UNIT 3

INSERT, DELETE, and UPDATE Statements in SQL

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE.

The INSERT Command

INSERT is used to add a single tuple to a relation.We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command.

INSERT INTO EMPLOYEE **VALUES** ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest, Katy, TX', 'M', 37000, '653298653', 4);

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple.

INSERT INTO EMPLOYEE (Fname, Lname, Dno, Ssn) **VALUES** ('Richard', 'Marini', 4, '653298653');

The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time.

DELETE FROM EMPLOYEE
WHERE Lname='Brown';

DELETE FROM EMPLOYEE
WHERE Ssn='123456789';

DELETE FROM EMPLOYEE
WHERE Dno=5;

DELETE FROM EMPLOYEE;

A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table. We must use the DROP TABLE command to remove the table definition.

The UPDATE Command

The **UPDATE** command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation.
SET clause in the UPDATE command specifies the attributes to be modified and their new values.

Ex: to change the location and controlling department number of project number 10 to 'Bellaire' and 5

UPDATE PROJECT
SET Plocation = 'Bellaire', Dnum = 5
WHERE Pnumber=10;

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the department no=5 a 10 percent raise in salary .

UPDATE EMPLOYEE
SET Salary = Salary * 1.1
WHERE Dno = 5;

Views (Virtual Tables) in SQL
Concept of a View in SQL
view in SQL terminology is a single table that is derived from other tables. These other tables can be *base tables* or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

Specification of Views in SQL
, the command to specify a view is **CREATE VIEW**. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view.

CREATE VIEW WORKS_ON1
AS SELECT Fname, Lname, Pname, Hours
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn=Essn **AND** Pno=Pnumber;

we did not specify any new attribute names for the view WORKS_ON1(although we could have); in this case,WORKS_ON1 *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.

WORKS_ON1

Fname	Lname	Pname	Hours

CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_sal)
AS SELECT Dname, **COUNT** (*), **SUM** (Salary)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber=Dno
GROUP BY Dname;

Renaming the View : explicitly specifies new attribute names for the view DEPT_INFO
one of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism.
A view is supposed to be *always up-to-date*; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes.
If we do not need a view any more, we can use the **DROP VIEW** command to dispose of it. For example, to get rid of the view V1, we can use the SQL statement in V1A:

V1A: DROP VIEW WORKS_ON1;

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a *single table* without any *aggregate functions* can be mapped to an update on the underlying base table under certain conditions.

To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith'from 'ProductX' to 'ProductY'. This view update is shown in UV1:

UV1: UPDATEWORKS_ON1
SET Pname = 'ProductY'
WHERE Lname='Smith' **AND** Fname='John'
AND Pname='ProductX';

UPDATEWORKS_ON
SET Pno= (**SELECT** Pnumber
FROM PROJECT
WHERE Pname='ProductY')
WHERE Essn **IN** (**SELECT** Ssn
FROM EMPLOYEE
WHERE Lname='Smith' **AND** Fname='John')

AND
Pno= (**SELECT** Pnumber
FROM PROJECT
WHERE Pname='ProductX');
(b)**: UPDATE**PROJECT **SET** Pname = 'ProductY'
WHERE Pname = 'ProductX'

- **Query modification:** present the view query in terms of a query on the underlying base tables
  - disadvantage: inefficient for views defined via complex queries (especially if additional queries are to be applied to the view within a short time period)

- View materialization: involves physically creating and keeping a temporary table
  - assumption: other queries on the view will follow
  - concerns: maintaining correspondence between the base table and the view when the base table is updated
  - strategy: incremental update

- Update on a single view without aggregate operations: update may map to an update on the underlying base table
- Views involving joins: an update *may* map to an update on the underlying base relations
  - not always possible
- Views defined using groups and aggregate functions are not updateable
- Views defined on multiple tables using joins are generally not updateable
- WITH CHECK OPTION: must be added to the definition of a view if the view is to be updated
  - to allow check for updatability and to plan for an execution strategy

Specifying General Constraints as Assertions in SQL
In SQL, users can specify general constraints—those that do not fall into any of the categories .

