Patterns: The *patteTn* of updates to a relation can also be significant. For exanlple, if tuples are inserted into the Ernployees relation in *eid* order and we have a B+ tree index on *eid,* each insert goes to the last leaf page of the B+ tree. This leads to hot spots along the path froIn the root to the rightrnost leaf page. Such considerations nlay lead us to choose a hash index over a B+- tree index or to index on a different field. Note that this pattern of access leads to poor perforrnance for ISAM indexes as well, since the last leaf page beCOlIles a hot spot. rrhis is not a problcln for hash indexes because the hashing process randornizes the bucket into which a record is inserted.

Partition Operations on Hot Spots: Consider a data entry transaction that appends new records to a file (e.g., inserts into a table stored as a heap file). Instead of appending records one-per-transaction and obtaining a lock on the last page for each record, we can replace the transaction by several other transactions, each of which writes records to a local file and periodically appends a batch of records to the rnain file. While we do rnore work overall, this reduces the lock contention on the last page of the original file.

As a further illustration of partitioning, suppose we track the nU111ber of records inserted in a counter. Instead of updating this counter once per record, the preceding approach results in updating several counters and periodically updating the main counter. rrhis idea can be adapted to many uses of counters, \vith varying degrees of effort. For exaInple, consider a counter that tracks the Ilurnber of reservations, with the rule that a new reservation is allowed only if the counter is below a, rnaxiullun value. We can replace this by three counters, each \vith one-third the origina11naxirIlurn threshold, and three transactions that use these counters rather than the original. We obtain greater concurrency, but

have to deal with the case where one of the counters is at the maximum value but SOHIe other counter can still be incrcrnented. Thus, the price of greater concurrency is increased cornplexity in the logic of the application code.

Choice of Index: If a relation is updated frequently, B+ tree indexes can becolne a concurrency control bottleneck, because all accesses through the index HUlst go through the root. Thus, the root and index pages just below it can bec()lne hot spots. If the DBMS uses specialized locking protocols for tree indexes, and in particular, sets finc-granularity locks, this problenl is greatly alleviated. l\Ilany current systeuls use such techniques.

Nonetheless, this consideration lllay lead us to choose an ISAM index in SOlllle situations. Because the index levels of an ISAM index are static, we need not obtain locks on these pages; only the leaf pages need to be locked. An ISAl\!l index rnay be preferable to a B+ tree index, for exalllple, if frequent updates occur but we expect the relative distribution of records and the nUlnber (and size) of records with a given range of search key values to stay approxirnately the saIne. In this case the ISAM index offers a lower locking overhead (and reduced contention for locks), and the distribution of records is such that few overflow pages are created.

I-Iashed indexes do not create such a concurrency bottleneck, unless the data distribution is very skewed and lnany data itenlS are concentrated in a few buckets. In this case, the directory entries for these buckets can beccnne a hot spot.

20.11 CASE STUDY: THE INTERNET SHOP

Revisiting our running case study, I)BDudes considers the expected workload for the B&N 1)00kstore. rrhe owner of the bookstore expects rnost of his CIIS-torners to search for books by ISBN nUluber before placing an order. Placing an order involves inserting one record into the ()rders table and inserting one or lllore records into the Orderlists relation. If a sufficient nurnber of books is available, a, shiprnent is prepared and a value for the *ship_date* in the Orderlists relation is set. In addition, the available quantities of books in stock changes all the tirne, since orders are placed that; decrease the quantity available and new books arrive frorn suppliers and increase the quantity available.

The DBDudes tearn begins by considering searches for books by ISBN'. Since *isbn* is a key, (l,n equality query on *isbn* returns at rnost one record. rrhereforc, to speed up queries frolll Cllstolllers who look for books with a given ISBN, I)BIJudes decides to build an unclustered hash index on *isbn*.

Next, it considers updates to book quantities. To update the *qty_in_stock* value for a book, we IIIUSt first search for the book by ISBN; the index on *isbn* speeds this up. Since the *qty_in_stock* value for a book is updated quite frequently, DBDudes also considers partitioning the Books relation vertically into the fo1-1c)\ving two relations:

BooksQty(*isbn*, *qty*)
BookRest(*isbn*, *title*, *author*, *price*, *year_published*)

Unfortunately, this vertical partitioning slows do\vn another very popular query: Equality search on ISBN to retrieve all infonnation about a book now requires a join between BooksQty and BooksH,est. So DBDudes decides not to vertically partition Books.

DBDudcs thinks it is likely that custonlers will also want to search for books by title and by author, and decides to add unclustered hash indexes on *title* and author-these indexes are inexpensive to rnaintain because the set of books is rarely changed even though the quantity in stock for a book changes often.

Next, DBDudes considers the Custorners relation. A custorner is first identified by the unique custorner identifaction nurnber. So the rnost COlnrnon queries on Custorners are equality queries involving the custolner identification nurnber, and DBDudes decides to build a clustered hash index on *cid* to achieve maxirnum speed for this query.

l\!Ioving on to the Orders relation, DBDudes sees that it is involved in two queries: insertion of new orders and retrieval of existing orders. Both queries involve the *ordernum* attribute as search key and so DBDudes decides to huild an index on it. What type of index should this be—a 13+ tree or a hash index? Since order nurnbers are assigned sequentially and correspond to the order date, sorting by *ordernum* effectively sorts by order date as well. So DBDudes decides to build a clustered B+ tree index on *ordernum*. A.lthough the operational requirernents rncntioned until no\v favor neither a 13+ tree nor a hash index, B&N\vill probably want to rnonitor daily a,ctivities and the clustered 13+ tree is a better choice for such range queries. ()f course, this 1118ans that retrieving all orders for a given custorner could be expensive for custolllers with InallY orders, since clustering by *ordernum* precludes clustering by other attributes, SllCh as *cicio*

The ()rderlists rela,tion involves lnostly insertions, with an occasional update of a shiprnent date or a query to list all cOlnponents of a given order. If Orderlists is kept sorted on *ordernum*, all insertions are appends at the end of the relation and thus very efficient. A clustered 13+ tree index on *ordernum* maintains this sort order and also speeds up retrieval of aU iterns for a given order. To update

a shiprnent date, we need to search for a tuple by *ordernum* and *isbn*. The index on *ordernum* helps here as well. Although an index on ⟨*ordernum*, *isbn*⟩ would be better for this purpose, insertions would not be as efficient as with an index on just *ordernum*; DBDudes therefore decides to index ()rderlists on just *ordernum*.

20.11.1 Tuning the Database

Several rnonths after the launch of the B&N site, DBDudes is called in and told that custorner enquiries about pending orders are being processed very slowly. B&N has becorne very successful, and the Orders and Orderlists tables have grown huge.

l'hinking further about the design, DBDudes realizes that there are two types of orders: *completed orders,* for which all books have already shipped, and *partially co'mpleted orders*, for which sorne books are yet to be shipped. l\Ilost custorIler requests to look up an order involve partially corIlpleted orders, which are a sInall fraction of all orders. DBDudes therefore decides to horizontally partition both the Orders table and the Orderlists table by *ordernu'Tn*. This results in four new relations: NewOrders, OldOrders, NewOrderlists, and OldOrderlists.

An order and its cornponents are always in exactly one pair of relations·····.·--and we can deterrIline which pair, old or new, by a sinlple check on *ordernum*—and queries involving that order can always be evaluated using only the relevant relations. SCHIle queries are now slower, such as those asking for all of a custoruer's orders, since they require us to search two sets of relations. lIowever, these queries are infrequent and their perforrnance is acceptable.

20.12 DBMS BENCHMARKING

Thus far, we considered how to irnprove the design of a database to obtain better perforrnance. 1\S the database grows, however; the underlying DBMS rnay no longer be able to provide adequate perforrnance, even with the best possible design, and we have to consider upgrading our systcrn, typically by buying faster harchva,re and additional rnernory. We IIlay also consider rnigrating our database to (1, new DBIVIS.

When evaluating DBMS products, perforrnal1ce is an iUlportant consideration. ADBIVIS is a cornplex piece of software, and different vendors rnay target their systerns toward different market segrnents by putting rnore effort into optirnizirlg certa,in parts of the systern or choosing different systern designs. For example, sorne systcrIls are designed to run cornplex queries efficiently, while others are designed to run Inany sirnple transactions per second. Within

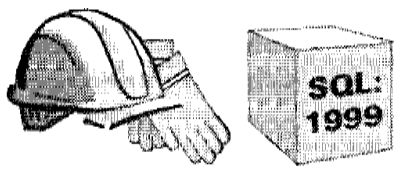
each category of systcrIls, there are many cornpeting products. To assist users in choosing a DBMS that is 'well suited to their needs, several performance benchmarks have been developed. These include benchrnarks for measuring the perforlnance of a certain class of applications (e.g., the TPC benclnnarks) and benchrnarks for measuring how well a DBIVlS perfOrII1S various operations (e.g., the \Visconsin benchrnark).

Benchmarks should be portable, easy to understand, and scale naturally to larger problenl instances. They should II1eaSUre *peak performance* (e.g., *transactions per second,* or *ips)* as well as *pTice/perforrnance ratios* (e.g., *$/tps)* for typical workloads in a given application donlain. The Transaction Processing Council (TPC) was created to define benchlnarks for transaction processing and database systerns. Other well-known benchlnarks have been proposed by acadelnic researchers and industry organizations. Benchrnarks that are proprietary to a given vendor are not very useful for cornparing different systerns (although they rnay be useful in deterrnining how well a given systern would handle a particular workload).

