Introduction to Database Systems

2023-Fall

3. User Interfaces and SQL Language （4/4）

Summary

- Query with one table or multiple table,
- Query with nested query or correlated nest query,
- Query with set operators,
- Query with aggregator operators,
- Query with Group By,
- Query with CAST expression,
- Query with CASE expression,
- Sub-query,
- Recursion.

SQL Language

- It can be divided into four parts according to functions.
  - Data Definition Language (DDL), used to define, delete, or alter data schema.
  - Query Language (QL), used to retrieve data
  - Data Manipulation Language (DML), used to insert, delete, or update data.
  - Data Control Language (DCL), used to control user’s access authority to data.
- QL and DML are introduced in detail in this chapter.

Data Manipulation Language

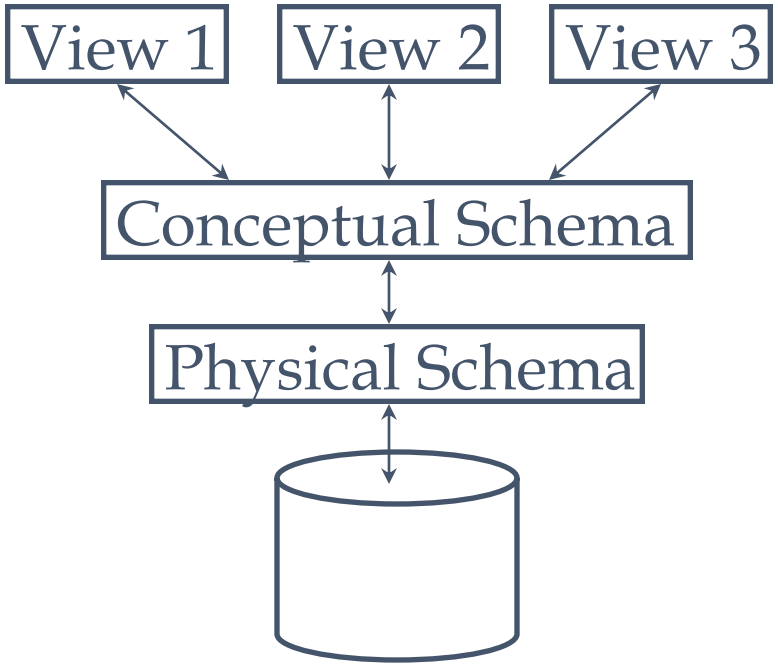
- Insert
  - Insert a tuple into a table
    - *INSERT INTO EMPLOYEES VALUES ('Smith', 'John', '1980-06-10', 'Los Angles', 16, 45000);*
- Delete
  - Delete tuples fulfill qualifications
    - *DELETE FROM Person WHERE LastName = 'Rasmussen' ;*
- Update
  - Update the attributes' value of tuples fulfill qualifications
    - *UPDATE Person SET Address = 'Zhongshan 23', City = 'Nanjing' WHERE LastName = 'Wilson';*

Data Manipulation Language

- Drop Table
  - $DROP\ TABLE\ r$
- Alter
  - $ALTER\ TABLE\ r\ ADD\ A\ D$
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.
    - All exiting tuples in the relation are assigned *null* as the value for the new attribute.
  - $ALTER\ TABLE\ r\ DROP\ A$
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.

Levels of Abstraction: ANSI-SPARC Architecture

- Many *views*, single *conceptual (logical) schema* and *physical schema*.
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.



☛ *Schemas are defined using DDL; data is modified/queried using DML.*

View in SQL

- General view
  - Virtual tables derived from base tables
  - Logical data independence
  - Security of data
  - Update problems of view
- Temporary view and recursive query
  - WITH (Common table expression)
  - RECURSIVE

Embedded SQL

- In order to access database in programs, and take further process to the query results, need to combine SQL and programming language (such as C / C++, etc.)
- Problems should be solved:
  - How to accept SQL statements in programming language
  - How to exchange data and messages between programming language and DBMS
  - The query result of DBMS is a set, how to transfer it to the variables in programming language
  - The data type of DBMS and programming language may not the same exactly.

General Solutions

- ***Embedded SQL***
  - The most basic method. Through pre-compiling, transfer the embedded SQL statements to inner library functions call to access database.
- ***Programming APIs***
  - Offer a set of library functions or DLLs to programmer directly, linking with application program while compiling.
- ***Class Library***
  - Supported after emerging of OOP. Envelope the library functions to access database as a set of class, offering easier way to treat database in programming language.

Usage of Embedded SQL (in C)

- SQL statements can be used in C program directly:
  - Begin with *EXEC SQL*, end with ';'
  - Through *host variables* to transfer information between C and SQL. Host variables should be defined begin with *EXEC SQL*.
  - In SQL statements, should add ':' before host variables to distinguish with SQL's own variable or attributes' name.
  - In host language (such as C), host variables are used as general variables.
  - Can't define host variables as Structure.
  - A special host variable, SQLCA (SQL Communication Area)

    EXEC SQL INCLUDE SQLCA
  - Use SQLCA.SQLCODE to justify the state of result.
  - Use *indicator* (short int) to treat *NULL* in host language.

