CHAPTER 13

SPECIAL IMPLEMENTATION CONSIDERATIONS

CHAPTER OBJECTIVES

- Study the role of languages in the implementation of a data model
- Discuss in detail SQL, the relational standard
- Learn how SQL is used for data definition, manipulation, maintenance, and control
- Review examples of simple and complex queries in SQL
- Examine how a query is optimized and executed
- Understand the various aspects of database deployment

When you review the phases of the database development life cycle (DDLC), physical design is the last stage in the design phase. In Chapter 12, we discussed physical design in considerable detail. You learned how to transform the logical model into the physical model of a relational database system. You studied how the physical model conforms to the target DBMS and the configuration of the computer system where the database has to reside.

Once physical design is completed, you are ready to implement the database system. Implementation brings all the efforts during the different phases of DDLC to culmination. You have moved from the generic data model to the relational model. You know that both of these form the logical data model. After that stage, you moved to the physical model, taking into account the features of the DBMS and the computer system. The data models must be implemented as the resultant database management system.

First, try to understand the significance and meaning of implementing the data model. Find out why we need languages for implementation. We need languages

Database Design and Development: An Essential Guide for IT Professionals by Paulraj Ponniah
ISBN 0-471-21877-4

for defining the data structures, for manipulating the data, and for maintaining the data. Structured Query Language (SQL) has evolved as the standard to fulfill these needs.

Implementation includes tasks required to move from an earlier data system to the new database system. Existing data from the current system have to be converted over to the new system. Furthermore, you need applications in the new environment that will tap into the new database system. Also, you need a proper strategy to deploy the database and distribute the data wherever they are needed.

Implementation considerations include the optimal processing of data queries. Data retrieval involves execution of queries in the most optimal way. Our discussion of implementation considerations will not be complete without an examination of query processing.

IMPLEMENTATION LANGUAGES

You carry out the implementation of a database system through languages. You know that language is a medium for communication and serves as a link. Just think about programming languages. Programming languages enable user transactions to be communicated to the computer systems. They therefore contain structures to indicate the logic, components to denote the data requirements, and methods to organize the flow of control.

Database implementation languages also serve to interpret and communicate database transactions to the DBMS and to the database system. In practice, because they support distinct functions, database implementation languages must have distinct features specially intended for the required functions. Several implementation languages have evolved for the relational data model; however, SQL has become the accepted standard.

Even though the name of the language implies that SQL is intended just for queries, the language has all the components to support the various aspects of implementation. In the following sections of this chapter, you will observe the distinct features of SQL and understand how it is a complete language for implementation of the relational data model.

Meaning of Model Implementation

At the outset, let us describe what we mean by implementing a data model as a database system. Consider a relational data model. The logical data model consists of relations or tables, tuples or rows indicating individual entities, columns showing the attributes, primary keys, and foreign keys. In the physical data model you find these components transformed into files, blocks, records, indexes, and constraints. So what do we mean by implementing the data model?

Implementation serves two primary purposes—storage of data and usage of data. As you know, these are the two basic aspects of a database system. Users need to access and use data; you need to store the data in a database for this purpose. A data model represents the information requirements for an organization. When you implement a data model, you make provisions to store the information content represented by the data model and also enable the stored information to be used.

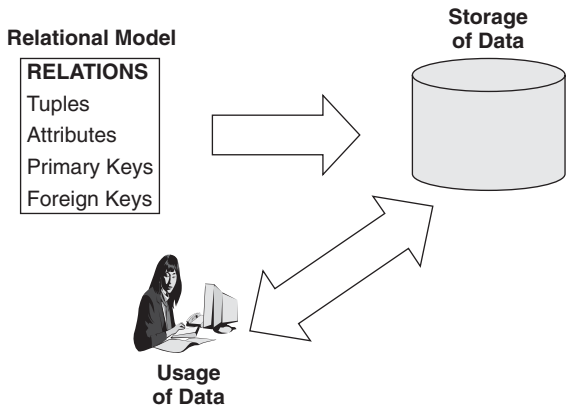


Figure 13-1 Aspects of data model implementation.

Figure 13-1 illustrates these two major aspects of implementing a data model as a database system. Note the model components on the left side of the figure and see how implementation of the data model provides for data storage and data usage.

Role of Languages

How is the implementation of a data model as a database system accomplished? Accomplishing the implementation includes storage of data and usage of data. You need a method, a medium, a language component for providing for each of the aspects of implementation. You must have a language component to support data storage; you need a component to enable the usage of data. In addition to these two primary requirements, there must be language components for some other aspects as well.

Let us review each of these aspects of model implementation and note the desired features of the language components. Note how each component serves a distinct purpose and understand how all these language components together make the implementation of the data model happen.

Defining Data Structure To store data, you need to define the data structure to the database management system. Data definition for the relational data model includes defining the logical model components as well as physical model components. The data definition language (DDL) module supports the storage of data in the database system.

Figure 13-2 shows the features of the DDL component and illustrates how this component enables data storage.

Data Retrieval Once the data structure is defined and the data are stored in the database, we need a language component to enable usage of the data. Data access is part of the overall ability to manipulate the data in the database. This part of the data manipulation language (DML) component enables retrieval of data from the

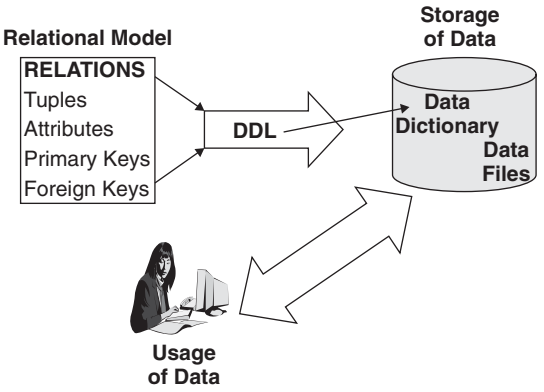


Figure 13-2 DDL component.

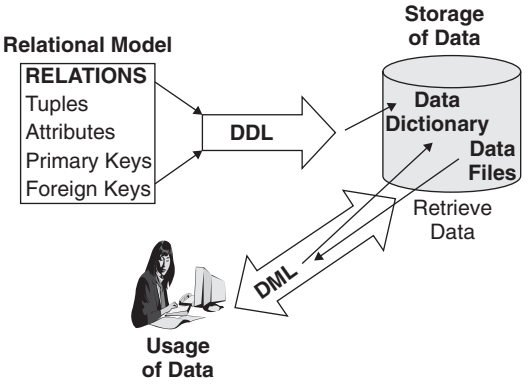


Figure 13-3 DML component: data access.

database system. Data access provides for retrieval of relevant portions of data based on various criteria.

Figure 13-3 indicates those features of DML that deal with data access.

Data Maintenance Data access and retrieval are just one part of data manipulation. The other parts include adding new data, deleting unwanted data, and revising data values as necessary. The data maintenance part of DML allows the maintenance of the data in the database.

Figure 13-4 presents the features of DML that are relevant to data maintenance in a database system.

Data Control "Usage of data in the database" implies controlled usage under proper authorization. There must be one language component to ensure proper data control. The data control language (DCL) enables data security and control through methods for proper access for authorized users. Authorization must include opening

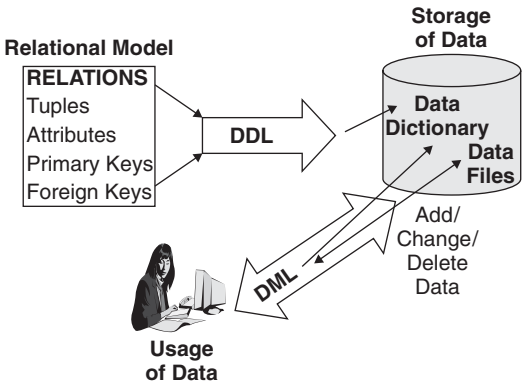


Figure 13-4 DML component: data maintenance.

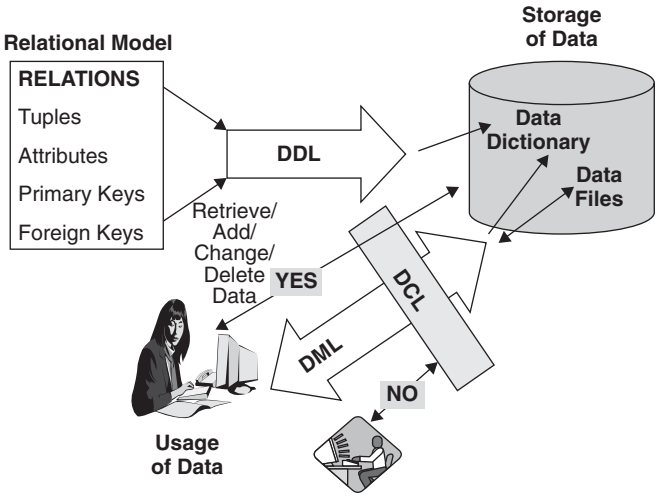


Figure 13-5 DCL component.

the database to authorized users not only for data retrieval, but also for adding, changing, or deleting data.

Figure 13-5 shows the features of the DCL component and how this component supports authorized data manipulation.

Languages for the Relational Model

As mentioned above, SQL has established itself as the preferred language for the relational model. Before we discuss SQL in detail in the next section, it is worthwhile to briefly introduce a few other commercial languages that support the relational data model. This introduction will give you some insight into the purpose and features of the language components that support the various aspects of data model implementation.