20.12.1 Well-Known DBMS Benchmarks

Online Transaction Processing Benchmarks: The TPC-A and TPC-B benchrnarks constitute the standard definitions of the *ips* and $/*tps* measures. TPC-A rneasures the perfonnance and price of a computer network in addition to the DBMS, whereas the TPC-B benclnnark considers the DBMS by itself. These bencln11arks involve a sirnple transaction that updates three data records, frolIl three different tables, and appends a record to a fourth table. A 11urnber of details (e.g., transaction arrival distribution, interconnect rnethod, systern properties) are rigorously specified, ensuring that results for different systenls can be rneaningfully cOI11pared. The T'PC-C benchmark is a l110re cornplex suite of transactional tasks than TPC-A and TPC-B. It rnodels a warehouse that tracks iterns supplied to custorners and involves five types of transactions. Each TPC-C transaction is rnuch rIlore expensive than a 1'PC-A or TPC-B transaction, a,nel TPC-C exercises a rnuch ,videI' range of systern capabilities, such as use of secondary indexes and transaction aborts. It has Inore or less cOlnpletely replaced TPC-A and rrpC-B as the standard transaction processing bencillnark.

Query Benchmarks: The Wisconsin l)cnchrnark is \videly used for 1neasnring the perforrnance of sirnple relational queries. The Set Query benclunark Hleasures the perforrnance of a suite of rJlore cornplex queries, and the $AS^3A.P$ l)enchrnark measures the perfonnance of (1, Inixed workload of transactions, relational queries, and utility fUllctions. The rrpC-I) benchn.lark is a suite of cornplex SQL queries intended to be representative of the (Incision-support ap-



21

SECURITY AND AUTHORIZATION

- What are the rnain security considerations in designing a database application?
- What IIlechanisms does a DBMS provide to control a user's access to data?
- What is discretionary access control and how is it supported in SQL?
- What are the weaknesses of discretionary access control? How are these addressed in lnandatory access control?
- What are covert channels and how do they cornpromise lnandatory access control?
- What lTIUst the DBA do to ensure security?
- What is the added security threat when a database is accessed re-rnotely?
- What is the role of encryption in ensuring secure access? How is it used for certifying servers and creating digital sig11atures?
- Key concepts: security, integrity, availability; discretionary access control, privileges, GRANT, REVOKE; rna.ndatory access control, objects, subjects, security classes, rnultilevel tables, polyinstantiation; covert channels, DoD security levels; statistical databases, inferring secure information; authentication for reIllote access, securing servers, digital signatures; encyption, public-key encryption. -

I know that's a secret, for it's whispered everywhere.

William Congreve

The data stored in a DBMS is often vital to the business interests of the organization and is regarded as a corporate asset. In addition to protecting the intrinsic value of the data, corporations rnust consider O\vays to ensure privacy and control access to data that must not be revealed to certain groups of users for various reasons.

In this chapter, we discuss the concepts underlying access control and security in a DBMS. After introducing database security issues in Section 21.1,we consider two distinct approaches, called *discretionary* and *mandatory*, to specifying and lTlanaging access controls. An access control Inechanism is a way to control the data accessible by a given user. After introducing access controls in Section 21.2, we cover discretionary access control,which is supported in SQL, in Section 21.3.vVe briefly cover n1andatory access control, which is not supported in SQL, in Section 21.4.

In Section 21.6, we discuss SOIne additional aspects of database security, such as security in a statistical database and the role of the database adrninistrator. We then consider SOlne of the unique challenges in supporting secure access to a DBMS over the Internet, which is a central problern in e-COlllInerce and other Internet database applications, in Section 21.5. We conclude this chapter with a discussion of security aspects of the Barns and Nobble case study in Section 21.7.

21.1 INTRODUCTION TO DATABASE SECURITY

There are three rnain objectives when designing a secure database application:

1. Secrecy: InfoI'rnation should not be disclosed to unauthorized users. For exarnple, a student should not be allowed to exarnine other students' grades.
2. Integrity: ()nly authorized users should be allowed to Hlodify data. For example, students 1Ilay be allowed to see their grades, yet not allowed (obviously) to rnodify thern.
3. Availability: Authorized users should not be denied access. For example, an instructor who wishes to change a grade should be allowed to do so.

To achieve these objectives, a clear and consistent security policy should be developed to describe what security measures rnust be enforced. In particular, we rnu8t detennine what part of the data is to be protected and which users get access to which portions of the data. Next, the security mechanisrns of the underlying I)B:JVIS and operating systenl, as well as externaJ mechanisms,

such as securing access to buildings, Illust be utilized to enforce the policy. We emphasize that security rneasures IIlust l)e taken at several levels.

Security leaks in the OS or network connections can cirCUlnvent database security rnechanisrns. For example, such leaks could allow an intruder to log on as the database acbninistrator, 'with all the attendant I)BlVIS access rights. Hurnan factors are another source of security leaks. :For exarnple, a user IHay choose a password that is easy to guess, or a user who is authorized to see sensitive data may luisuse it. Such errors account for a large percentage of security breaches. \Ve do not discuss these aspects of security despite their irllportance because they are not specific to database rnanagerllent systelIls; our IIlain focus is on database access control rllechanisrns to support a security policy.

We observe that vie\vs are a valuable tool in enforcing security policies. The view rnechanisrll can be used to create a 'window' 011 a collection of data that is appropriate for SOllle group of users. 'Views allow us to liUlit access to sensitive data by providing access to a restricted version (defined through a view) of that data, rather than to the data itself.

We use the following schemas in our exaurples:

Sailors(*sid:* integer, *snarne:* string, *rating:* integer, *age:* real)
Boats(*bid:* integer, *bnarne:* string, *color:* string)
Reserves(*sid:* integer, *bid:* integer, *day:* dates)

Increasingly, as database systcrlls becorne the backbone of e-COlluncrce applications requests originate over the Internet. This rnakes it irnportant to be able to authenticate a user to the database systern. A.fter all, enforcing a security policy that allows user Sarn to read a table and Ehner to write the table is not of l11uch use if Sam can rnasquerade as Ebner. COllversely, we Inus!; be able to assure users that they a,re COIlluIlunicating \vith a legitilnate systern (e.g., the real Arnazoll.col11 server, and not a spurious application intended to steal sensitive inforrnation such as a credit card nurl11>cr). vVhile the details of authentication are outside the scope of our coverage, we discuss the role of authentication and the l)Hsic ideas involved in Section 21.5, after covering database access control rnechanisrIls.

21.2 ACCESS CONTROL

A database for an enterprise contains a great deal of inforrnation and usually has several groups of users. 1\IJost users need to access onl,y a sruall pa;rt of the database to carry out their tasks. J\l1owing users unrestricted access to all the

data can be undesirable, and a DBMS should provide rnechanisHls to control access to data.

A DBMS offers two rnain approaches to access control. Discretionary access control is based on the concept of access rights, or privileges, and rnechanisrllS for giving users such privileges. A privilege allows a user to access Borne data object in a certain IllHnIler (e.g., to read or 11lOdify). A user who creates a database object such as a table or a view autornatically gets all applicable privileges on that object. The D.BMS subsequently keeps track of how these privileges are granted to other users, and possibly revoked, and ensures that at all tirnes only users with the necessary privileges can access all object. SQL supports discretionary access control through the GRANT and REVOKE conunands. The GRANT cOllnnand gives privileges to users, and the REVOKE cornrnand takes away privileges. We discuss discretionary access control in Section 21.3.

Discretionary access control rnechanisrns, while generally effective, have certain weaknesses. In particular, a devious unauthorized user can trick an authorized user into disclosing sensitive data. Mandatory access control is based on systemwide policies that cannot be changed by individual users. In this approach each database object is assigned a *security class,* each user is assigned *clearance* for a security class, and rules are irnposed on reading and writing of database objects by users. The DBMS deterrnines whether a given user can read or write a given object based on certain rules that involve the security level of the object and the clearance of the user. These rules seek to ensure that sensitive data can never be 'passed on' to a user without the necessary clearance. 'rhe SQL standard does not include any support for rnandatory access control. 'We discuss rnandatory access control in Section 21.4.

21.3 DISCRETIONARY ACCESS CONTROL

SQL supports discretionary access control through the GRANT and REVOKE corn-rnands. The GRANT cornrnand gives users privileges to base tables and views. 'rhe syntax of this command is as follows:

```
GRANT privileges ON object TO users [WITH GRANT OPTION]
```

For our purposes object is either a base table or a view. SQL recognizes certain other kinds of objects, but we do not discuss thcrn. Several privileges can be specified, including these:

- SELECT: The right to access (read) all colurnns of the table specified as the object, *including columns added later* through ALTER TABLE cornrnands.

- INSERT(*column-name*): The right to insert rows with (non-*null* or non-default) values in the naTned cohnnn of the table named as object. If this right is to be granted with respect to all cohunns, including coluulns that rnight be added later, we can sirnply usc INSERT. The privileges UPDATE(*column-name*) and UPDATE are sirnilar.
- DELETE: The right to delete rows frorn the table narned as object.
- REFERENCES (*col'Urnn-namJe):* The right to define foreign keys (in other tables) that refer to the specified cohnnn of the table object. REFERENCES without a colurnn naUIe specified denotes this right with respect to all colurnns, including any that are added later.