CREATE ASSERTION statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT *
FROM EMPLOYEE E, EMPLOYEE M,
DEPARTMENT D
WHERE E.Salary>M.Salary
AND E.Dno=D.Dnumber
AND D.Mgr_ssn=M.Ssn ) );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK,which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied.
Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**. The constraint is **satisfied** by a data

Introduction to Triggers in SQL

The CREATE TRIGGER statement is used to implement such actions in SQL. Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database .Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION,5 which will notify the supervisor. The trigger could then be written as in R5 below. Here we are using the syntax of the Oracle database system.

```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
WHERE SSN = NEW.SUPERVISOR_SSN ) )
INFORM_SUPERVISOR(NEW.Supervisor_ssn,NEW.Ssn );
```

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger has three components:

The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the

keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.

2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated.

If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Database Programming: Techniques and Issues

Most database systems have an **interactive interface** where these SQL commands can be typed directly into a monitor for execution by the database system. The interactive interface is quite convenient for schema and constraint creation or for occasional ad hoc queries. However, in practice, the majority of database interactions are executed through programs that have been carefully designed and tested.

These programs are generally known as **application programs** or **database applications**,

and are used as *canned transactions* by the end users

Another common use of database programming is to access a database through an application program that implements a **Web interface**, for example, when making airline reservations or online purchases. In fact, the vast majority of Web electronic commerce applications include some database access commands.

Approaches to Database Programming

Several techniques exist for including database interactions in application programs. The main approaches for database programming are the following:

1. **Embedding database commands in a general-purpose programming language.**

In this approach, database statements are **embedded** into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string EXEC SQL, which precedes all SQL commands in a host language program. A **precompiler** or **preproccessor** scans the source program code to identify database statements and extract them for processing by the DBMS. They are replaced in the program by function calls to the DBMS-generated code. This technique is generally referred to as **embedded SQL**.

2. **Using a library of database functions.** A **library of functions** is made available to the host programming language for database calls. For example, there could be functions to connect to a database, execute a query, execute an update, and so on. The actual database query and update commands and any other necessary information are included as parameters in the function calls. This approach provides what is known as an **application programming interface** (**API**) for accessing a database from application programs.

3. **Designing a brand-new language.** A **database programming language** is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional statements are added to the database language to convert it into a full fledged programming language.

Impedance Mismatch

Impedance mismatch is the term used to refer to the problems that occur because of differences between the database model and the programming language model. For example, the practical relational model has three main constructs: columns (attributes) and their data types, rows (also referred to as tuples or records), and tables (sets or multisets of records).

The first problem that may occur is that the *data types of the programming language* differ from the *attribute data types* that are available in the data model.

Another problem occurs because the results of most queries are sets or multisets of tuples (rows), and each tuple is formed of a sequence of attribute values. Impedance mismatch is less of a problem when a special database programming language is designed that uses the same data model and data types as the database model.

Embedded SQL, Dynamic SQL, and SQLJ

It give an overview of the technique for how SQL statements can be embedded in a general-purpose programming language. We focus on two languages:

C and Java. The examples used with the C language, known as **embedded SQL.**

In this embedded approach, the programming language is called the **host language**. Most SQL statements—including data or constraint definitions, queries, updates, or view definitions—can be embedded in a host language program.

Retrieving Single Tuples with Embedded SQL

To illustrate the concepts of embedded SQL, we will use C as the host programming language.4When using C as the host language, an embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a **preprocessor** (or **precompiler**) can separate embedded SQL statements from the host language code. The SQL statements within a program are terminated by a matching END-EXEC or by a semicolon (;). Similar rules apply to embedding SQL in other programming languages.

```
0) int loop ;
1) EXEC SQL BEGIN DECLARE SECTION ;
2) varchar dname [16], fname [16], lname [16], address [31] ;
3) char ssn [10], bdate [11], sex [2], minit [2] ;
4) float salary, raise ;
5) int dno, dnumber ;
6) int SQLCODE ; char SQLSTATE [6] ;
7) EXEC SQL END DECLARE SECTION ;
```

Figure :C program variables used in the embedded SQL examples
Notice that the only embedded SQL commands in Figure are lines 1 and 7,which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE because they can be included in embedded SQL statements—as long as they are preceded by a colon (:).
Lines 2 through 5 are regular C program declarations.