Example of *host variables* defining

```
EXEC SQL BEGIN DECLARE SECTION;
   char SNO[7];
   char GIVENSNO[7];
   char CNO[6];
   char GIVENCNO[6];
   float GRADE;
   short GRADEI;                    /*indicator of GRADE*/
EXEC SQL END DECLARE SECTION;
```

Executable Statements

- ***CONNECT***
  - EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
- ***Execute DDL or DML Statements***
  - EXEC SQL INSERT INTO SC(SNO,CNO,GRADE)
    VALUES(:SNO, :CNO, :GRADE);
- ***Execute Query Statements***
  - EXEC SQL SELECT GRADE
    INTO :GRADE :GRADEI
    FROM SC
    WHERE SNO=:GIVENSNO AND
    CNO=:GIVENCNO;
- Because {SNO,CNO} is the key of SC, the result of this query has only one tuple. ***How to treat result if it has a set of tuples?***

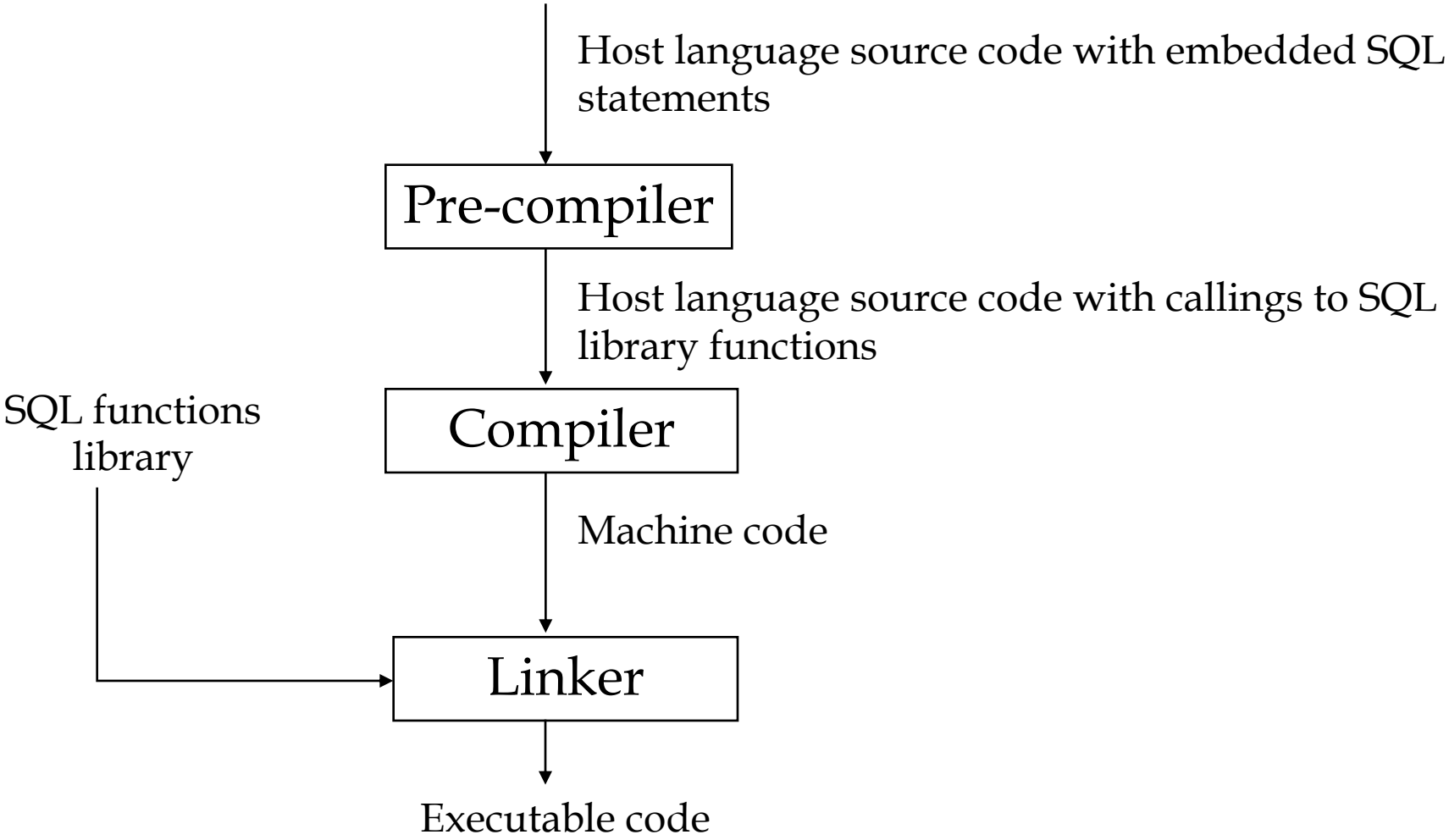
Example of Query with Cursor

```
        :
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT SNO, GRADE
        FROM SC
        WHERE CNO = :GIVENCNO;
EXEC SQL OPEN C1;
if (SQLCA.SQLCODE<0) exit(1);              /* There is error in query*/
while (1) {
   EXEC SQL FETCH C1 INTO :SNO, :GRADE :GRADEI
   if (SQLCA.SQLCODE==100)          break;
   /* treat data fetched from cursor, omitted*/
        :
}
EXEC SQL CLOSE C1;
        :
```