If a user has a privilege with the grant option, he or she can pass it to another user (with or without the grant option) by using the GRANT conunand. A user who creates a base table autolnatically has all applicable privileges on it, along with the right to grant these privileges to other users. A user who creates a view has precisely those privileges on the view that he or she has on *everyone* of the views or base tables used to define the view. The user creating the view Inust have the SELECT privilege on each underlying table, of course, and so is always granted the SELECT privilege on the view. The creator of the view has the SELECT privilege with the grant option only if he or she has the SELECT privilege with the grant option on every underlying table. In addition, if the view is updatable and the user holds INSERT, DELETE, or UPDATE privileges (with or without the grant option) on the (single) underlying table, the user autornatically gets the same privileges on the view.

()nly the owner of a scherna can execute the data definition statcrnents CREATE, ALTER, and DROP on that schcrna. The right to execute these staternents cannot be granted or revoked.

In conjullction with the GRANT and REVOKE cOl1llnands, views are an irnportant cornponent of the security rnechanisrns provided by a relational DBMS. By defining views on the base tables, we can present needed inforrnation to a user "while *hiding* other inforrnation that the user should not be given access to. For example, consider the following view definition:

```
CREATE VIEW }\ctiveSajlors (name, age, day)
        AS SELECT S.snarne, S.age, R.day
           FROM   Sailors S, Reserves R
           WHERE  S.sid = R.sid AND S.rating > 6
```

A user who can access ActiveSailors but not Sailors or Reserves kno\vs the names of sailors who have reservations but cannot find out the *bids* of boats reserved by a given sailor.

Role-Based Authorization in SQL: Privileges are assigned to users (authorization IDs, to be precise) in SQL-92. In the real world, privileges are often associated with a user's job or *role* within the organizat;ion. Many DBMSs have long supported the concept of a role and allowed privileges to be assigned to roles. Roles can then he granted to users and other roles. (Of courses, privileges can also be granted directly to users.) l'he SQL:1999 standard includes support for roles. Roles can be created and destroyed using the CREATE ROLE and DROP ROLE eornrnands. Users can be granted roles (optionally, with the ability to pass the role on to others). The standard GRANT and REVOKE connnands can assign privileges to (and revoke from) roles or authorization IDs.
What is the benefit of including a feature that Inany systerns already support? 'This ensures that, over tiIne, all vendors who comply with the standard support this feature. 'rhus, users can use the feature without worrying about portability of their application across DBMSs.

Privileges are assigned in SQL to authorization IDs, which can denote a single user or a group of users; a user lllUSt specify an authorization ID and, in Inany systerns, a corresponding *password* before the DBMS accepts any C0111-rnancls from hirn or her. So, technically, *Joe*, *Michael*, and so on are authorization IDs rather than user nan1es in the following exalllpies.

Suppose that user Joe has created the tables Boats, Reserves, and Sailors. Senne exarnples of the GRANT cOllunand that Joe can now execute fo11o\v:

```
GRANT INSERT, DELETE ON Reserves TO Yuppy WITH GRANT OPTION
GRANT SELECT ON Reserves TO Michael
GRANT SELECT ON Sailors TO Michael WITH GRANT OPTION
GRANT UPDATE (rating) ON Sailors TO Leah
GRANT REFERENCES (bid) ON Boats TO Bill
```

Yuppy can insert or delete Reserves rows and authorize SOlneone else to do the sarne. I\1ichael can execute SELECT queries on Sailors and H,eserves, and 118 can pass this privilege to others for Sailors but not for R,eserves. With the SELECT privilege, Michael can create a view that accesses the Sailors and Reserves tables (for example, the ActiveSailors vic\v), but he cannot grant SELECT on ActiveSailors to others.

()rI the other hand, suppose that Michael creates the foUo\ving view:

```
CREATE VIEWYoungSailors (sicl, age, rating)
        AS SELECT S.sicl, S.age, S.rating
```

```
FROM    Sailors S
WHERE   S.age < 18
```

The only underlying table is Sailors, for which Michael has SELECT with the grant option. He therefore has SELECT with the grant option on YoungSailors and can pass on the SELECT privilege on YoungSailors to Eric and Guppy:

```
GRANT SELECT ON YoungSailors TO Eric, Guppy
```

Eric and Guppy can now execute SELECT queries on the view YoungSailors-note, however, that Eric and Guppy do *not* have the right to execute SELECT queries directly on the underlying Sailors table.

Michael can also define constraints based on the inforrnation in the Sailors and Reserves tables. For exarnple, Michael can define the following table, which has an associated table constraint:

```
CREATE TABLE Sneaky (lnaxrating   INTEGER,
                     CHECK (maxrating >=
                            ( SELECT MAX (S.rating)
                              FROM    Sailors S )))
```

By repeatedly inserting rows with gradually increasing *rnaxrating* values into the Sneaky table until an insertion finally succeeds, lVIichael can find out the highest *rating* value in the Sailors table. This exarnple illustrates why SQL requires the creator of a table constraint that refers to Sailors to possess the SELECT privilege on Sailors.

Returning to the privileges granted by Joe, Leah can update only the *rating* colulnn of Sailors rows. She can execute the following cornmand, which sets all ratings to 8:

```
UPDATE  Sailors S
SET     S.rating = 8
```

However, she cannot execute the same cOllunand if the SET clause is changed to be SET *S.age* = 25, because she is not allowed to update the *age* field. A rnoro subtle point is illustrated by the following cOIrllnand, which decrelnents the rating of all 'sailors:

```
UPDATE  Sailors S
SET     S.ratillg = S.rating-l
```

Leah cannot execute this cOlInnand because it requires the SELECT privilege 011 the *S.rating* colurnn anei Leah does not have this privilege.

Bill can refer to the *bid* colurnn of Boats as a foreign key in another table. For example, Bill can create the Reserves table through the following cOlnnland:

```
CREATE TABLE R"eserves (sid     INTEGER,
                        bid     INTEGER,
                        day     DATE,
                        PRIMARY KEY (bid, day),
                        FOREIGN KEY (sid) REFERENCES Sailors ),
                        FOREIGN KEY (bid) REFERENCES Boats)
```

If Bill did not have the REFERENCES privilege on the *bid* coh1111n of Boats, he would not be able to execute this CREATE staternent because the FOREIGN KEY clause requires this privilege. (A sirnilar point holds with respect to the foreign key reference to Sailors.)

Specifying just the INSERT privilege (sirnilarly, REFERENCES and other privileges) in a GRANT conlmand is not the sarne as specifying SELECT(*colurnn-name)* for each column currently in the table. Consider the following command over the Sailors table, which has cohllnns *sid*, *snarne*, *rating*, and *age:*

```
GRANT INSERT ON Sailors TO J\!Iichael
```

Suppose that this conunand is executed and then a colurnn is added to the Sailors table (by executing an ALTER TABLE cOlIllnand). Note that Michael has the INSERT privilege with respect to the newly added colurnn. If we had executed the following GRANT cornrnand, instead of the previous one, Michael would not have the INSERT privilege on the new cohllnn:

```
GRANT   INSERT ON Sailors(sid), Sailors(sna1ne), Sailors(rating),
        Sailors(age), TO J\!Iichael
```

There is a cornplernentary corl1rnand to GRANT that allows the withdrawal of privileges. The syntax of the REVOKE cOllunand is as follows:

```
REVOKE [GRANT OPTION FOR ] privileges
       ON object FROM users {RESTRICT I CASCADE }
```

The cOIlnnand CH,n be used to revoke either a privilege or just the grant option on a privilege (by using the optional GRANT OPTION FOR clause). One of the two alternatives, RESTRICT or CASCADE, HUlst be specified; we see 'what this choice IneaI1S shortly.

The intuition behind the GRANT command is clear: rrhe creator of a base table or a view is given all the appropriate privileges \vith respect to it and is allowed

to pass these privileges—including the right to pass along a privilege—to other users. The REVOKE comuland is, as expected, intended to achieve the reverse: A user who has granted a privilege to another user rnay change his or her lnincI and want to withdraw the gra,nted privilege. The intuition behind exactly 'what effect a REVOKE cornrnand has is conlplicated by the fact that a user Inay be granted the sarne privilege rnultiple tilnes, possibly by different users.

\Vhen a user executes a REVOKE cornmand with the CASCADE keyword, the effect is to \vithdraw the named privileges or grant option froIn all users who currently hold these privileges *solely* through a GRANT cOllunand that was previously executed by the sallIe user who is now executing the REVOKE cOl1nnand. If these users received the privileges with the grant option and passed it along, those recipients in turn lose their privileges as a consequence of the REVOKE cOIurnand, unless they received these privileges through an additional GRANT comIuand.

We illustrate the REVOKE cOllllnand through several examples. First, consider what happens after the following sequence of eornmands, where Joe is the creator of Sailors.