Queries on ORDER relation

Print all order numbers for Customer "Robert Smith"

OrdrNo	OrdrDate	OrdrAmt	CustName
P.			Robert Smith

Find the order numbers and order dates of all orders with amounts more than $750

OrdrNo	OrdrDate	OrdrAmt	CustName
P.	P.	>750	

Find the order numbers and order amounts of all orders not for "Bill Jones"

OrdrNo	OrdrDate	OrdrAmt	CustName
P.		P.	NOT Bill Jones

Figure 13-6 QBE queries.

Query-by-Example Query-by-example (QBE) adopts a visual approach to implementing the data model through templates. You enter an example value into the template for data retrieval. Queries are expressed as examples that are provided through the language.

QBE was developed in early 1970s at IBM's Watson Research Center. Later, the QBE data manipulation language component was used in IBM's Query Management Facility (QMF). Today, many relational DBMSs including Microsoft ACCESS and IBM's Paradox use QBE in one form or another.

In QBE, you use a skeletal two-dimensional table to express your query. Queries are in the form of tables, not free-form text with statements or commands. You fill in a skeletal table with example rows. An example row consists of example data elements and constants. The result of the query is also presented as a table.

Figure 13-6 shows examples of QBE queries.

QBE has provisions to perform the following:

- Indicate conditions for data retrieval through the use of condition boxes
- Display results taking values from several database relations
- Order the display of result rows
- Perform aggregations
- Data maintenance—deletion, insertion, and update

Quel Developed at the University of California, Berkeley, Quel was introduced as the implementation language for the Ingres database system. Three basic types of clauses exist: range of, retrieve, and where. The underlying structure resembles that of relational calculus.

You can express most of the queries just by using these three standard clauses. The first clause, "range of" declares a tuple or row variable taking on values of tuples

in a certain relation. The next clause, "retrieve" serves to indicate the values of selected attributes from the declared relation that must be found in the result. The "where" clause sets the conditions and constraints for the data retrieval.

Examine the following Quel queries and note the basic constructs.

Query: List the names of the customers in Japan.

range of r **is** *customer*
retrieve unique *r.customer-name* **where** *r.customer-country* = 'Japan'

Query: List the names and countries of customers who bought product 2241.

range of r **is** *customer*
range of s **is** *sale*
retrieve unique *(r.customer-name, r.customer-country)*
where *s.product-id = '*2241*'* **and** *r.customer-id = s.customer-id*

Quel has provisions to perform the following functions, all functions being structured in a similar basic format:

- Perform arithmetic aggregate functions such as count, sum, average, maximum, minimum, and so on
- Execute simple and complex queries
- Data maintenance—deletion, insertion, and update
- Do set operations

Datalog As you know, a nonprocedural language simply states the desired solution without writing out a set of procedural statements to arrive at the result. You have seen that relational calculus is a generic nonprocedural language. In the same way, Datalog is a nonprocedural query language developed based on Prologic, a logic-programming language. In a Datalog query, the user simply describes the desired information without writing a detailed procedure on how to obtain that information.

A Datalog program is made up of a set of rules. A Datalog view defines a view of a given relation. You may consider a view as a subset of the given relation. Figure 13-7 illustrates the definition of a Datalog view and the description of a Datalog query. Notice the nonprocedural nature of the language.

SQL: THE RELATIONAL STANDARD

Designed and developed at IBM Research in the early 1970s as the language interface for an experimental relational database, SQL has become the accepted relational standard. Consider the DBMS of any relational database vendor. You will find that every vendor has adopted the standard version of SQL and added some embellishments. It is even doubtful that the relational data model would have gained such popularity and acceptance without SQL.

SQL makes the transition from one relational DBMS to another easy. Businesses that move into relational databases from hierarchical or network database systems

VIEW DEFINITION:

Define a view relation v1 containing order numbers, order dates, and order amounts for customer "Bill Jones" with order amount greater than $1,000

V1 (A, B) := ORDER (A, B, C, "Bill Jones"), C > 1000

QUERY

Retrieve order numbers, order dates, order amounts for customer "Bill Jones"

? V1 (A, B) := ORDER (A, B, C, "Bill Jones")

QUERY

Retrieve order numbers, order dates, order amounts for order with amounts less than $ 150"

? V1 (A, B) := ORDER (A, B, C), C < 150

Figure 13-7 Datalog view and query.

or from file-oriented data systems may select an initial relational DBMS. As the organizations gain experience and desire to move forward to another, more robust relational DBMS, SQL makes the transition possible.

SQL is a command-driven, declarative, nonprocedural language serving as DDL, DML, and DCL. With SQL you can define database objects, manipulate data, and exercise data control. You can write SQL code as free-form text. The structure of the commands consists of standard English words such as CREATE, SELECT, FROM, WHERE, DROP, and so on. The language may be used by DBAs, analysts, programmers, and even end users with equal ease—a language easy to learn, but perhaps not so easy to master.

Let us now begin an in-depth discussion of SQL—its features, history and evolution, major commands, and usefulness. You will learn how to use facilities in SQL to define data structures, manipulate data, compose queries, provide data control, and manage data. You will also study examples of simple and complex queries written in SQL. Nevertheless, this coverage of SQL is not meant to be a detailed reference guide. You are encouraged to continue your study of SQL with the aid of books specifically intended for that purpose.

Overall Features

SQL is a comprehensive database or model implementation language mostly based on tuple relational calculus. The language also includes some relational algebra operations. It combines features and functions both for definition of data structures and for usage of data in the database. The widespread use and acceptance of SQL enables it to evolve and become more robust.

Because the underlying construct is based on relational calculus, a nonprocedural language, there is no sequencing of the commands in SQL. An SQL query is simply written as a solution statement declaring the outcome desired from execution of the query. How the result is formatted and presented to the user is left outside of SQL. In SQL you find only definitional and manipulative commands. You do not have flow of control commands—no if-then-else navigation within SQL code. Therefore, SQL has to be embedded in a host language to provide flow of control. The host

language constructs provide the flow of control logic and also the presentation of results on GUI screens or reports.

Major Aspects of SQL

- *Data definition language component*—for creation, deletion, and modification of definitions for relational tables and user views; for creation and dropping of indexes (as provided in commercial versions); for specifying integrity constraints
- *Data manipulation language component*—consisting of a query language based on relational calculus and some relational algebra
- *Data maintenance facility*—to insert, modify, or delete rows from relational tables
- *Security control*—to grant or revoke access privileges to users
- *Transaction control*—commands specifying beginnings and endings of transactions with explicit locking by some implementations for concurrency control
- *Embedded SQL*—providing calls from host languages
- *Dynamic SQL*—for constructing and executing queries dynamically at run time
- *Triggers*—actions coded as program modules executed by DBMS whenever changes to the database meet conditions stipulated in the trigger
- *Client/server adaptation*—for client application programs to connect to an SQL database server
- *Remote access*—access of relational database over a network

Brief History and Evolution

It all began in the early 1970s when Dr. E. F. Codd of IBM's San Jose Research Laboratory proposed the relational data model. IBM developed a prototype DBMS called System R to validate the feasibility of the relational data model. D. Chamberlin, from the same laboratory, defined a language called the Structured English Query Language as the language interface for System R. This language came to be known as SEQUEL. However, SEQUEL owes its beginning to another language called SQUARE (Specifying Queries as Relational Expressions), designed to implement relational algebra with English, which formed the basis for the development of SEQUEL. Development of SEQUEL, later known as SQL (pronounced as *es-que-el*), formed the high point of the relational model projects at San Jose.

In the late 1970s, Oracle became the first commercial implementation of a relational DBMS based on SQL. INGRES followed shortly thereafter as an implementation of relational DBMS with QUEL as the interface language. IBM's first commercial DBMS on the relational data model known as SQL/DS appeared in 1981 for DOS/VSE environments, and in 1982 for the VM/CMS environments. Subsequently, in 1983 IBM introduced DB2 for MVS environments. Now the relational DBMS of every vendor is based on SQL. Each vendor provides all the standard features of SQL. Each vendor also provides additional features of its own, called extensions to standard SQL. Vendors attempt to distinguish their SQL versions with these extensions.

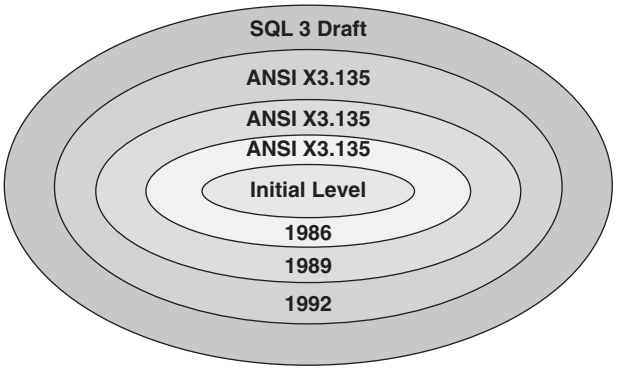


Figure 13-8 SQL: evolution of standards.

Evolution of Standards Both the American National Standards Institute (ANSI) and the International Standards Organization (ISO) have adopted and standardized SQL over the years. Figure 13-8 shows the evolution of SQL as standardization continues. Note how each new level adds and consolidates features.

Highlights of Each Version

ANSI X3.135—1986
Definition of language constructs: verbs, clauses, operators, syntax
Details of functions left to the discretion of implementers

ANSI X3.135—1989
Integrity enhancement
Support for primary and foreign key

ANSI X3.135—1992 (SQL 2)
Language enhancements
Detailed definitions for existing functions

SQL 3 Draft
Object-oriented capabilities

Data Definition in SQL

When you design a database system, you proceed through the logical and physical phases. The components developed in these phases form the basis for your database system. For the database management system (DBMS) to manage the database content, the components of the logical design and the physical design must be defined to the DBMS. In a relational database system, you have to define the logical structures in terms of tables, columns, and primary keys. You need to relate the tables and rows to physical files, blocks, and records. These also must be defined to the DBMS.