The C program variables declared in lines 2 through 5 correspond to the attributes of the EMPLOYEE and DEPARTMENT tables The variables declared in line 6—SQLCODE and SQLSTATE—are used to communicate errors and exception conditions between the database system and the executing program.

Line 0 shows a program variable loop that will not be used in any embedded SQL statement

Connecting to the Database. The SQL command for establishing a connection to a database has the following form:

CONNECT TO <server name>**AS** <connection name>
AUTHORIZATION <user account name and password> ;
In general, since a user or program can access several database servers, several connections
can be established, but only one connection can be active at any point in time.
The programmer or user can use the <connection name> to change from the currently active connection to a different one by using the following command:
SET CONNECTION <connection name> ;
Once a connection is no longer needed, it can be terminated by the following command:

DISCONNECT <connection name> ;

```
//Program Segment E1:
0) loop = 1 ;
1) while (loop) {
2) prompt("Enter a Social Security Number: ", ssn) ;
3) EXEC SQL
4) select Fname, Minit, Lname, Address, Salary
5) into :fname, :minit, :lname, :address, :salary
6) from EMPLOYEE where Ssn = :ssn ;
7) if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8) else printf("Social Security Number does not exist: ", ssn) ;
9) prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10) }
```

Program segment E1,a C program segment with embedded SQL.

This example also illustrates how the programmer can *update* database records.When a cursor is defined for rows that are to be modified (**updated**), we must add the clause **FOR UPDATE OF** in the cursor declaration and list the names of any attributes that will be updated by the program.

Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes.

```
//Program Segment E2:
0) prompt("Enter the Department Name: ", dname) ;
1) EXEC SQL
2) select Dnumber into :dnumber
3) from DEPARTMENT where Dname = :dname ;
4) EXEC SQL DECLARE EMP CURSOR FOR
5) select Ssn, Fname, Minit, Lname, Salary
6) from EMPLOYEE where Dno = :dnumber
7) FOR UPDATE OF Salary ;
8) EXEC SQL OPEN EMP ;
9) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE == 0) {
11) printf("Employee name is:", Fname, Minit, Lname) ;
12) prompt("Enter the raise amount: ", raise) ;
13) EXEC SQL
14) update EMPLOYEE
15) set Salary = Salary + :raise
16) where CURRENT OF EMP ;
17) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
18) }
19) EXEC SQL CLOSE EMP ;
```

Database Stored Procedures and SQL/PSM

This section introduces two additional topics related to database programming. the extensions to SQL that are specified in the standard to include general-purpose programming constructs in SQL. These extensions are known as SQL/PSM (SQL/Persistent Stored Modules) and can be used to write stored procedures. SQL/PSM also serves as an example of a database programming language.

Database Stored Procedures and Functions

Stored procedures are useful in the following circumstances:

- If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.
- Executing a program at the server can reduce data transfer and communication cost between the client and server in certain situations.
- These procedures can enhance the modeling power provided by views byallowing more complex types of derived data to be made available to the database users. Additionally, they can be used to check for complex constraints that are beyond the specification power of assertions and triggers.

The general form of declaring stored procedures is as follows:

CREATE PROCEDURE <procedure name> (<parameters>)
<local declarations>
<procedure body> ;

The parameters and local declarations are optional, and are specified only if needed.

For declaring a function, a return type is necessary, so the declaration form is

CREATE FUNCTION <function name> (<parameters>)
RETURNS <return type>
<local declarations>
<function body> ;

If the procedure (or function) is written in a general-purpose programming language, it is typical to specify the language as well as a file name where the program code is stored. For example, the following format can be used:

CREATE PROCEDURE <procedure name> (<parameters>)
LANGUAGE <programming language name>
EXTERNAL NAME <file path name> ;

In general, each parameter should have a **parameter type** that is one of the SQL data types. Each parameter should also have a **parameter mode**, which is one of IN, OUT, or INOUT.