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION   (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION   (executed by Art)
REVOKE SELECT ON Sailors FROM Art CASCADE          (executed by Joe)
```

Art loses the SELECT privilege on Sailors, of course. Then Bob, who received this privilege from Art, and only Art, also loses this privilege. Bob's privilege is said to be abandoned when the privilege froIn which it was derived (Art's SELECT privilege with grant option, in this exarnple) is revoked. When the CASCADE keyword is specified, all abandoned privileges are also revoked (possibly causing privileges held by other users to becOlne abandoned and thereby revoked recursively). If the RESTRICT keyword is specified in the REVOKE corllmand, the cornrnand is rejected if revoking the privileges *just* frorn the users specified in the cOIllluand would result in other privileges becorning abandoned.

Consider the following sequence, as another exarnple:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION   (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION   (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION   (executed by Art)
REVOKE SELECT ON Sailors FROM Art CASCADE          (executed by Joe)
```

As before, Art loses the SELECT privilege on Sailors. But what about Bob? Bob received this privilege fronl Art, but he also received it independently

(coincidentally, directly fro111 Joe). So Bob retains this privilege. Consider a third example:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION    (executed by Joe)
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION    (executed by Joe)
REVOKE SELECT ON Sailors FROM Art CASCADE           (executed by Joe)
```

Since Joe granted the privilege to Art twice and only revoked it once, does Art get to keep the privilege? As per the SQL standard, no. Even if Joe absentmindedly granted the saIne privilege to Art several tirnes, he can revoke it with a single REVOKE cOIlunand.

It is possible to revoke just the grant option on a, privilege:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION    (executed by Joe)
REVOKE GRANT OPTION FOR SELECT ON Sailors
      FROM Art CASCADE                              (executed by Joe)
```

This cOlnmand would leave Art with the SELECT privilege on Sailors, but Art no longer has the grant option on this privilege and therefore cannot pass it on to other users.

These exarnples bring out the intuition behind the REVOKE cOilllland, and they highlight the cOlllplex interaction between GRANT and REVOKE cOlnnlands. When a GRANT is executed, a privilege descriptor is added to a table of such descriptors Inaintained by the DElVIS. The privilege descriptor specifies the following: the *grantor* of the privilege, the *grantee* who receives the privilege, the *granted privilege* (including the narne of the object involved), and whether the grant option is included. When a user creates a table or view and 'autornatically' gets certain privileges, a privilege descriptor with *system,* as the grantor is entered into this table.

rrhe effect of a series of GRANT cornrnands can be described in terrns of an authorization graph in which the nodes are users—technically, they are authorization IDs·—and the arcs indicate .how privileges are passed. There is an arc fron1 (the node for) user 1. to user 2 if user 1. executed a GRANT cOIIunand giving a privilege to user 2; the arc is labeled with the descriptor for the GRANT cOIlllnand. A GRANT cOIlnnand has no effect if the saIne privileges ha.ve already been granted to the same grantee by th.e sarne grantor. The following sequence of commands illustrates the sernantics of GRANT and REVOKE connnands when there is a *cycle* in the authorization graph:

```
GRANT SELECT ON Sailors TO .Art WITH GRANT OPTION   (executed by Joe)
GRANT SELECT ON Sa.ilors TO Bob WITH GRANT OPTION   (executed by Art)
```

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION   (executed by Bob)
GRANT SELECT ON Sailors TO Cal WITH GRANT OPTION   (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION   (executed by Cal)
REVOKE SELECT ON Sailors FROM Art CASCADE          (executed by Joe)
```

The authorization graph for this exarnple is shown in Figure 21.1. Note that we indicate how Joe, the creator of Sailors, acquired the SELECT privilege fror11 the DBMS by introdtIcing a *System* node and drawing an arc froIn this node to Joe's node.

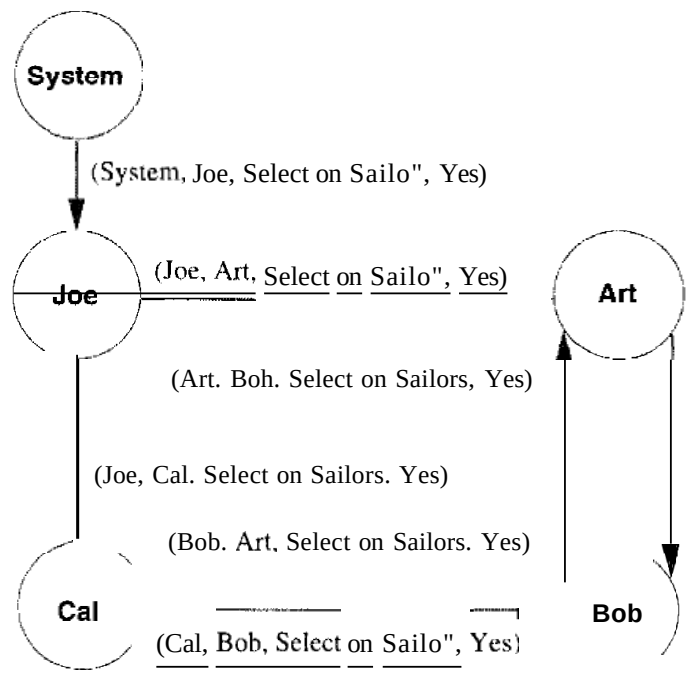


Figure 21.1 Example Authorization Graph

As the graph dearly indicates, Bob's grant to Art and Art's grant to Bob (of the same privilege) creates a cycle. Bob is subsequently given the saIne privilege by Cal, who received it independently froIn Joe. At this point Joe decides to revoke the privilege he granted Art.

Let us trace the effect of this revocation. The arc [raIn Joe to Art is removed because it corresponds to the granting action that is revoked. All rernaining nodes have the following property: *1f node N has an outgoing arc labeled with a privilege, there is a path fTorn the System node to 'node N in 'which each aTC label contains the same privilege plus the grant opt'ion.* That is, any rernaining granting action is justified by a privilege received (directly or indirectly) frorn the Systern. The execution of Joe's REVOKE conllnand therefore stops at this POiIlt,\vith everyone continuing to hold the SELECT privilege on Sailors.

rrhis result may seenl nnintuitive because Art continues to have the privilege only because he received it fr0111 Bob, and at the time that Bob granted the privilege to Art, he had received it only frorn Art. Although Bob acquired the privilege through Cal subsequently, should we not undo the effect of his grant

to Art when executing Joe's REVOKE command? 'rhe effect of the grant frorn Bob to Art is *not* undone in SQL. In effect. if a user acquires a privilege rnultiple tilnes frolIl different grantors, SQL treats each of these grants to the user as having occurred *befoTe* that user passed on the privilege to other users. This implementation of REVOKE is convenient in 111any reaJ-\world situations. .For exanlple, if a lIl.anager is fired after passing on sorne privileges to subordinates (who lnay in turn have passed the privileges to others), we can ensure that only the rnanager's privileges are rernoved by first redoing all of the Illanager's granting actions and then revoking his or her privileges. That is, we need not recursively redo the subordinates' granting actions.

To return to the saga of Joe and his friends, let us suppose that Joe decides to revoke Cal's SELECT privilege as well. Clearly, the arc frorn Joe to Cal corresponding to the grant of this privilege is rerIloved. The arc frorH Cal to Bob is reilloved as well, since there is no longer a path fronl SystelIl to Cal that gives Cal the right to pass the SELECT privilege on Sailors to Bob. The authorization graph at this interrnediate point is shown in Figure 21.2.

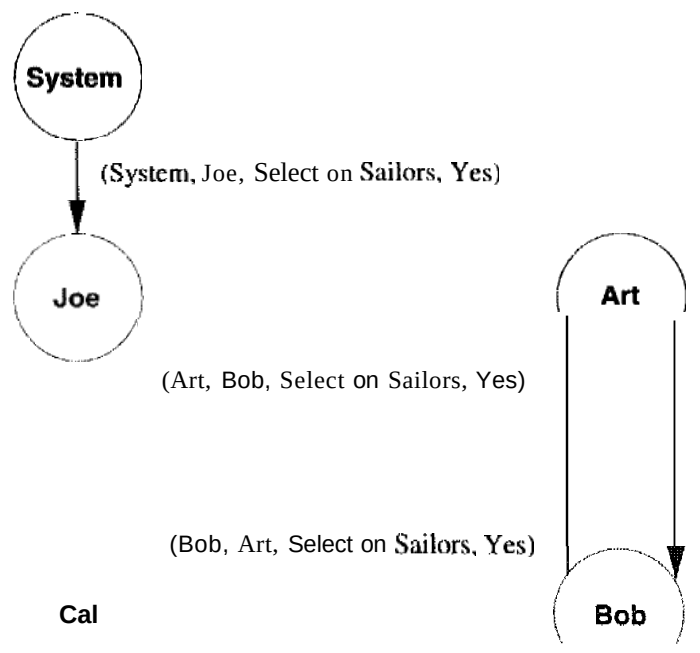


Figure 21.2 Example Authorization Graph during Revocation

rrhe graph now contains two nodes (Art and Bob) for which there are outgoing arcs with labels containing the SELECT privilege on Sailors; therefore, these users have granted this privilege. lInwever, although each node contains an incorning arc carrying the saIne privilege, *there is no such path from Systern to either of these nodes*; so these users' right to grant the privilege has been abandonecL We therefore rernove the outgoing arcs as well. In general, these nodes rnight have other arcs incident on theIn, but in this exarnplc, they now have no incident arcs. Joe is left as the only user\vith the SELECT privilege on Sailors; Art and Bob have lost their privileges.

21.3.1 Grant and Revoke on Views and Integrity Constraints

The privileges held by the creator of a view (\vitll respect to the vie\v) change over time as he or she gains or loses privileges on the underlying tables. If the creator loses a privilege held 'with the grant option, users who were given that privilege on the view lose it as \vell. There are SOIHO subtle aspects to the GRANT and REVOKE conunands when they involve views or integrity constraints. We consider senne exarnples that highlight the following irnportant points:

1. A view Inay be dropped because a SELECT privilege is revoked froIn the user who created the view.
2. If the creator of a vie"v gains additional privileges on the underlying tables, he or she autornatically gains additional privileges on the view.
3. The distinction between the REFERENCES and SELECT privileges is irnportanto

Suppose that Joe created Sailors and gave Michael the SELECT privilege on it with the grant option, and J\!Iichael then created the view YoungSailors and gave Eric the SELECT privilege on YoungSailors. Eric now defines a view called FineYoungSailors:

```
CREATE VIEW FineYoungSailors (naIne, age, rating)
        AS SELECT S.snarne, S.age, S.rating
           FROM    YoungSailors S
           WHERE   S.rating > 6
```

What happens if Joe revokes the SELECT privilege on Sailors froln l\1icha,el? Michael no longer has the authority to execute the query used to define Young-Sailors because the definition refers to Sailors. rrherefore, the view YoungSailors is dropped (i.e., destroyed). In turn, Fine'{oungSailors is dropped as well. Both view definitions are rernoved fr0111 the systcln catalogs; even if (1, rerIlorseful Joe decides to give ba,ckthe SELECT privilege on Sailors to Michael, the views are gone a11d rnust be created afresh if they are required.

On a Inore happy note, suppose that everything proceeds as just described until Eric defines FineYoungSailors; then, instead of revoking the SELECT privilege on Sailors frorll Michael, .Joe decides to also give Michael the INSERT privilege 011 Sailors. Michael's privileges on the view YoungSailors are upgraded to what he would have if he were to create the vie\v *now*. He therefore acquires the INSERT privilege on 'YourlgSailors as well. (Note that this view is updatal)le.) What about Eric? His privileges are unchanged.

Whether OI' II.ot Michael has tlle INSERT privilege 011 \roungSailors with the gra11t ()ption depends 011 whether or not .Joe gives hirn the INSERT I)rivilege OII

Sailors with the grant option. To understand this situation, consider Eric again. If Michael has the INSERT privilege on YoungSailors with the grant option, he can pass this privilege to Eric. Eric could then insert rows into the Sailors table because inserts on YoungSailors are effected by rnodifying the underlying base table, Sailors. Clearly, we do not want Michael to be able to authorize Eric to rnake such changes unless Michael has the INSERT privilege on Sailors with the grant option.

rrhe REFERENCES privilege is very different froIll the SELECT privilege, as the following exarIlple illustrates. Suppose that Joe is the creator of Boats. He can authorize another user, say, Fred, to create H,eserves with a foreign key that refers to the *bid* colurnn of Boats by giving Fred the REFERENCES privilege with respect to this colulnn. ()n the other hand, if Fred has the SELECT privilege on the *bid* colurnn of Boats but not the REFERENCES privilege, Fred *cannot* create R.eserves with a foreign key that refers to Boats. If Fred creates R,eserves with a foreign key colunlll that refers to *bid* in Boats and later loses the REFERENCES privilege on the *bid* colurnn of boats, the foreign key constraint in Reserves is dropped; however, the R,eserves table is *not* dropped.

To understand why the SQL standard chose to introduce the REFERENCES privilege rather than to siInply allow the SELECT privilege to be used in this situation, consider what happens if the definition of Reserves specified the NO ACTION option with the foreign key------.-Joe, the owner of Boats, Inay be prevented from deleting a row fronl Boats because a row in Reserves refers to this Boats row. Giving Fred, the creator of Reserves, the right to constrain updates on Boats in this rnanner goes beyond. siInply allowing hinl to read the values in Boats, which is all that the SELECT privilege authorizes.

21.4 MANDATORY ACCESS CONTROL

Discretionary access coutrollnechanislns, while generally effective, have certain \veaknesses. In particular they are susceptible to *Trojan horse* schelnes whereby a devious unauthorized user can trick an authorized user into disclosing sensitive data. For exalnple, suppose that student rrricky Dick wants to break into the grade tables of instructor Trustin Justin. IJick does the following:

- He creates a new table called MineAllMine and gives INSERT privileges on this tahle to .Justin (who is blissfully unaware of all this attention, of course).
- He rllodifies the code of SOllIe I}BlVIS application that Jllstin uses often to do a couple of additional things: first, read the Grades table, Ctlld next, write the result into MineAllMine.

Then he sits back and waits for the grades to be copied into MineAllMine and later undoes the Illodifications to the application to ensure that Justin does not sOlnehow find out later that he has been cheated. Thus, despite the DBMS enforcing all discretionary access controls—only Justin's authorized code was allowed to access Grades—sensitive data is disclosed to an intruder. The fact that Dick could surreptitiously modify Justin's code is outside the scope of the DBMS's access control rnechanisrn.

NIandatory access control meehanisrns are airned at addressing such loopholes in discretionary access control. The popular rllodel for mandatory access control, called the Bell-LaPadula Illodel, is described in tenllS of objects (e.g., tables, views, rows, columns), subjects (e.g., users, prograrlls), security classes, and clearances. Each database object is assigned a *security class,* and each subject is assigned *clearance* for a security class; we denote the class of an object or subject *A* as *class(A).* The security classes in a systerll are organized according to a partial order, with a most secure class and a least secure class. For sirnplicity, we assume that there are four classes: *top secret (T8)*, *secret (8), confidential (C), and unclassified (U).* In this system, $T8 > S > C > U$, where $A > B$ rneans that class *A* data is more sensitive than class *B* data.

The Bell-LaPadula model imposes two restrictions on all reads and writes of database objects:

1. Simple Security Property: Subject *S* is allowed to read object 0 only if $class(8) \geq class(())$. For exarllple, a user with *TS* clearance can read a table with C clearance, but a user with C clearance is not allowed to read a table with *TS* classification.

2. *-Property: Subject *S* is allowed to write object 0 only if $class(S) \leq class(O)$. For exarllple, a user with *S* clearance can write only objects with *S* or *TS* classification.

If discretionary a,ccess controls are also specified, these rules represent additionaJ restrictions. Therefore, to read or write a database object, a user lllUst have the necessary privileges (obtained via GRANT cornrnands) *and* the security classes of the user and the object rnust satisfy the preceding restrictions. Let us consider how such a mandatory control rnech.an.isrn lui.ght h.ave foiled Tricky I)ick. rfhe Grades table could be classified as *S*, .Justin could be given clearance for *S*, and Tricky Di.ck could be given a lower clearance (C). Dick can create objects of only C or lower classification; so the table MineAllMine can have at Inos1, the classification C. When the application prograrIl running on behalf of ..Justin (and therefore\vith clearance S) tries to copy Grades into MineAllMine, it is not allowed to do so because $class(MineAllMine) < class(applicat'ion)$, and the *-Property is violated.

21.4.1 Multilevel Relations and Polyinstantiation

rro apply Inandatory access control policies in a relational DBMS, a security class must be assigned to each database object. The objects can be at the granularity of tables, rows, or even individual colurnn values. Let us assU111e that each row is assigned a security class. This situation leads to the concept of a multilevel table, which is a table with the surprising property that users with different security clearances see a different collection of rows when they access the sarne table.

Consider the instance of the Boats table shown in Figure 21.3. Users with *S* and *TS* clearance get both rows in the answer when they ask to see all rows in Boats. A user with C clearance gets only the second row, and a user with *[J* clearance gets no rows.

bid	*bname*	*color*	**Security Class**
101	Salsa	Red	*S*
102	Pinto	Brown	*C*

Figure 21.3 An Instance *B1* of Boats

The Boats table is defined to have *bid* as the prirnary key. Suppose that a user with clearance C wishes to enter the row *(101, Picante, Scarlet, C)*. We have a dilemrna:

- If the insertion is perlnitted, two distinct rows in the table have key 101.
- If the insertion is not pennitted because the priInary key constraint is violated, the user trying to insert the new row, who has clearance C, can infer that there is a boat with *bid=101* whose security class is higher than C. This situation cOlnpromises the principle that users should not be able to infer any infonnation about objects that have a higher security classification.

This dilerrllla is resolved by effectively treating the security classification as part of the key. rrhus, the insertion is allo\ved to continue, and the table instance is rnodified as shown in Figure 21.4.

bid	*bna'me*	*color*	**Security Class**
101	'--'Salsa	Red	*S*
101	Picante	Scarlet	*C*
102	Pinto	Brown	*C*

Figure 21.4 Insta.nce 131 after Insertion

lJsers\vith clearance *C* or [1 see just the rows for Picante and Pinto, but users with clearance *S* or *TS* see all three rows. The two ro\vs with *bid*=1()1 can be interpreted in one of two ways: only the rc)\v\vith the higher classification (Salsa, with classification 8) a,ctually exists, or both exist and their presence is revealed to users according to their clearance level. The choice of interpretation is up to application developers and users.

The presence of data objects that appear to have different values to users with different clearances (for exarnple, the boat with *b'id* 101) is called polyinstantiation. If we consider security classifications associated with individual colurnns, the intuition underlying polyinstantiation can be generalized in a straightforward manner, but SOIne additional details Inust be addressed. We relnark that the rnain drawback of rnandatory access control schelnes is their rigidity; policies are set by systeIll adrninistrators, and the classification 1nechanisrns are not flexible enough. A satisfactory cornbination of discretionary and rnandatory access controls is yet to be achieved.

21.4.2 Covert Channels, DoD Security Levels

Even if a DElVIS enforces the rnandatory access control schenle just discussed, inforrnation can flow frorn a higher classification level to a lower classification level through indirect rneans, called covert channels. For exanlplc, if a transaction accesses data at rnore than one site in a distributed DBMS, the actions at the two sites 1nust be coordina,ted. The process at one site rTlay have a lower clearance (say, C) than the process at another site (say, *S*), and both processes have to agree to cOllnnit before the transaction can be conunitted. This requirernent can be exploited to pass i11fo1'rnatio11 with an *S* classification to the process with a *C* clearance: The transaction is repeatedly invoked, and the process \vith the *C* clearance always agrees to cOlIllnit, whereas the process with the *S* clearance agrees to conunit if it wants to translnit a 1 bit and does not agree if it wants to transrnit a 0 l)it.

In this (adrnittedly tortuous) lIlanllcr, infonnation with an *S* clearance can be sent to a process with a *C* clearance as a strea111 of bits. This covert cllannel is an indirect violation of the intent behind the *-Property. Additional exarnples of covert channels can be found readily in statistical databases, which we cliscuss ın Scetlon 21,.6.2".

DBMS vendors recently started irnplcrnenting rnandatory access control mechaniSlns (although they are not part: of the SQL standard) because the United States l)epartnlent of J)efense (1)01)) requires such support for its systems. The Dol) requirernents can be described in terrns of security levels *A*, *,13*, *C*, and *D*. of \vhich *./1* is the 1J10st secure and 1) is the least secure.