Data definition language serves to define the logical and physical components to the DBMS. You know that the definitions are recorded in the data dictionary of the DBMS. The data definition language component of SQL makes it possible to define the logical and physical components of the relational data model to the DBMS.

Definition of Database Objects SQL provides functions to define and create database objects. In addition to the provision of statements to define and create the overall schema consisting of the complete set of relations in the database, SQL provides statements for the specification of the following for each relation:

- Schema and name for the relation
- Security and authorization
- List of attributes
- Data type and domain of values for each attribute
- Primary and foreign keys
- Integrity constraints
- Indexes
- Structure on physical storage medium

Here is a list of the major data definition statements in SQL:

CREATE SCHEMA	Define a new database schema for the set of relations or tables
DROP SCHEMA	Remove the definition and contents of an existing database
CREATE TABLE	Define a new table with its columns
ALTER TABLE	Add or delete columns to an existing table
DROP TABLE	Remove the definition and contents of an existing table
CREATE VIEW	Define a logical data view consisting of columns from one or more tables or views
DROP VIEW	Remove an existing view
CREATE DOMAIN	Define a domain name with certain a data type (domain name to be used later to define the data types of attributes)
ALTER DOMAIN	Change the data types of a defined domain
DROP DOMAIN	Remove a defined domain
CREATE INDEX	Define an index on one or more columns
DROP INDEX	Remove an existing index

SQL-based database management systems usually provide additional definition features as noted below:

CREATE SYNONYM	Define an alternative name for a table or view (usually an abbreviation or acronym to reduce the number of key strokes for referring to the table or view)
DROP SYNONYM	Remove a defined synonym
LABEL	Define a column heading for an attribute when results are displayed
COMMENT	Provide remarks or comments for table columns (stored as part of the table in the data dictionary)

Data Types SQL supports a variety of data types for the table columns representing the attributes. In the database, legal values may be entered for each column based on the declared data type. The standard versions of SQL allow for accepted data types; in addition, specific vendor implementation may have additional data types or some variations of a standard data type.

The following is a list of the common SQL data types:

Character data	
CHAR(*n*)	Alphanumeric character data of fixed length *n*
VARCHAR (*n*)	Alphanumeric character data of variable length up to a maximum length of *n*
Numeric data	
DECIMAL (*m*,*n*)	Signed number, where *m* is the total number of digits including the sign and *n* is the number of digits to the right of the decimal point
INTEGER	Large positive or negative whole numbers up to 11 digits
SMALLINT	Small positive or negative whole numbers up to 5 or 6 digits
FLOAT (*m*)	Floating point number represented in scientific notation with specified precision of at least *m* digits
Date/time data	
DATE	Calendar date showing year, month, and day in a prescribed format (year/month/day, month/day/year, or day/month/year)
TIME	Hour, minute, second with time zone
Interval data	
INTERVAL	Representation of periods of time
Bit data	
BIT (*n*)	Bit string of fixed length *n*
Logical data	
LOGICAL	"True" or "False" values

SQL Data Definition Example You have now gone through the various SQL statements available for defining database objects. You have also noted the provisions to indicate the different data types for values to be stored in the database. Let us put what we have covered so far in a practical example. In Figure 11-5, you studied an entity-relationship diagram for a florist business. Also, Figure 11-10 presents the relational schema for that business. We will refer to those figures and create data definitions with SQL statements. While doing so, we will provide examples of most of the data definition statements listed above.

Carefully observe all the statements now shown in Figure 13-9. This figure presents the definition of the schema with the tables, columns, indexes, and so on for

```
CREATE SCHEMA RAINBOW-FLORIST
        AUTHORIZATION Amanda-Russo;
CREATE DOMAIN ItemIdentifier CHARACTER (4) DEFAULT "ZZZZ"
        CHECK (VALUE IS NOT NULL);
CREATE TABLE CUSTOMER (
        CustNo                          CHARACTER (8),
        CustAddr                        CHARACTER (45),
        CustName        CHARACTER (35),
        PRIMARY KEY (CustNo) );
CREATE TABLE ORDER (
        OrdrNo                          CHARACTER (8),
        OrdrDate                        DATE,
        OrdrAmt                         DECIMAL (9,2)  NOT NULL,
        CustNo                          CHARACTER (8),
        PRIMARY KEY (OrdrNo),
        FOREIGN KEY (CustNo) REFERENCES CUSTOMER (CustNo)
                        ON DELETE CASCADE );
CREATE TABLE PAYMENT (
        OrdrNo                          CHARACTER (8),
        PmntNo                          CHARACTER (4),
        PmntDate                        DATE,
        PmntAmt                         DECIMAL (9,2)  NOT NULL,
        PRIMARY KEY (OrdrNo, PmntNo),
        FOREIGN KEY (OrdrNo) REFERENCES ORDER (OrdrNo) );
CREATE TABLE PRODUCT (
        ProdID                          ItemIdentifier,
        ProdCost                        DECIMAL (5,2)  NOT NULL,
        CHECK (ProdCost < 100.00 AND ProdCost > 1.00),
        PRIMARY KEY (ProdID) );
```

```
CREATE TABLE FLOWER (
                ProdID                          ItemIdentifier,
                ProdCost                        DECIMAL (5,2)  NOT NULL,
                FlwrName                        CHARACTER (25),
                FlwrSize                        SMALLINT,
                PRIMARY KEY (ProdID),
                FOREIGN KEY (ProdID) REFERENCES PRODUCT (ProdID) );
CREATE TABLE ARRANGEMENT (
                ProdID                          ItemIdentifier,
                ProdCost                        DECIMAL (5,2)  NOT NULL),
                ArngeType                       CHARACTER (15),
                PRIMARY KEY (ProdID),
                FOREIGN KEY (ProdID) REFERENCES PRODUCT (ProdID) );
CREATE TABLE CHANNEL (
                ChnlID                          ItemIdentifier,
                ChnlName                        CHARACTER (35),
                PRIMARY KEY (ChnlID) );
CREATE TABLE GROWER (
                GrwrID                          ItemIdentifier,
                GrwrName                        CHARACTER (35),
                PRIMARY KEY (GrwrID) );
CREATE TABLE SHIPMENT (
                OrdrNo                          CHARACTER (8),
                ChnlID                          ItemIdentifier,
                GrwrID                          ItemIdentifier,
                ShipDate                        DATE,
                ShipQty                         INTEGER,
                PRIMARY KEY (OrdrNo, ChnlID, GrwrID),
                FOREIGN KEY (OrdrNo) REFERENCES ORDER (OrdrNo),
                FOREIGN KEY (ChnlID) REFERENCES CHANNEL (ChnlID),
                FOREIGN KEY (GrwrID) REFERENCES GROWER (GrwrID) );
```

Figure 13-9 Florist business: data definition in SQL.

the florist business. Walk through each data definition statement, note its purpose, and see how it is constructed.

Data Retrieval in SQL

Whereas data manipulation in SQL deals with accessing and fetching desired data from a relational database, data maintenance in SQL relates to the adding, deleting, and changing of data. SQL provides commands for both types of data handling—data retrieval and data maintenance. The SQL query is used to fetch data from a relational database. Let us therefore begin our discussion on data manipulation by examining the SQL query format.

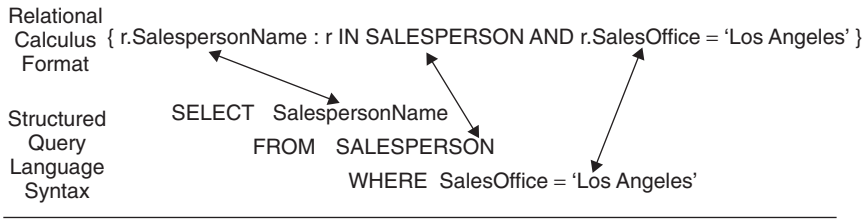
The basic structure of an SQL query expression consists of three simple clauses or commands: SELECT, FROM, and WHERE. Take a simple example. Let us say that you want to list all sales persons working out of the Los Angeles office. For this purpose, you need to write a query in SQL to retrieve the data about all the sales persons from those rows in the SALESPERSON table where the SalesOffice column contains the value "Los Angeles." Here is this simple query in English:

> What are the names of the sales persons in the Los Angeles office?

Now let us write the query using the data manipulation language (DML) component of SQL.

```
SELECT SalespersonName
  FROM SALESPERSON
    WHERE SalesOffice = 'Los Angeles'
```

SELECT This clause contains the list of columns from the base table to be projected into the result relation. An asterisk (*) in the clause indicates projec-



SELECT– *Lists the columns from base tables to be projected into the result relation.*

FROM – *Identifies tables from which columns will be chosen.*

WHERE – *Includes the conditions for row selection within a single table and conditions between tables for joining*

Figure 13-10 Query in relational calculus and in SQL.

tion of all columns from the base table. The operation corresponds to the project operation of relational algebra.

FROM This clause identifies the table or tables from which rows and columns will be chosen. The operation corresponds to the Cartesian product operation of relational algebra.

WHERE This clause includes the conditions for row selection within a single table or for joining two tables. The clause corresponds to the select or join operations of relational algebra.

Figure 13-10 shows the correspondence between the components of a relational calculus expression and the basic structure of an SQL query. Note how the query for the names of sales persons from the Los Angeles office is shown in the relational calculus format and with SQL query syntax.