SQL/PSM: Extending SQL for Specifying Persistent Stored Modules

SQL/PSM is the part of the SQL standard that specifies how to write persistent stored modules. It includes the statements to create functions and procedures that we described in the previous section. It also includes additional programming constructs to enhance the power of SQL for the purpose of writing the code (or body) of stored procedures and functions.

In this section, we discuss the SQL/PSM constructs for conditional (branching) statements and for looping statements. These will give a flavor of the type of constructs that SQL/PSM has incorporated;20 then we give an example to illustrate how these constructs can be used.

The conditional branching statement in SQL/PSM has the following form:

IF <condition> THEN <statement list>
ELSEIF <condition> THEN <statement list>
...
ELSEIF <condition> THEN <statement list>
ELSE <statement list>
END IF ;

Specifying Constraints in SQL

Basic constraints that can be specified in SQL as part of table creation: key and referential integrity constraints

Restrictions on attribute domains and NULLs constraints on individual tuples within a relation

Specifying Attribute Constraints and Attribute Defaults

SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the primary key of each relation, but it can be specified for any other attributes whose values are required not to be NULL.

It is also possible to define a default value for an attribute by appending the clause **DEFAULT** <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute.

CREATE TABLE DEPARTMENT (. . . ,Mgr_ssn CHAR(9) NOT NULL DEFAULT '888665555', ---------- ----------)

Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition . For example, suppose that department numbers are restricted to integer numbers between 1

and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

Dnumber INT **NOT NULL CHECK** (Dnumber > 0 **AND** Dnumber < 21);

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement.For example, we can write the following statement:

CREATE DOMAIN D_NUM **AS** INTEGER

Specifying Constraints in SQL

Basic constraints that can be specified in SQL as part of table creation: key and referential integrity constraints

Restrictions on attribute domains and NULLs constraints on individual tuples within a relation.

Examples

Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

SELECT SUM (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)

FROM EMPLOYEE;

Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

SELECT SUM (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)

FROM (EMPLOYEE **JOIN** DEPARTMENT **ON** Dno=Dnumber)

WHERE Dname='Research';

Count the number of distinct salary values in the database.

SELECT COUNT (**DISTINCT** Salary)

FROM EMPLOYEE;

To retrieve the names of all employees who have two or more dependents

SELECT Lname, Fname

FROM EMPLOYEE

WHERE (**SELECT COUNT** (*)

FROM DEPENDENT

WHERE Ssn=Essn) >= 2;

For each department, retrieve the department number, the number of employees in the department, and their average salary.

SELECT Dno, **COUNT** (*), **AVG** (Salary)
FROM EMPLOYEE
GROUP BY Dno;

For each project, retrieve the project number, the project name, and the number of employees who work on that project.

SELECT Pnumber, Pname, **COUNT** (*)
FROM PROJECT, WORKS_ON
WHERE Pnumber=Pno
GROUP BY Pnumber, Pname;

For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

SELECT Pnumber, Pname, **COUNT** (*)
FROM PROJECT, WORKS_ON
WHERE Pnumber=Pno **GROUP BY** Pnumber, Pname **HAVING COUNT** (*) > 2;

For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

SELECT Pnumber, Pname, **COUNT** (*) **FROM**
PROJECT, WORKS_ON, EMPLOYEE WHERE
Pnumber=Pno **AND** Ssn=Essn **AND** Dno=5 **GROUP BY** Pnumber, Pname;

Example: For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

SELECT Dnumber, **COUNT** (*)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber=Dno **AND** Salary>40000 **AND**
(**SELECT** Dno **FROM** EMPLOYEE **GROUP BY** Dno
HAVING COUNT (*) > 5);

consider the bank database with the following tables

- *branch (branch_name, branch_city, assets)*
- *customer (customer_name, customer_street, customer_city)*
- *account (account_number, branch_name, balance)*
- *loan (loan_number, branch_name, amount)*
- *depositor (customer_name, account_number)*
- *borrower (customer_name, loan_number)*