Current Systems: Commercial RDBMSs are available that support discretionary controls at the *C2* level and mandatory controls at the *B1* level. IBM DB2, Inforruix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support SQL's features for discretionary access controL In general, they do not support lnandatory access control; Oracle offers a version of their product with support for rnandatory access control.

Level C requires support for discretionary access control. It is divided into sublevels *Cl* and *C2*; *C2* also requires soiiic degree of accountability through procedures such as login verification and audit trails. Level *B* requires support for lnandatory access control. It is subdivided into levels *Bl, B2,* and *B3.* Level 132 additionally requires the identification and clirnination of covert channels. Level *B3* additionally requires 11laintenance of audit trails and the designation of a security administrator (usually, but not necessarily, the DBA). Level *A,* the most secure level, requires a n1athernatical proof that the security rnechanisrn enforces the security policy!

21.5 SECURITY FOR INTERNET APPLICATIONS

When a DBMS is accessed frorn a secure location, we can rely upon a shnple password rnechanisrn for authenticating users. IIowever, suppose our friend Sarn wants to place an order for a hook over the Internet. rrhis presents sorne unique challenges: Saln is not even a known user (unless he is a repeat custonler). Fron1 Alnazon's point of view, we have an individual asking for a book and offering to pay with a credit card registered to Saln, but is this individual really Sarn? From Sarn's point of view, he sees a fornl asking for credit card inforrnation, but is this indeed a legitirnate part of Arnazon's site, and not a rogue application designed to trick hill1 into revealing his credit card nurnber?

Tlhis exarnple illustrates the need for a rnore sophisticated approach to authentication than a sirnple password rnechanisrn. Encryption techniques provide the foun,dation for rnodern authentica,tion.

21.5.1 Encryption

The basic idea behind encryption is to apply an encryption algorithrn to the data, using a user-specified or IJBA-SI)Ccified encryption key. The output of the algorithrn is the encrypted version of the data. There is aJso a decryption algorithrrL ·which takes the encrypted data and a decryption key as input and then returns the original data.\Vithont the correct decryption key, the decryption algoritll111produces gibl)crish. rrhe encryption and clecryption

DES and AES: The DES standard, adopted in 1977, has a 56-bit encryption key. Over time, computers have become so fast that, in 1999, a special-purpose chip and a network of PCs were used to crack DES in under a day. The systern was testing 245 billion keys per second when the correct key was fonnd! It is estimated that a special-purpose hardware device can be built for under a 1l1iUioll dollars that can crack DES in under four hours. Despite growing concerns about its vulnerability, DES is still widely used. In 2000, a successor to DES, called the Advanced Encryption Standard (AES), was adopted as the new (syrrunetric) encryption standard. AES has three possible key sizes: 128, 192, and 256 bits. With a 128 bit key size, there are over $3 \cdot 10^{38}$ possible AES keys, which is on the order of 10^{24} Inore than the number of 56-bit DES keys. Asslllne that we could build a conlputer fast enough to crack DES in 1 second. This COIllputer would. cornpnte for about 149 trillion years to crack a 128-bit AES key. (Experts think the universe is less than 20 billion years old.)

algorithrns thernselves are assunled to be publicly known, but one or both keys are secret (depending upon the encryption scheme).

In symmetric encryption, the encryption key is also used as the decryption key. The ANSI Data Encryption Standard (DES), which has been in use since 1977, is a well-known exarnple of syllunetric encryption. It uses an encryption algorithrn that consists of character substitutions and pernlutations. The nlain weakness of synunetric encryption is that all authorized users rnust be told the key, increasing the likelihood of its becorning known to an intruder (e.g., by sirnple Inllnan error).

Another approach to encryption, called public-key encryption, has becorne increasingly popular in recent years. The encryption scheniC proposed by Hjvest, Sharnir, and Adlernan, called RSA, is a well-known exarnple of public-key encryption. Each authorized user has a public encryption key, known to everyone, and a private decryption key, known only to hini or her. Since the private decryption keys are known only to their owners, the weakness of 1)ES is avoided.

A central issue for public-key encryption is how encryption and decryption keys are chosen. Technically, public-key encryption algorithrns rely on the existence of one-way functions, whose inverses are cornplltationally very hard to deterrnine. rrhe RSA algoritllIn, for example, is based on the observation that, although checking whether a given nurnber is prirne is easy, deterrnining the prirne factors of a nonprime nurnber is extrernely hard. (I)eterlnining the

Why RSA Works: The essential point of the scherne is that it is easy to compute d given e, p, and q, but *very* hard to cornpute d given just e and L. In turn, this difficulty depends on the fact that it is hard to determine the prime factors of L, which happen to be p and q. *A caveat:* Factoring is widely believed to be hard, but there is no proof that this is so. Nor is there a proof that factoring is the only way to crack RSA; that is, to comp d frolll e and L.

prirne factors of a nurnber with over 100 digits can take years of CPlJ tirne on the fastest available COIllputers today.)

We now sketch the idea behind the RSA algorithrn, assuming that the data to be encrypted is an integer I. To choose an encryption key and a decryption key for a given user, we first choose a very large integer L, larger than the largest integer we will ever need to encode.[1] We then select a nUl1lber e as the encryption key and cornpute the decryption key d based on e and L; how this is done is central to the approach, as we see shortly. Both L and e are l1lade public and used by the encryption algorithrn. However, d is kept secret and is necessary for decryption.

- The encryption function is $S \quad Ie \bmod L$.
- The decryption function is $1 \quad Sd \bmod L$.

We choose L to be the product of two large (e.g., 1024-bit), distinct prirne nurnbers, $p * q$. The encryption key e is a randornly chosen nlunber between 1 and L that is relatively prirne to $(p-1) * (q-1)$. The decryption key d is cornputed such that $d * e = 1 \bmod ((p-1) * (q-1))$. Given these choices, results in nurnber theory can be used to prove that the decryption function recovers the original ruessage frorll its encrypted version.

A very irnportant property of the encryption and decryption algoritluns is that the roles of the encryption and decryption keys can be reversed:

$$decrypt(\mathrm{d}, (encrypt(\mathrm{e}, \mathrm{I}))) = \mathrm{I} = decrypt(\mathrm{c}, (encrypt(\mathrm{d}, \mathrm{I})))$$

Since In.any protocols rely on this property, we henceforth sirnply refer to public and private keys (since both keys can be used for encryption as well as decryption).

[1] A message that is to be encrypted is decomposed into blocks such that each block can be treated as an integer less tha.n L.

While we introduced encryption in the context of authenticatioll, we note that it is a fundaIIental tool for enforcing security. A DBMS can use *encryption* to protect inforrnation in situations where the norrnal security rnechanisrns of the DBlVIS are not adequate. For exarnple, an intruder rnay steal tapes containing SOUIC data or tap a conunu.nieation line. By storing and transrnitting data in an encrypted forIn, the DBMS ensures that such stolen data is not intelligible to the intruder.

21.5.2 Certifying Servers: The SSL Protocol

Suppose we associate a public key and a decryption key with Alnazon. Anyone, say, user Sam, can send Alnazon an order by encrypting the order using Arnazon's public key. ()nly Arnazon can decrypt this secret order because the decryption algorithrn requires Arnazon's private key, known only to Arnazon.

This hinges on 8arn's ability to reliably find out Arnazon's public key. A number of cornpanies serve as certification authorities, e.g., Verisign. Arnazon generates a public encryption key e_A (and a private decryption key) and sends the public key to Verisign. Verisign then issues a certificate to Arnazon that contains the following inforrnation:

$$\langle \textit{Verisign Arnazoin, htl;P8:/www.arnazon.corn,} \; e_A \rangle$$

The certificate is encrypted using Verisign's own *private* key, which is known to (i.e., stored in) Internet Explorer, Netscape Navigator, and other browsers.

When 8an1 carnes to the Amazon site and wants to place an order, his browser, running the SSL protocol,[2] asks the server for the Verisign certificate. The browser then validates the certificate by decrypting it (using ·Verisign's public key) and checking that the result is a certificate with the HaIne Verisign, and that theURL it contains is that of the server it is talking to. (Note that an atternpt to forge a certificate will fail because certificates are encrypted using Verisign's private key, whieh is known only to Verisign.) Next, the browser generates a random session key, encrypt it using Arnazon's public key (which it obtained frorn the validated certificate anel therefore trusts), and sends it to the Amazon server.

Frorn this point on, the Arnazon server and the browser can use th.c session key (which both know and are confident tliat only they know) and a *symmetric* encrypticHl protc)collike AES or IJES to exchange securely encrypted rnessages: Messages are encrypted by the sender anel decrypted by the receiver using the sa,HIe session key. rrhe encrypted Inessages travel over the Internet and rnay be

[2] A browser uses the SSL protocol if the target URL begins with *https*.

intercepted, but they cannot be decrypted without the session key. It is useful to consider why \ve need a session key; after all, the bro\vser could sirnply have encrypted 8a1n's original request using Arnazon's public key and sent it securely to the Arnazon server. The reason is that, without the session key, the Amazon server has no way to securely send infonnation back to the bro\vser. A further advantage of session keys is that syrnrnetric encryption is cOlnputationally nluch faster than public key encryption. The session key is discarded at the end of the session.

Thus, 8aIn can be assured that only Alnazon can see the inforrnation he types into the fonn shown to hirn by the AIuazon server and the inforrnation sent back to hiln in responses froIn the server. However, at this point, Amazon has no assurance that the user running the browser is actually Sanl, and not SOlneone who has stolen Sarn's credit card. l-'ypically, rnerchants accept this situation, which also arises when a custoIner places an order over the phone.