In addition to the three basic clauses, SQL provides a few optional clauses as follows:

GROUP BY	To form groups of rows with the same values in given columns
HAVING	To filter the groups of rows subject to a given condition
ORDER BY	To specify the order of the rows in the result

SELECT, the most common clause in SQL, may contain specifications as to the selection of the columns to be presented in the result set. These specifications are indicated by the following keywords:

DISTINCT	To eliminate duplicates in the result
ALL	To state explicitly that duplicates must not be eliminated
AS	To replace a column or attribute name in the table with a new name

In a later subsection, we will consider several examples of SQL queries—both simple and complex. Let us now indicate the general format of the SQL query and move on.

```
SELECT      [DISTINCT|ALL] {*|[column-name [AS new-name ], […..], ….}
FROM        table-name [alias], ……..
WHERE       condition
GROUP BY    column-list        HAVING condition
ORDER BY    column-list
```

Data Maintenance in SQL

As discussed thus far, SQL provides you with facilities to define the database objects and to retrieve data once they are stored. But you know that the data in a database keep changing all the time. If you have a customer database table, you need to add more rows to the table as and when your organization gets new customers. If your database has a table for customer orders, new rows must be added to this table almost daily. Whenever changes are made to any of the rows in the customer or order tables, you need to update the relevant rows. After a period of time, you may want to delete the old orders and archive them on a separate storage medium. In this case, you must be able to delete selected rows from the orders table. All of these actions form part of data maintenance. Every database is subject to continual data maintenance.

SQL provides three commands to perform data maintenance. Let us discuss each command and understand it with some examples.

Adding Data

SQL Command

INSERT

Function

Add a new row to a table using the given values for the columns, or add one or more rows to a table using the result of a query.

Examples

1. Add a new row into the EMPLOYEE table with the supplied values for all columns.

   ```
   INSERT INTO EMPLOYEE
   VALUES   (1234, 'Karolyn', 'Jones', '733 Jackie Lane, Baldwin Harbor, NY
            11510', '516-223-8183', 'Senior VP', 95000, 'MGR')
   ```

2. Add a new row into the EMPLOYEE table with the supplied values for most of the columns, leaving some column values as NULL.

   ```
   INSERT INTO EMPLOYEE
   VALUES   (3344, 'Robert', 'Moses', '142 Crestview Drive, Long Branch, NJ
            08835', NULL, NULL, 40000, 'STF')
   ```

3. Add new rows into the MANAGER, selecting managers from the EMPLOYEE table.

```
INSERT INTO MANAGER
   SELECT *
      FROM EMPLOYEE
         WHERE EmployeeCode = 'MGR'
```

Modifying Data

SQL Command

UPDATE

Function

Change values in one or more columns of specified rows by replacing current values with supplied constants or with results of calculations.

Examples

1. Change last name in a specified row in EMPLOYEE table with supplied value.

```
UPDATE EMPLOYEE
   SET LastName = 'Raj'
      WHERE EmployeeNo = 1234
```

2. Increase the salary of all managers by 5%.

```
UPDATE EMPLOYEE
   SET Salary = Salary * 1.05
      WHERE EmployeeCode = 'MGR'
```

3. Modify multiple columns in a specified row in EMPLOYEE table with supplied values.

```
UPDATE EMPLOYEE
   SET EmployeePosition = 'Asst. Supervisor', Salary = 60000, EmployeeCode
   = 'SPR'
      WHERE EmployeeNo = 3344
```

Deleting Data

SQL Command

DELETE

Function

Delete one or more specified rows from a table.

Examples

1. Delete all rows from the MANAGER table.

```
DELETE FROM MANAGER
```

2. Delete one specific row from the EMPLOYEE table.

```
DELETE FROM EMPLOYEE
   WHERE EmployeeNo = 3344
```

3. Delete from EMPLOYEE table those rows for staff earning less than 25000.

```
DELETE FROM EMPLOYEE
   WHERE EmployeeCode = 'STF' and Salary <25000
```

Data Control in SQL

A relational database management system controls access to the database based on three concepts: user identifier, data ownership, and access privileges. The database administrator assigns a user identifier and password to each authorized user for signing onto the database system. A user with a proper user identifier performs every action on the database. Access privileges indicate which database objects a user may access and what actions he or she may perform. Chapter 16 explores this topic of database security in greater detail.

In a relational database, every object has an owner. The user identifier indicates the owner as defined in the AUTHORIZATION clause during schema definitions. Initially, only the owner has all access privileges to the database objects. When an authorized user creates a table using the CREATE TABLE statement, the user automatically becomes the owner of the table, receiving all access privileges to that table. Later, the owner may give access privileges to other authorized users.

SQL provides two commands for controlling access to the relational database. The database administrator has these commands to give access privileges to individuals or groups of users and to take away the privileges as soon as the privileges are to be removed. Access privileges permit users to perform various actions on tables or views. The privileges include:

SELECT	Permission only to retrieve data from a table or view
INSERT	Permission to add new rows into a table
UPDATE	Permission to modify the contents of a table
DELETE	Permission to remove rows from a table
REFERENCES	Permission to reference columns of a named table in integrity constraints

INSERT, UPDATE, and REFERENCES may be restricted to specific columns in the applicable table.

Providing Access Privileges

SQL Command

GRANT

Function

Give access privileges on database objects to specific users.

Examples

Use the following tables for the examples:

DEPARTMENT (DeptNo, DeptName, DeptLocation)
EMPLOYEE (EmployeeNo, FirstName, LastName, Address, Phone, EmployeePosition, Salary, EmployeeCode, DeptNo)
Foreign Key: DeptNo REFERENCES DEPARTMENT

1. Give permission to Rogers to read all of EMPLOYEE table.

```
GRANT SELECT
   ON EMPLOYEE
      TO Rogers
```

2. Give Miller and Rodriguez full privileges to the DEPARTMENT table.

```
GRANT ALL PRIVILEGES
   ON DEPARTMENT
      TO Miller, Rodriguez WITH GRANT OPTION
```

3. Give Chen read and update privileges on specific columns in EMPLOYEE table.

```
GRANT SELECT, UPDATE (FirstName, LastName, Address, PhoneNo)
   ON EMPLOYEE
      TO Chen
```

Removing Access Privileges

SQL Command

REVOKE

Function

Remove access privileges previously granted to database objects from users.

Examples

Use the same DEPARTMENT and EMPLOYEE tables for the following examples:

1. Remove from all users the privilege to delete DEPARTMENT table rows.

```
REVOKE DELETE
   ON DEPARTMENT
      FROM PUBLIC
```

2. Remove from Miller all privileges on DEPARTMENT table.

```
REVOKE ALL PRIVILEGES
   ON DEPARTMENT
      FROM Miller
```

3. Remove from Chen update privileges on EMPLOYEE table.

```
REVOKE UPDATE
   ON EMPLOYEE
      FROM Chen
```

Queries

Let us now examine a few examples of queries. First, let consider simple queries, mostly deriving results from single database tables. These will increase your grasp of the basic query statements in SQL. After that, we will go over a few examples of more advanced queries where the results need to be obtained from more than one table. There you will become familiar with queries within queries. The technique of using subqueries enables a complex query to be broken down into simple queries to produce the desired result.

Do not assume that the examples here form a complete set of all possible types of queries. The intention here is to provide you with a good introduction to SQL queries. Any standard SQL reference guide will provide you with a more comprehensive coverage of all types of queries.

Simple The queries in these examples are based on the data model for a university we considered in earlier chapters. As you are already familiar with this data model, let us use it to illustrate the following query types. Go back and review the following figures from earlier chapters:

Figure 7-22 ERD: university teaching aspects
Figure 9-22 University ERD: transformation into relational data model

1. **Find the name of the course with course number 10971.**

```
SELECT CourseDesc
   FROM COURSE
      WHERE CourseNo = '10971'
```

2. **Find the names of students with computer science major.**

```
SELECT StudntName
   FROM STUDENT
      WHERE Major = 'Computer Science'
```

3. **List all data for faculty member John Saunders.**

```
SELECT *
   FROM FACULTY
      WHERE FacltyName = 'John Saunders'
```

4. **Find all the majors for which students have enrolled.**

```
SELECT DISTINCT Major
   FROM STUDENT
```

5. **Find the maximum and minimum price of prescribed textbooks.**

```
SELECT MAX (Price), MIN (Price)
   FROM TEXTBOOK
```

6. **Add 5 points to minimum score for mid-term examination.**

```
SELECT MinScore, (MinScore+5)
   FROM EXAMTYPE
      WHERE TypeId = 'MIDTERM'
```

7. **Print the total number of faculty members in the English department.**

```
SELECT "The total number of faculty members in", Department, "is"
COUNT (FACULTY)
   FROM FACULTY
      WHERE Department = 'English'
```

8. **Find the number of credits for the course with course number 10971.**

```
SELECT Credits
   FROM COURSE
      WHERE CourseNo = '10971'
```

9. **List students by social security number in a given class with total score greater than given number.**

```
SELECT SocSecNo, ClassNo, SUM (Score)
   FROM GRADE
      WHERE ClassNo = '11223344'
         GROUP BY SocSecNo, ClassNo
            ORDER BY SocSecNo, ClassNo
               HAVING SUM (Score) > 60
```

10. **Find the students who have not yet declared a major.**

```
SELECT SocSecNo, StudntName
   FROM STUDENT
      WHERE Major IS NULL
```

11. **Find the names and phone numbers of faculty members with specialization in data systems.**

```
SELECT FacltyName, Phone
   FROM FACULTY
      WHERE Speczn LIKE 'Data%'
```

Advanced These examples of advanced queries are based on the relational data model for a medical center as shown in Figure 13-11. In these queries, observe how subqueries are used to obtain the desired results.