Write an assertion to specify the constraint that the Sum of loans taken by a customer does not exceed 100,000

CREATE ASSERTION sumofloans
CHECK (100000> = ALL
SELECT customer_name,sum(amount)
FROM borrower b, loan l
WHERE b.loan_number=l.loan_number
GROUP BY customer_name);

Write an assertion to specify the constraint that the Number of accounts for each customer in a given branch is at most two

CREATE ASSERTION NumAccounts
CHECK (2 >= ALL
SELECT customer_name,branch_name, count(*)
FROM account A , depositor D
WHERE A.account_number = D.account_number
GROUP BY customer_name, branch_name);

Introduction to Triggers in SQL

A trigger is a procedure that runs automatically when a certain event occurs in the DBMS. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. The CREATE TRIGGER statement is used to implement such actions in SQL.

General form:

CREATE TRIGGER <name>

BEFORE | **AFTER** | <events>

FOR EACH ROW |FOR EACH STATEMENT

WHEN (<condition>)

<action>

A trigger has three components

Event: When this event happens, the trigger is activated Three event types : Insert, Update, Delete Two triggering times: Before the event

After the event.

Embedded SQL

The use of SQL commands within a host language is called **Embedded SQL**. Conceptually, embedding SQL commands in a host language program is straight forward. SQL statements can be used wherever a statement in the host language is allowed. SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Any host language variable used to pass arguments into an SQL command must be declared in SQL.

There are two complications:

Data types recognized by SQL may not be recognized by the host language and vice versa This mismatch is addressed by casting data values appropriately before passing them to or from SQL commands.

SQL is set-oriented

- Addressed using cursors

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 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

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. Once a cursor is declared, we can

open it (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 previous row etc by specifying additional parameters for the fetch command)

Close the cursor

Cursor 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:

We usually need to open a cursor if the embedded statement is a SELECT. we can avoid opening a cursor if the answer contains a single row

INSERT, DELETE and UPDATE statements require no cursor. some variants of DELETE

and UPDATE use a cursor.

Examples:

Find the name and age of a sailor, specified by assigning a value to the host variable c_sid, declared earlier

EXEC SQL SELECT s.sname,s.age

INTO :c_sname, :c_age **FROM** Sailaor s **WHERE** s.sid=:c.sid;

The **INTO** clause allows us assign the columns of the single answer row to the host variable c_sname and c_age. Therefore, we do not need a cursor to embed this query in a host language program.

Compute the name 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;

The query returns a collection of rows. The INTO clause is inadequate. The solution is to use a cursor:

DECLARE sinfo **CURSOR FOR**

SELECT s.sname,s.age

FROM sailors s

WHERE s.rating>:c_minrating;

Dynamic SQL

Dynamic SQL Allow construction of SQL statements on-the-fly. 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. SQL provides some facilities to deal with such situations; these are referred to as **Dynamic SQL**.

Example:

char c_sqlstring[] = {"**DELETE FROM** Sailors **WHERE** rating>5"};

EXEC SQL PREPARE readytogo **FROM** :csqlstring;

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*

The third statement executes the command

An Introduction to JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. 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 compiled to work with different DBMSs, the final executable works only with one specific DBMS.

ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language.

In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs *Without recompilation.*

The architecture of JDBC has four main components: Application

Driver manager Drivers

Data sources

Application

initiates and terminates the connection with a data source

sets transaction boundaries, submits SQL statements and retrieves the results

Driver manager

Load JDBC drivers and pass JDBC function calls from the application to the correct driver

Handles JDBC initialization and information calls from the applications and can log all function calls

Performs some rudimentary error checking

Drivers

Establishes the connection with the data source

Submits requests and returns request results

Translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard

Data sources

Processes commands from the driver and returns the results