If we want to be sure of the user's identity, this can be accoIuplished by additionally requiring the user to login. In our exarnple, 8arn 11lUSt first establish an account with Alnazon and select a password. (Sam's identity is originally established by calling hiln back on the phone to verify the account inforrnation or by sending elnail to an elnail address; in the latter case, all we establish is that the owner of the account is the individual with the given clnail address.) Whenever he visits the site and Anlazon needs to verify his identity, AInazon redirects hinl to a login fo1'1n *after* using SSL to establish a session key. The password typed in is transrnitted securely by encrypting it with the session key.

()ne rcrnaining drawback in this approach is that Arnazon now kno\vs Sarn's credit card nlunber, and he rnust trust Alnazon not to rnisuse it. The Secure Electronic Transaction protocol addresses this lirnitation. Every custolner rnust now obtain a certificate, with his or her own private and public keys, and every transaction involves the Alnazon server, the cust(nner's browser, and the server of a trusted third party, such as Visa for credit card transactions. The basic idea is that the browser encodes non-credit caTd inforrnation using AInazon's public key and the credit ca.rd infonnation using Visa's public key and sends these to the AJnazon server, which for"vards the credit card inforrnation (which it cannot decrypt) to the Visa server. If the Visa server a,pproves the inforrnation, the transa,ction goes through.

21.5.3 Digital Signatures

Suppose tllat ,Elnlcr, who works for ArnazoIl, a,nd Betsy, who works for McGraw-lIill,need to COllllll1Unicate with each other about inventory. Public key encryption can be used t() create digital signatures for rnessages. rrhat is, rnessages

can be encoded in such a way that, if Elmer gets a Inessage supposedly fr(nTI Betsy, he can verify that it is fronl Betsy (in addition to being able to decrypt the rnessage) and, further, *prove* that it is froIn Betsy at McGraw-lIill, even if the ll1cssage is sent froIn a IIotrnail account when Betsy is traveling. Sirnilarly, Betsy can authenticate the originator of Inessages froln Ellner.

If Ellner encrypts Inessages for Betsy using her public key, and vice-versa, they can exchange inforrnation securely but cannot authenticate the sender. Sorueone who wishes to irnpersonate Betsy could use her public key to send a rnessage to EIrner, pretending to be Betsy.

A clever use of the encryption sche111e, however, allows Elmer to verify whether the rnessage was indeed sent by Betsy. Betsy encrypts the rnessage using her *private* key and then encrypts the result using Elrner's public key. When Ellner receives such a 111essage, he first decrypts it using his private key and then decrypts the result using Betsy's public key. rrhis step yields the original un-encrypted rnessage. Furthermore, Ehner can be certain that the rnessage was composed and encrypted by Betsy because a forger could not have known her private key, and without it the final result would have been nonsensical, rather than a legible 111essage. Further, because even Elmer does not know Betsy's private key, Betsy cannot clairn that Ehner forged the ruessage.

If authenticating the sender is the objective and hiding the rnessage is not im-portant, we can reduce the cost of encryption by using a message signature. A signature is obtained by applying a one-way function (e.g., a hashing schelne) to the rnessagc and is considerably sInaHer. We encode the signature as in the basic digital signature approach, and send the encoded signature together with the full, unencoded 111cssage. rrhe recipient can verify the sender of the signa-ture as just described, and validate the I11essage itself by applying the one-way function and cOlnparing the result with the signature.

21.6 ADDITIONAL ISSUES RELATED TO SECURITY

Security is a l)road topic, and our coverage is necessarily lirnited. 'rhis section briefly touches on sorne additional irnportant issues.

21.6.1 Role of the Database Administrator

rrhe database administrator (IJBA) plays an irnportant role in enforcing the security-related aspects of a database design. In conjunction with the o\vners of the data, the I)BA aJso COlltributes to developing a security policy. The I)BA has a special i:1,ccount, which we call the systenl **account**, and is responsible

for the overall security of the systeru. In particular, the DBA. deals with the follo\ving:

1. Creating New Accounts: Each new user or group of users Blust be assigned an authorization ID and a password. Note that application prograIns that access the database have the saIne authorization ID as the user executing the prograill.

2. Mandatory Control Issues: If the DBMS supports rnandatory control·-···-S0111e custornized systeIns for applications with very high security requirernents (for exarllple, rnilitary data) provide such support—the DBA lllUst assign security classes to each database object and assign security clearances to each authorization ID in accordance with the chosen security policy·

The DBA is also responsible for rnaintaining the audit trail, which is essentially the log of updates with the authorization ID (of the user executing the transaction) added to each log entry. This log is just a Ininor extension of the log mechanislll used to recover from crashes. Additionally, the DBA may choose to rnaintain a log of *all* actions, including reads, perfornled by a user. Analyzing such histories of how the DBMS was accessed can help prevent security violations by identifying suspicious patterns before an intruder finally succeeds in breaking in, or it can help track down an intruder after a violation has been detected.

21.6.2 Security in Statistical Databases

A statistical database contains specific inforrnation on individuals or events but is intended to perlnit only statistical queries. For exarnple, if we rnailltained a statistical database of inforrna,tion about sailors, we would allow statistical queries about average ratings, rnaxirnurn age, and so on, but not queries about individua.l sailors. Security in such databases poses new probleurs because it is possible to infer protected inforrnation (such as a sailor's rating) frorn answers to perrnitted statistical queries. Such inference opportunities represent covert channels that can cornprornise the security policy of the database.

Suppose that sailor Sneaky Pete wants to kncrw the rating of .A.clrniral Ho1'n-tooter, the esteemed chairrnarl of the sailing clu1), and happens to kno\v that IIorntooter is the oldest sailor in the club. Pete repeatedly asks queries of the forln "How InClny sailors are there whose age is greater than X?" for various values of X, until the answer is 1. Obviously, this sa,iIor is IIorntooter, the oldest sailor. Note that each of these queries is a valicl statistical query and is permitted. Let the value of X at this point be, say, 65. Pete no\v asks the query, "What is the nraxirnurn rating of all sailors \vhose age is greater than

65?" Again, this query is pennitted because it is a statistical query. However, the answer to this query reveals J101'ntooter's rating to Pete, and the security policy of the database is violated.

One approach to preventing such violations is to require that each query rnust involve at least S01ne lnininuull nUluber, say, N, of l'()\VS. With a reasonable choice of N, Pete \vould not be able to isolate the inforrnation about 1101'ntooter, because the query about the maximum rating would fail. rrhis restriction, however, is easy to overCOIne. By repeatedly asking queries of the forIII, "How ruany sailors are there whose age is greater than X?" until the systenl rejects one such query, Pete identifies a set ofN sailors, including Florntooter. Let the value of X at this point be 55. Now, Pete can ask two queries:

- "What is the SlUll of the ratings of all sailors whose age is greater than 557" Since N sailors have age greater than 55, this query is perrnitted.
- "What is the SUIll of the ratings of all sailors, other than l1orntooter, whose age is greater than 55, and sailor Pete?" Since the set of sailors whose ratings are added up now includes Pete instead of Horntooter, but is otherwise the sallle, the rnunber of sailors involved is still N, and this query is also pennitted.

From the answers to these two queries, say, Al and A_2, Pete, who knows his rating, can easily calculate Horntooter's rating as $Al - A_2$ + *.Pete'8 rating.*

Pete succeeded because he was able to ask two queries that involved Illany of the sarne sailors. 'The nurnber of rows exalnined in corllrnon by two queries is called their intersection. If a limit were to be pla,ced on the alIlount of intersection perrnitted bet\veen any two queries issued by the same user, Pete could be foiled. Actually, a truly fiendish (and patient) user can generally find out inforruation about specific individuals even if the systcrn places a, rniniruurn nUlnber of ro\vs bound *(N)* and a rnaxirnurn intersection bound (M) on queries, hut the n1.11n])er of queries required to do this gro\vs in proportion to N/M. We can try to additionally lirnit the total nUlnbe1' of queries that a user is allowed to ask. but two users could still conspire to breach security. By Il1aintaining a log of all activity (including read-only accesses), such query patterns can be detected, icleally before a security violation occurs. This discussion should rnake it clear, however, that security in statistical databases is difficult to enforce.

21.7 DESIGN CASE STUDY: THE INTERNET STORE

We return to our case study and our friends at DBI)udes to consider security issues. 'There are three groups of users: custolners, employees, and the owner of the l>ook shop. (()f course, there is also the database adrninistrator, who

has universal access to all data and is responsible for regular operation of tlle database systcrTl.)

The owner of the store has full privileges on all tables. Custorners can query the Books table and place orders online, but they should not have access to other customers' records nor to other c11storne1'8' orders.DBDudes restricts access in two ·ways. First, it designs a simple Web page with several fonus similar to the page shown in Figure 7.1 in Chapter 7. This allo\vs custo1ners to subrnit a s111all collection of valid requests without giving tho1n the ability to directly access the underlying DBMS through an SQL interface. Second, I)B.Dudes uses the security features of the DBMS to li111it access to sensitive data.

rIhe \vebpage allows custorners to query the Books relation by ISBN nU111bcr, narne of the author, and title of a book. The webpage also has two buttons. The first button retrieves a list of all of the custolIler's orders that are not completely fulfilled yet. l'he second button displays a list of all cornpleted orders for that custorner. Note that custolllers cannot specify actual SQL queries through the Web but only fill in SCHne pararneters in a forn1 to instantiate an autonlatically generated SQL query. All queries generated through fonll input have a WHERE clause that includes the *cid* attribute value of the current custo1ner, and evaluation of the queries generated by the two buttons requires knowledge of the custolller identification nUlnber. Since all users have to log on to the website before browsing the catalog, the business logic (discussed in Section 7.7) lllust 1naintain state inforrnation about a custoDler (i.e., the Cllstorner identification nUlnber) during the custorner's visit to the website.

The second step is to configure the database to lirnit access according to each user group's need to know. DBI)udes creates a special customer account that has the following privileges:

```
SELECT ON Books, NewOrders, ()ldOrders, NewOrderlists, OldOrderlists
INSERT ON New()rders, OldOrders, New()rderlists, ()ld()rderlists
```

Ernployees should be able to acid new books to the catalog, upda,te the quantity of a book in stock, revise custorner orders if necessary, and update all custorner inforrnation *except the credit card information.* In fact, ernployees should not even be able to see a custorner's credit card nurnber. 1]lcreforc,DBDucles creates the following view:

```
CREATE VIEW CustomerInfo (cid,cnarnc,address)
       AS SELECT C.cid, C.cname, C.(l.cldress
          FROMCllstolners C
```

I)BI)udes gives the employee account the follc)\ving privileges:

```
SELECT ON CustomerInfo, Books,
          NewOrders, ()IdOrders, NewOrderlists, Old()rderlists
INSERT ON Cllstornerlnfo, Books,
          Nc\vOrders, 01dC)rders, NewOrderlists, ()ldOrderlists
UPDATE ON CustolnerInfo, Books,
          New()rders, OldOrders, NewOrderlists, Old()rderlists
DELETE ON Books, NewOrders, OldOrders, NewOrderlists, ()ldOrderlists
```

Observe that ernployees can rnodify Custornerlnfo and even insert tuples into it. This is possible because they have the necessary privileges, and further, the view is updatable and insertable-into. While it seerns reasonable that elllployees can update a custorner's address, it does sceln odd that they can insert a t1lple into Cllstornerlnfo even though they cannot see related infonna.tion about the custorner (i.e., credit card nurnber) in the Cllstorners tahle. The reason for this is that the store wants to be able to take orders 1'1'0111 first-tirne custorners over the phone without asking for credit card inforrnation over the phone. Ernployees can insert into Custornerlnfo, effectively creating a new Custoillers record without credit card inforluation, and custorners can subsequently provide the credit card nurnber through a Web interface. (Obviously, the order is not shipped until they do this.)

In addition, there are security issues when the user first logs on to the website using the cllstolner identification nUlnber. Sending the nUlnber unencrypted over the Internet is a security hazard, and a secure protocol such as SSL should be used.

Cornpanies such as CyberCash and DigiCash offer electronic conunerce pay-rIlcnt solutions, even inclu.ding *electronic cash.* Discussion of how to incorporate such techniques into the website are outside the scope of this book.

21.8 REVIEW QUESTIONS

Answers to the review questions can be founel in the listed sections.

- What are tIle 1n,ain objectives in designing a secure datal)ase application? Explain the te1'rns *secrecy*, *integrity*, *availability*, and *authentication.* (Section 21.1)
- Explain the terms *security policy* and *security mechanism* arid how tllCy are related. (Section 21.1)
- What is the Blain idea behind *discretionary access control?* What is the idea behind *mandatory access control?* What are the relative merits of these two approaches? (Section 21.2)

- Describe the privileges recognized in SQL? In particular, describe SELECT, INSERT, UPDATE, DELETE, and REFERENCES. For each privilege, indicate \vho acquires it automatically on a given table. (Section **21.3)**
- I-Io\v are the owners of privileges identified? In particular, discuss *authorization ID8* and *roles.* (Section 21.3)
- What is an *authorization graph?* Explain SQL's GRANT and REVOKE eOl11-mands in terrns of their effect on this graph. In particular, discuss what happens when users pass on privileges that they receive frorn sorneone else. (Section **21.3)**
- Discuss the difference between having a privilege on a table and on a vie\v defined over the table. In particular, how can a user have a privilege (say, SELECT) over a view 'without also having it on all underlying tables? Who 11111st have appropriate privileges on all underlying tables of the view? (Section 21.3.1)
- What are *objects, subjects, security classes,* and *cleaTances* in rnandatory access control? IJiscuss the Bell-LaPadula restrictions in tenns of these concepts. Specifically, define the *simple security property* and the **-pToperty.* (Section 21.4)
- What is a *Trojan horse* attack and how can it cOlnprornise discretionary access control? Explain how lnandatory a,ccess control protects against Trojan horse attacks. (Section 21.4)
- What do the tenns *multilevel table* and *polyinstantiation* mean? Explain their rela.tionship, a,nd how they arise in the context of lnandatory access control. (Section 21.4.1)
- What are *covert channels* and how can they arise when both discretionary and luandatory access controls are in place? (Section **21.4.2)**
- Discuss the I)oD security levels for database systclns. (Section 21.4.2)
- Explain why a sirnple password rnechanisrn is insufficient for authentication of users who access a database renJotely, say, over the Internet. (Section 21.5)
- What is the difference between *symmetric* and *public-key encryption?* Give examples of well-known encryption algoritluns of both ki.lldso What is the rnain weakness of synunetric encryption and how is this addressed in public-key encryption? (Section 21.5.1)
- 1)i8c118s the choice of encrYIltion and decryption keys in public-key en.cryption and how they are llsed to encrypt and decrypt data. Explain the role of *one-way functions.* What H.ssurance do\ve h.ave that the RSA scheme cannot be cornprornised? (Section 21.5.1)

- What are *certification authorities* and why are they needed? Explain how *certificates* are issued to sites and validated by a bro\vser using the *SSL protocol*; discuss the role of the *session key.* (Section 21.5.2)
- If a user connects to a site using the SSL protocol, explain \vhy there is still a need to login the USCI'. Explain the use of SSL to protect pass\vords and other sensitive infonnation being exchal1ged. What is the *secure electronic transaction protocol?* What is the added value over SSL? (Section 21.5.2)
- A *digital signature* facilitates secure exchange of rnessages. Explain what it is and how it goes beyond sirnply encrypting rnessages. Discuss the use of *message signatures* to reduce the cost of encryption. (Section 21.5.3)
- What is the role of the database achninistrator with respect to security? (Section 21.6.1)
- Discuss the additional security loopholes introduced in *statistical databases.* (Section 21.6.2)

EXERCISES

Exercise 21.1 Briefly answer the following questions:

1. Explain the intuition behind the two rules in the Bell-LaPadulamodel for rnandatory access control.
2. Give an exarnple of how covert channels can be used to defeat the Bell-LaPadula rnodel.
3. Give an exarnple of polyinstantiation.
4. Describe a scenario in whichrnandatory access controls prevent a breach of security that cannot be prevented through discretionary controls.
5. Describe a scenario in which discretionary access controls are required to enforce a seCll-rity policy that cannot be enforced using only mandatory controls.
6. If a DBMS already supports discretionary and Jnandatory access controls, is there a need for encryption?
7. Explain the need for each of the following lirnits in a statistical database systern:
   (a) A maxirnurn on the munber of queries a user can pose.
   (b) A rninirnUln on the munber of tuples involved in ans\vering a query.
   (c) A maximurn on the intersection of two queries (i.e., on the number of tuples that both queries exarnine).
8. Explain the use of an audit trail, with special reference to a statistical database system.
9. \,Vllat is the role of the DBA with respect to security?
10. Describe AES and its relationship to DES.
11. What is public-key encryption? How does it differ frorn the encryption approach taken in the Data Encryption Standard (DES), and in what ways is it better than DES?