1. **List the services billed to patient whose number is 224466.**

```
SELECT SERVICE.ServiceCode, ServiceDesc
   FROM SERVICE, BILLING
      WHERE SERVICE.ServiceCode = BILLING.ServiceCode and
         PatientNo = 224466
```

2. **List the services billed to patient whose number is 224466. (Alternative query.)**

```
SELECT SERVICE.ServiceCode, ServiceDesc
   FROM SERVICE
      WHERE ServiceCode =
         (SELECT ServiceCode FROM BILLING
            WHERE PatientNo = 224466)
```

PATIENT (PatientNo, PatientName, DOB, AdmissionDate, DischargeDate)

DOCTOR (DoctorID, DoctorName, DoctorPhone)

SERVICE (ServiceCode, ServiceDesc)

TREATMENT (PatientNo, DoctorID, TreatmentDesc)
Foreign Key: **PatientNo** REFERENCES **PATIENT**
DoctorID REFERENCES **DOCTOR**

BILLING (PatientNo, ServiceCode, BilledAmount)

Foreign Key: **PatientNo** REFERENCES **PATIENT**
ServiceCode REFERENCES **SERVICE**

Figure 13-11 Relational data model: medical center.

3. List those patients who have been charged more than the average for service SCAN.

```
SELECT DISTINCT PatientNo
   FROM BILLING
      WHERE ServiceCode = 'SCAN'
         And BilledAmount >
         (SELECT AVG (BilledAmount)
            FROM BILLING
               WHERE ServiceCode = 'SCAN')
```

4. List those patients who have been charged for XRAY and SCAN.

```
SELECT DISTINCT PatientNo
   FROM BILLING B1
      WHERE EXISTS
         (SELECT *
            FROM BILLING B2
               WHERE B1.PatientNo = B2.PatientNo and
                  B1.ServiceCode = 'XRAY' and
                  B2.ServiceCode = 'SCAN')
```

5. List all patients who have been charged the same for XRAY and SCAN.

```
SELECT PatientNo, PatientName
   FROM PATIENT
      WHERE PatientNo IN
         (SELECT PatientNo
            FROM BILLING
               WHERE ServiceCode = 'XRAY' and
                  BilledAmount = ALL
         (SELECT BilledAmount
```

```
        FROM BILLING
          WHERE ServiceCode = 'SCAN') )
```

6. **Obtain a list of all billings including those patients for whom no billings have been done until now.**

```
SELECT BILLING.PatientNo, PatientName, BilledAmount
   FROM BILLING, PATIENT
      WHERE BILLING.PatientNo = PATIENT.PatientNo
UNION
SELECT PatientNo, PatientName, 0
   FROM PATIENT
      WHERE PatientNo NOT IN
        (SELECT PatientNo
           FROM BILLING)
```

Summary of SQL Query Components

Having reviewed various types of SQL queries, let us summarize the different query components. Figure 13-12 provides a summary of the main clauses, operators, and built-in functions that are part of typical SQL queries.

Database Access from Application Program

The queries discussed in the previous subsection illustrate one method of using SQL. We have covered the interactive use of SQL. To obtain the desired results from the database, you write SQL code in the database environment. The commercial database systems support SQL and execute the SQL statements interactively. The end of the SQL code, usually signified by a semi-colon (;), triggers the

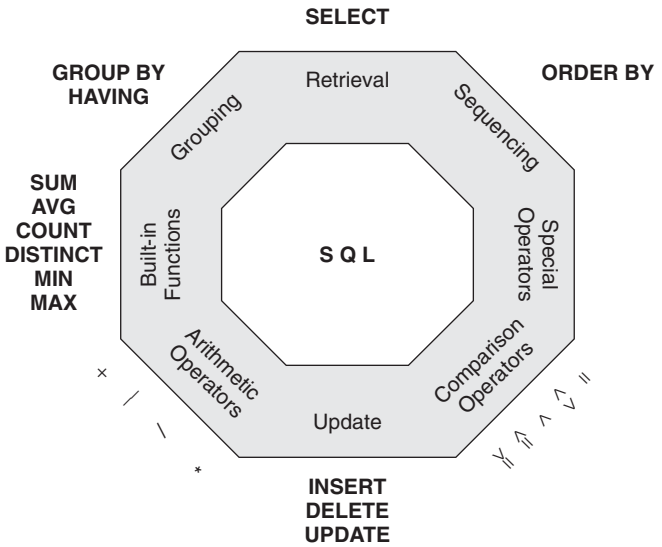


Figure 13-12 SQL query and manipulation components.

execution of the SQL statements; no other code in any programming language is necessary. However, the results are displayed or printed only in certain standard formats.

What if you are writing an application program with the necessary logic in a programming language such as Visual Basic, C, C++, COBOL, Ada, and so on, and want your program to access a relational database? SQL is the standard language for relational database access. How can your application program interface with the database using the facilities of SQL? The following two methods provide database access to an application program.

Embedded SQL In this approach, SQL statements are included directly in the source code of the application program. At the points in the application program where database access is necessary, SQL statements are embedded into the program. The application program directly accesses the database using SQL statements. A special precompiler replaces the SQL statements in the application programs written in a host programming language with database functions.

Figure 13-13 shows a sample program written in C with embedded SQL. Observe how standard SQL statements are embedded at the appropriate points in the program.

Application Programming Interface (API) In this alternative approach, the application programmer is provided with a set of standard functions that can be invoked or called from the application program. This type of call-level interface allows the application program to access the database. These APIs enable the application program to access the database as do the embedded SQL statements in the other approach. However, this approach does not require any precompilation of the program.

An initial function call in the application program establishes the connection to the database. Then, at the point in the program where database access is required, a call is made to the appropriate database function to process the relevant SQL statements passed on in the call.

Figure 13-14 illustrates how an application program uses an SQL call-level interface (CLI).

Although this approach does away with precompilation, unfortunately, each database vendor's API is specific to that vendor's DBMS. This is the case even though all the calls result in the execution of standard SQL statements. No standards institution has established a common interface. However, Microsoft's Open Database Connectivity (ODBC), a SQL-based API, has become the industry standard.

QUERY PROCESSING

As you have seen, SQL provides a high-level data manipulation language to write queries and obtain the desired data from a relational database. You construct a query with SQL statements. You can either execute a query interactively or embed a query in a host language for execution at appropriate points in the application program. Query processing is a major consideration in the implementation of database systems.

```
EXEC SQL BEGIN DECLARE SECTION ;
        int depno ;                                  /* department number */
        int empno ;                                  /* employee number */
        varchar depname (20) ;                       /* department name */
EXEC SQL END DECLARE ;
/* SQL communication area and declaration of exceptions */
EXEC SQL INCLUDE SQLCA ;
EXEC SQL WHENEVER NOT FOUND GOTO errA ;
EXEC SQL WHENEVER SQLERROR GOTO          errB ;

/* MAIN PROGRAM */
main ( )
{
        empno = 5555 ;
        strcpy (depname.arr, "FINANCE") ;
        dname.len = strlen (depname.arr) ;

        EXEC SQL SELECT DeptNo INTO   :depno
        FROM DEPARTMENT
        WHERE          DeptName =    :depname ;

        EXEC SQL UPDATE EMPLOYEE
        SET DeptNo =   :depno
        WHERE EmployeeNo =  :empno ;

        printf ("Transfer of employee %d to department %s complete. \n",
                        empno, depname.arr) ;

        EXEC SQL COMMIT WORK ;
        exit (0) ;

        errA:
                        printf ("Department named %s not found.   \n, dname.arr) ;
                        EXEC SQL ROLLBACK WORK ;
                        exit (1)

        errB:
                        printf ("SQL error – no updates to database made. \n") ;
                        EXEC SQL ROLLBACK WORK ;
                        exit (1)
}    /* -------- END --------    */
```

Figure 13-13 Sample program with embedded SQL.

What happens when you write and run an SQL query? How are the various SQL statements executed? Are they verified by the DBMS before execution begins? Are they executed in the order in which the statements are translated? If so, will the natural sequence in the way a query is written be the optimal method for execution?

In earlier database systems like the hierarchical or network database systems, the application programmer writing a query in a procedural language was able to determine and direct the flow of the execution steps. A nonprocedural language such as SQL does not consist of sequential execution steps; the programmer is not in control of how a query must be executed. So it becomes the responsibility of the DBMS to examine each query, analyze it, and come up with the best possible sequence and plan for its execution. The optimal plan for the execution of a query must have

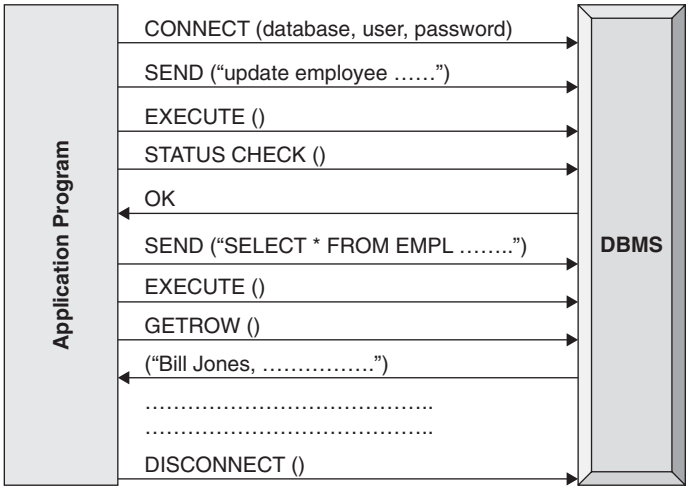


Figure 13-14 Sample program with SQL CLI.

minimum cost in terms of resource utilization and must result in minimum response time for the query.

Relational DBMSs translate a nonprocedural SQL data request into an optimal execution plan using computer science optimization techniques. We will briefly examine the various steps necessary to parse, analyze, prepare an optimal query plan, and execute an SQL query. You will gain an insight into the mechanics of query processing and understand how various components of a relational DBMS work together to process a query. Query optimization and execution techniques are major and extensive topics. For our purposes, we need not go into the complexities of the methods and algorithms. You need a broad and general view of the topics.

Query Processing Steps

Consider what happens to a query once it is written and presented for execution. What are the distinct steps, and what functions does DBMS carry out in each of these steps? How do these steps finally produce the result of the query in an optimal manner?

Let us begin with a diagram. Figure 13-15 highlights the major steps and indicates the functions in each step.

Note the following highlights of each step shown in the figure and the functions of each DBMS component:

Scanning. Carefully examine the text of the query and identify the language components such as SQL keywords, names of relations, names of attributes, built-in functions, and operators.

Parsing. Verify that the query syntax conforms to SQL syntax grammar.

Validating. Check and validate relation and attribute names and verify the semantics of the query.

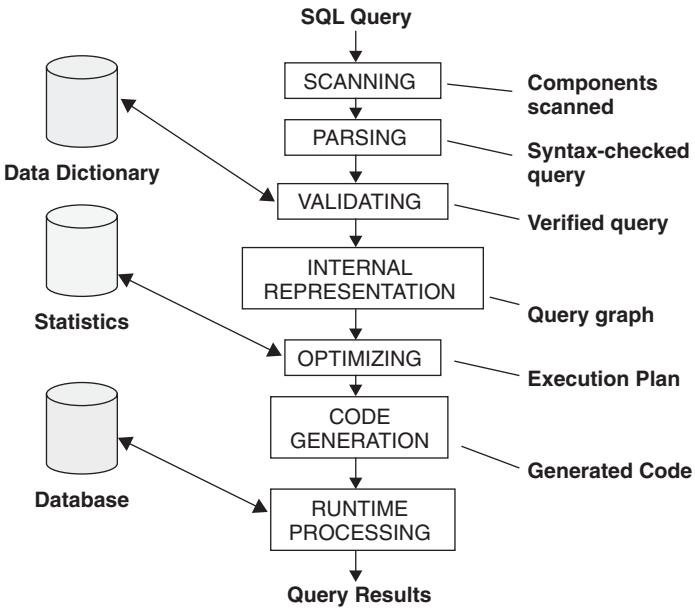


Figure 13-15 Query processing steps.

Internal representation. Create an internal representation or data structure for the query in the form of a query tree or query graph. Usually this data structure consists of translations of query components into relational algebra operations forming a query tree or query graph.

Optimizing. Examine alternative execution strategies for the query, choose an optimal strategy, and produce an execution plan.

Code generation. Generate code for executing the plan produced by the optimizer.

Runtime processing. Run the query code, compiled or interpreted, to produce the query result.

The Query Optimizer

In a relational database system, the query optimizer plays a vital role by enhancing performance a great deal. If every query is translated and executed in the haphazard manner in which it is written, database access tends to be slow and overall performance suffers. User productivity diminishes in such an environment.

Figure 13-16 presents the overall function of the query optimizer.

First, the query optimizer must analyze the query and determine the various ways in which the query may be executed. In the best of conditions, the query optimizer must consider every possible way a query may be executed. Let us say that a query consists of just three operations: SELECT, JOIN, and PROJECT. Even this simple

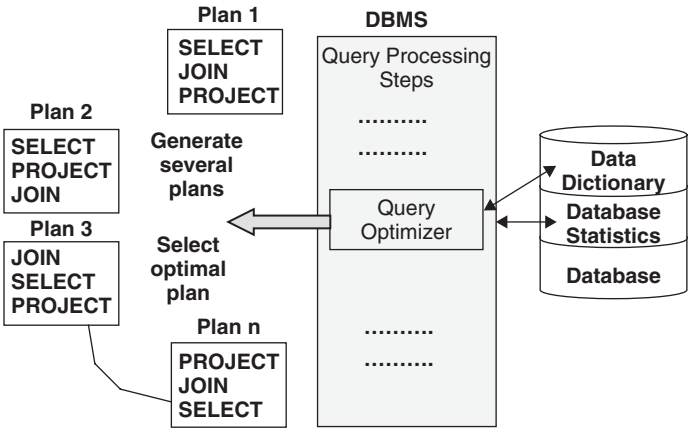


Figure 13-16 Query optimizer: overall function.

query may be executed in six different sequences. In practice, however, queries contain several operations with complex selection and join conditions. Therefore, it is not practical to review every possible way a query may be executed. Query optimizers only consider a feasible subset of all the possible ways.

Next, the query optimizer must have a proven method for choosing the best execution strategy from the competing strategies. The optimizer adopts an optimization technique to evaluate each execution plan. On the basis of the optimization technique, the optimizer selects the best possible query plan.

We will briefly scrutinize two common techniques used by the query optimizer for selecting the optimal execution plan. One technique adopts a heuristic approach; the other approach utilizes cost comparisons. These techniques examine the query tree for each of the possible ways a query may be executed and then select the tree that represents the optimal execution plan.

Heuristic Approach

If you consider a single query, it can be transformed into several different distinct sets of relational algebra operations. Each set of these operations comprises a relational algebraic expression. When you are able to transform a single query into several distinct algebraic expressions, in effect, all these expressions are equivalent and expected to produce the same result. Now the question arises: Which of the several algebraic expressions will produce the result in the most optimal manner using the minimum resources and consuming the least time?

As a consequence of parsing the query, an initial query tree representing the SQL query is constructed. This version of the query tree just represents the initial transformation of the query into a relational algebraic expression without any optimization. The optimizer must now transform this initial relational algebraic expression or its representation as a query tree into a more efficient expression or tree. The process of transformation continues until the most efficient expression or tree emerges.

How does the optimizer proceed to refine the query tree and make it more and more efficient? In the heuristic approach, the optimizer applies heuristic rules to convert the initial tree into a more efficient tree in each step. Therefore, this approach is also known as rule-based optimization.

Here are a few examples of good heuristic practices for making an initial relational algebraic expression or query tree more efficient:

- Perform selection operations first. This means moving selection operations down the query tree. Again, apply the most restrictive selection operation earliest.
- Combine a Cartesian product operation and a subsequent selection operation and replace them with a join operation.
- Perform projection operation as early as possible. This means moving projection operation down the query tree.
- Compute repeating expressions once, store the result, and reuse the result.

Now, let us see how rules such as these are applied in the heuristic approach. Use Figure 13-11 to consider the following query:

```
SELECT   PatientName
FROM     PATIENT P, TREATMENT T, DOCTOR D
WHERE    DoctorName = "Ruby Ross"
         AND D.DoctorID = T.DoctorID
         AND P.PatientNo = T.PatientNo
         AND DOB > 12DEC1952
```

Figures 13-17 and 13-18 illustrate the consecutive steps in the transformation process of the query trees for this query using the heuristic approach.

Follow optimization steps 1 through 5. Note how, in step 1, the initial query tree will produce a large result set from the PRODUCT operations on the PATIENT, DOCTOR, and TREATMENT relations. Therefore, this initial query tree is very inefficient. Note, however, that the query needs only one row from the DOCTOR relation and only those PATIENT relation rows where the date of birth is after Dec. 31, 1952. So the next version of the query tree, in step 2, moves the SELECT operations down to be executed first. A further improvement, in step 3, is achieved by switching the order of these two SELECT operations because the SELECT operation on DOCTOR relation will retrieve only one record. Proceed further and examine the improvements in steps 4 and 5.

Cost-Based Optimization

In the heuristic approach, the optimizer applies a number for rules to arrive at the optimal query tree for execution. The optimizer selects the appropriate rules for a particular query and applies the rules in the proper sequence as the process of query tree transformation continues. At the end of each step when a more efficient query tree emerges, you do not know exactly how much better the newer version of the

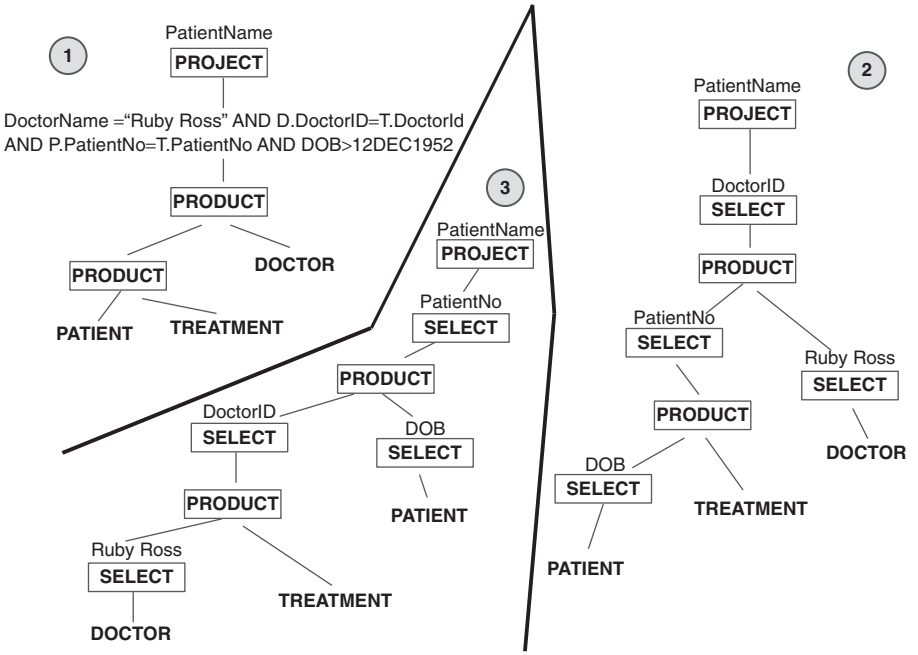


Figure 13-17 Heuristic approach: optimization steps—Part 1.

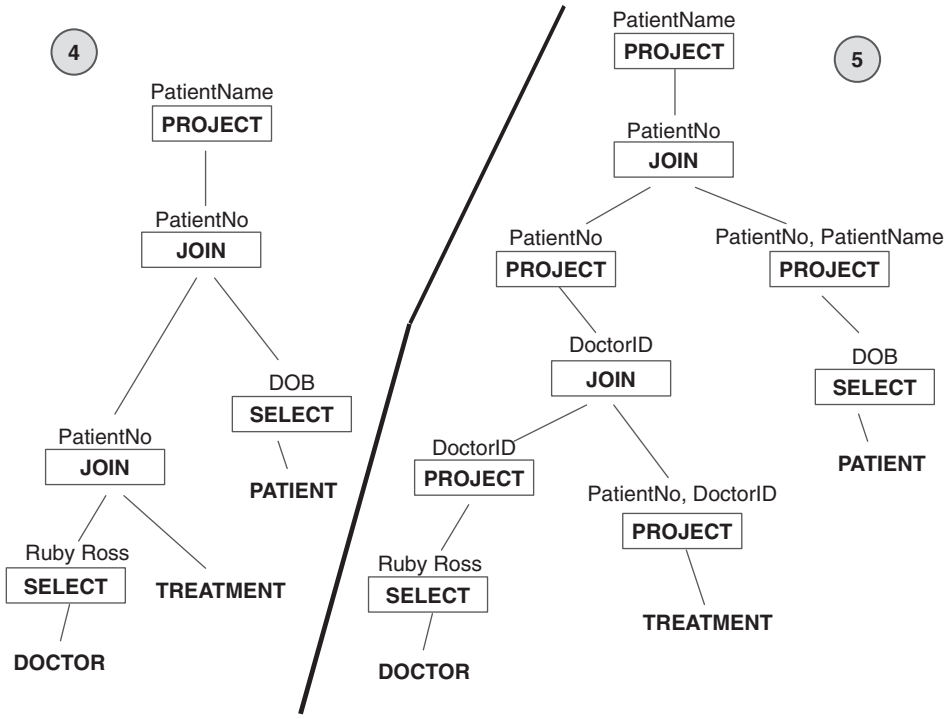


Figure 13-18 Heuristic approach: optimization steps—Part 2.

query tree is. The optimizer does not compare any numbers between tree versions to justify each transformation step.

On the other hand, the underlying principle in cost-based optimization is the ability to compute cost of resources for each version of the tree, compare the costs, and then choose the tree version that is optimal. However, you know that each query may be represented by an enormous number of versions of the query tree. It is not, therefore, practical to compute the costs for every possible version of the query tree and make the comparisons. The query optimizer limits the number of versions of the query tree by rejecting the unpromising ones.

For computing query costs, the optimizer must determine the cost components of a query and the actual calculation of cost for each component. We will briefly examine these two aspects of cost-based optimization before moving on to another topic.

Cost Components Although the cost for accessing data storage constitutes the primary cost for query execution, the total cost includes the following factors:

Cost to store data. For storing any intermediate result sets during query execution—could be substantial for queries involving large data volumes.

Cost to access data storage. For searching and reading data blocks from disk storage with or without indexes and for writing data blocks to disk storage.

Cost for sorting and merging data. For sorting and merging during query execution and for performing any heavy-duty computations.

Cost for memory buffers. For allocation and usage of numerous and large memory buffers.

Cost for data transmission. For data communication between user machine and database server, wherever these are.

Cost Computation In the calculation of the cost for query execution, the cost of data access is by far the largest element. Reduction of the cost of data access through query optimization produces the biggest payoff. Query optimizers basically concentrate on selection of the best query plan by comparing costs of data access.

The data access cost for a particular query depends on a number of factors. If the query accesses a CUSTOMER table, pertinent data for cost calculation include information about the number of rows in the table, number of data blocks for that relation, indexes, and so on. The query optimizer has to get these statistics to compute data access costs every time a query is executed. Query execution will incur a large overhead if the optimizer has to spend too much time to calculate the costs for many alternative plans every execution.

Therefore, this approach to query optimization is more suitable for compiled queries where calculations and optimization could be done at compile time and the resulting code based on optimal execution plan is stored. Then at runtime, recalculation and reoptimization are not necessary every time the query is run. For interpreted queries, all of this calculation has to be redone every time the query is run.

What about the statistics for calculating the costs? Where are they stored? Typically, the data dictionary or system catalog stores the following statistics to be used for cost calculation:

Data cardinality. Number of rows for each relational table.

Data volume. Number of data blocks for each relational table.

Index cardinality. Number of distinct values for each index.

Index height. Number of nonleaf levels in each B-tree index.

Index range. Current minimum and maximum values of the key for each index.

As you know, day-to-day database transactions will be affecting these statistics for each relation continually. The number of rows of a particular relation is likely to change, as are other statistics about the relation. As the statistics change, so does the calculation of the access costs for queries on given relations. If the statistics for a given relation have to be changed by every relevant transaction, such an arrangement will slow the transactions down. Therefore, the general practice is to run special utility jobs for updating the statistics. When the statistics are updated, any affected, compiled queries must be reoptimized

To examine the principles of cost-based optimization, let us consider a query discussed above.

```
SELECT SERVICE.ServiceCode, ServiceDesc
   FROM SERVICE, BILLING
      WHERE SERVICE.ServiceCode = BILLING.ServiceCode and
         PatientNo = 224466
```

Figure 13-19 presents two versions of the query tree for executing this query. Note the differences between the two tree versions. Even with a cursory examination, the cost of running version 1 of the query is seen to be much larger because of the large product operations. Cost-based optimizers calculate the costs for performing each operation as you move up the query tree. The cost for each type of operation depends on a number of factors including the types of indexes available. Just note how the query optimizer first comes up with a reasonable set of query tree options and then decides on the optimal tree after calculating the costs of executing each tree version. The details of how these cost calculations are performed are beyond the scope of our discussion.

DATABASE SYSTEM DEPLOYMENT

Let us trace back and find our place in the database development life cycle. The life cycle begins with planning and feasibility study. Then you gather the information requirements and produce the requirements definition document. Next, in the design phase, you complete the logical design followed by the physical design of the

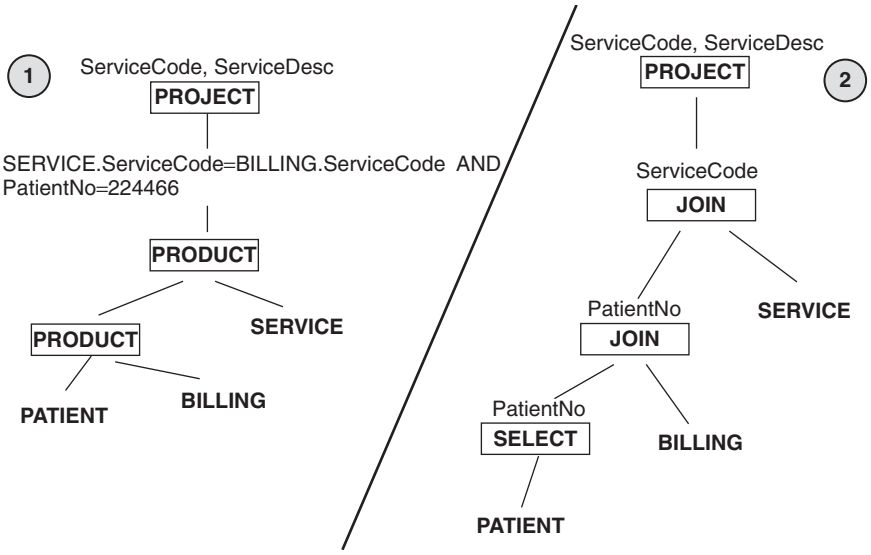


Figure 13-19 Sample query: query tree versions.

database system. We covered physical design in sufficient detail in Chapter 12. Now you are ready for implementation and deployment of the database system.

In the deployment phase, you make the database system ready and available to users. The final database system must contain data to satisfy the information requirements as defined in the requirements definition phase and then used to perform the logical and physical design.

We will now review the major tasks for deployment. Access to the database through application programs must be established. Storage space for files must be allocated. Data dictionary entries must be made. Database files must be populated with initial data. Let us highlight the major tasks and also discuss special issues relating to implementation and deployment of the database system.

Deployment Tasks

Let us continue from the end of the physical design step of the design phase. As you will note, the design has been verified and now you are ready to implement the physical design as the database system. We will list and highlight the major deployment tasks. In later subsections, we will also consider special issues relating to the deployment phase.

Here is a list of major deployment tasks:

Install DBMS. As you know, the physical design took the target or selected database management system into consideration. Now, set all the initial parameters and complete the installation of the selected DBMS.

Build data dictionary. Using the data definition facility provided in the selected DBMS—usually a version of SQL—create the data dictionary entries to define the relations, attributes, key constraints, indexes, and other constraints.

Provide application interfaces. Install and test software interfaces to provide links for application programs written in host languages to access the database.

Complete final testing. Test a final set of queries and application programming interfaces. Obtain user acceptance.

Complete data conversions. Special programs are written to convert data from the old data systems in the organization and to populate the new database. Complete testing of the data conversion programs.

Populate database. Use the outputs of the data conversion programs to populate the new database. Special programs may have to be written to use these outputs and add data to the new database. Alternatively, you may use these outputs to create sequential files with exact layouts of the database tables. Such files are called load image files. DBMS provides utility programs to load the database with the initial data from load image files. Initial data loading may be inordinately long if you create all the indexes during the load process. It is customary to suspend creation of indexes during the load process and to create the indexes after the loading is complete.

Build indexes. After the initial data load, build primary and secondary indexes.

Get user desktops ready. Install all system and other software needed at each desktop. Deployment of the database may become an opportunity to upgrade user desktops. Test each user machine including logging on to the database system.

Complete initial user training. Train users on database concepts, the overall database content, and the access privileges. If applicable to your environment, train power users in creating and running queries.

Institute initial user support. Set up support to assist users in basic usage and to answer questions on database access and performance.

Deploy in stages. Consider opening up the database system in well-defined stages to the various user groups. This approach is especially recommended if this is the first database system for the organization.

Implementation in Centralized Architecture

Earlier computer systems based on mainframes centralized all transaction processing. User applications were run on mainframes; database functionality was provided in a centralized architecture. User terminals connected the users to the mainframe and the centralized database systems. These terminals did not possess innate processing power; they were just display terminals. Data manipulation was done remotely at the mainframe, and the results were propagated to the display terminals.

Figure 13-20 illustrates a centralized DBMS architecture.

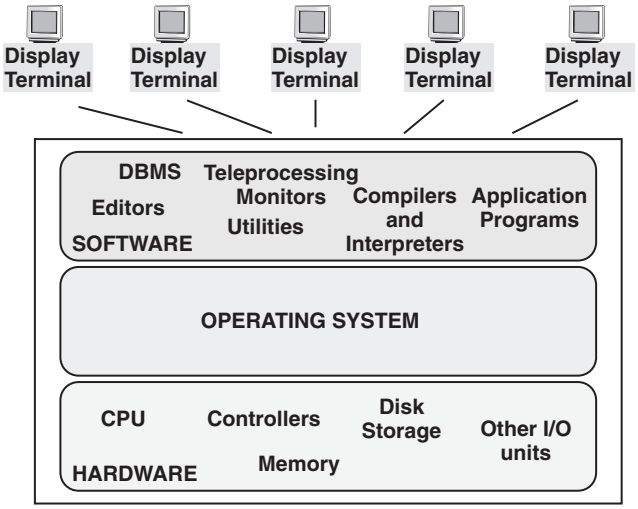


Figure 13-20 Centralized DBMS architecture.

Implementation in Client/Server Architecture

You are probably familiar with client/server computing. However, let us first review the client/server computing model and briefly describe its features. Here is a broad description:

- Client/server architecture implies cooperative or distributed processing. Two or more systems are linked through network connections.
- There could be one or many servers on the network—print server, file server, communications server, and relational database server. However, to the user signing on to the computer system, the system appears to be a unified system.
- The client system executes some parts of the application and provides the user interface. The client system requests services from the various servers.
- The server system provides services requested by client systems. Typically, a relational DBMS engine runs on a server machine.

Figure 13-21 illustrates a simple client/server architecture at the logical level. Note the various client machines attached to a communications network in a centralized configuration.

Although many database management systems started out as mainframe versions, now every vendor has adapted its DBMS for client/server architecture. User interface and application programs used to reside on the mainframe computer system. Now, these are the first programs to be moved to the client side in a client/server configuration. Query and transaction processing performs on the server side. The database component resides on the server side.

In a typical client/server setup, application programs and user interface software run on the client machines. When an application program on the client side requires a database access, it establishes a connection to the DBMS on the server side. When

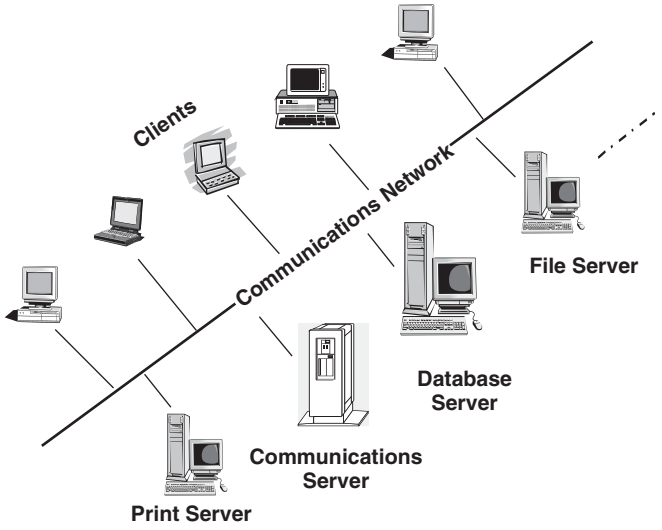


Figure 13-21 Logical client/server architecture.

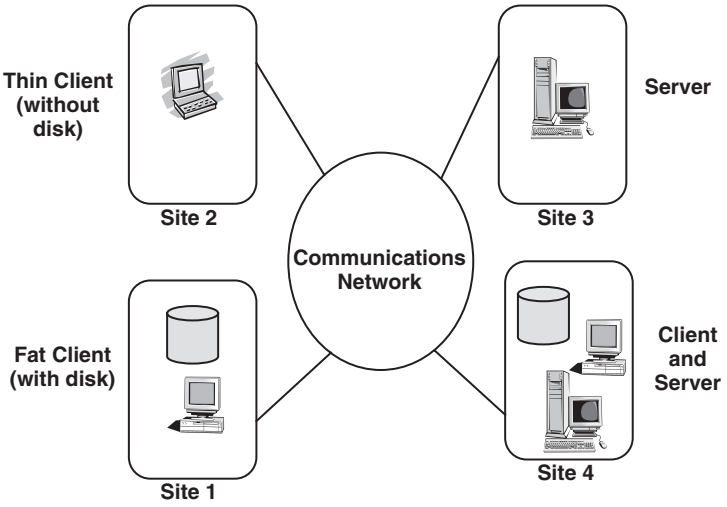


Figure 13-22 Physical client/server architecture configurations.

both client and server machines have the necessary software, ODBC, the current standard, provides an Application Programming Interface (API) to call the DBMS from the client side. Most DBMS vendors provide ODBC drivers for their databases. Query results are sent to the client machine, which formats and displays the results as necessary. Similarly, JAVA client programs can access the DBMS through another interface known as JDBC.

Figure 13-22 provides a sample of physical client/server architecture configurations. Note the different configurations at various sites.

CHAPTER SUMMARY

- Database implementation includes definition of data structures, enabling of database usage, optimal processing of data requests, and other ancillary tasks to prepare users.
- Languages are essential for implementation of a database system. These languages serve two primary purposes—storage of data and usage of data.
- The data definition language (DDL) component defines the data structures and supports the storage of data.
- The data manipulation language (DML) component enables data retrieval and data maintenance. Data maintenance consists of addition, deletion, and modification of data.
- Earlier relational languages include Query-by-Example (QBE), Quel, and Datalog. These languages contain DDL and DML components.
- Structured Query Language (SQL), a command-driven, declarative, nonprocedural language, has evolved as the standard for relational database systems.
- Both ANSI and ISO have adopted and standardized SQL. The process of standardization and enhancing SQL continues.
- As a DDL, SQL provides statements for the specification of logical and physical components for the relational database. SQL supports a variety of data types.
- Data retrieval in SQL is accomplished through an SQL query made up of three simple clauses or commands—SELECT, FROM, and WHERE.
- SQL has provisions to grant and revoke data access privileges to users. Access privileges may be granted to users to perform only specific functions such as data retrieval, data addition, data update, and so on.
- Data access from an application program to a relational database is generally accomplished in two ways: embedding SQL statements in host language programs or using an application programming interface (API) consisting of standard database functions.
- Query processing consists of several steps including the important step of optimizing the query. Two common approaches for query optimization are heuristic or rule-based optimization and cost-based optimization.

REVIEW QUESTIONS

1. What is the meaning of implementation of a data model? Which two primary purposes are accomplished by implementation?
2. How do languages enable implementation of a data model? What is the role of languages in implementation?
3. Distinguish the functions: data retrieval, data maintenance, and data control.
4. List any four of the overall features of SQL.
5. Name any five data definition statements in SQL. Describe the purpose of each.

6. List and explain any four data types supported by SQL.
7. What are the three primary clauses in an SQL query? Describe the function of each clause.
8. How is data modification done in SQL? Describe with an example.
9. List the steps in query processing. Describe the tasks carried out in two of these steps.
10. What is cost-based query optimization? How does the optimizer compute costs?

EXERCISES

1. Match the columns:

1. query validating	A. eliminate duplicates in result
2. delete row	B. rule-based
3. distinct specification in SQL	C. directly access database
4. QBE	D. call-level interface to database
5. float	E. verify relation names
6. index height	F. verify query semantics
7. heuristic query optimization	G. data type in SQL
8. API	H. used by query optimizer
9. query parsing	I. data maintenance function
10. embedded SQL	J. adopts a visual approach

2. Create a relational data model for a doctor's office consisting of the following objects: PATIENT, DOCTOR, VISIT, SERVICE, and BILLING. Using SQL, code data definition statements to create the schema for this database.
3. Write five different SQL queries using the above DOCTOR database. Explain the functions of the SQL statements in each of the queries.
4. Write five different SQL data control commands granting and revoking access privileges to the above DOCTOR database. Explain the purpose of each command.
5. You are the database administrator in a database project for an airline company. Write a memo to your project manager describing the functions of the query optimizer of your DBMS. Discuss the two common approaches, and state your preference with reasons.