Cursor

1. ***Define a cursor***
   - EXEC SQL DECLARE <*cursor name*> CURSOR FOR
     SELECT …
     FROM …
     WHERE …
2. ***EXEC SQL OPEN <cursor name>***
   - Some like open a file
3. ***Fetch data from cursor***
   - EXEC SQL FETCH <*cursor name*>
     INTO :hostvar1, :hostvar2, …;
4. ***SQLCA.SQLCODE will return 100 when arriving the end of cursor***
5. ***CLOSE CURSOR <cursor name>***

Conceptual Evaluation



Dynamic SQL

- In above embedded SQL, the SQL statements must be written before compiling. But in some applications, the SQL statement can’t decided in ahead, they need to be built dynamically while the program running.
- Dynamic SQL is supported in SQL standard and most RDBMS pruducts
  - Dynamic SQL ***executed directly***
  - Dynamic SQL with ***dynamic parameters***
  - Dynamic SQL for ***query***

Dynamic SQL executed directly

- ***Only used in the execution of non query SQL statements***

```
        :
EXEC SQL BEGIN DECLARE SECTION;
char sqlstring[200];
EXEC SQL END DECLARE SECTION;
char cond[150];
strcpy( sqlstring, "DELETE FROM STUDENT WHERE ");
printf(" Enter search condition :");
scanf("%s", cond);
strcat( sqlstring, cond);
EXEC SQL EXECUTE IMMEDIATE :sqlstring;
        :
```

Dynamic SQL with dynamic parameters

- ***Only used in the execution of non query SQL statements.***
- ***Use place holder to realize dynamic parameter in SQL statement.***
- ***Some like the macro processing method in C.***

```
        :
EXEC SQL BEGIN DECLARE SECTION;
char sqlstring[200];
int birth_year;
EXEC SQL END DECLARE SECTION;
strcpy( sqlstring, "DELETE FROM STUDENT WHERE
                    YEAR(BDATE) <= :y; ");
printf(" Enter birth year for delete :");
scanf("%d", &birth_year);
EXEC SQL PREPARE purge FROM :sqlstring;
EXEC SQL EXECUTE purge USING :birth_year;
        :
```

Dynamic SQL for query

- ***Used to form query statement dynamically***

```
        :
EXEC SQL BEGIN DECLARE SECTION;
char sqlstring[200];
char SNO[7];
float GRADE;
short GRADEI;
char GIVENCNO[6];
EXEC SQL END DECLARE SECTION;
```

```
char orderby[150];
strcpy( sqlstring, "SELECT SNO,GRADE FROM SC
WHERE CNO= :c ");
printf(" Enter the ORDER BY clause :");
scanf("%s", orderby);
strcat( sqlstring, orderby);
printf(" Enter the course number :");
scanf("%s", GIVENCNO);
EXEC SQL PREPARE query FROM :sqlstring;
EXEC SQL DECLARE grade_cursor CURSOR FOR
query;
EXEC SQL OPEN grade_cursor USING :GIVENCNO;
```

Dynamic SQL for query (Cont.)

```
if (SQLCA.SQLCODE<0) exit(1);                    /* There is error in query*/
while (1) {
   EXEC SQL FETCH grade_cursor INTO :SNO, :GRADE :GRADEI
   if (SQLCA.SQLCODE==100)        break;
   /* treat data fetched from cursor, omitted*/
              :
}
EXEC SQL CLOSE grade_cursor;
              :
```

Stored Procedure

- Used to ***improve performance*** and ***facilitate users***.
- With it, user can take frequently used database access program as a procedure, and ***store it in the database after compiling***, then call it directly while need.
  - Make user convenient. User can call them directly and don’t need code again. They are reusable.
  - Improve performance. The stored procedures have been compiled, so they don’t need parsing and query optimization again while being used.
  - Expand function of DBMS. (can write script)

Example of a Stored Procedure

```
EXEC SQL
    CREATE PROCEDURE drop_student
                (IN student_no CHAR(7),
                 OUT message CHAR(30))
    BEGIN ATOMIC
                DELETE FROM STUDENT
                                WHERE SNO=student_no;
                DELETE FROM SC
                                WHERE SNO=student_no;
                SET message=student_no || 'droped';
    END;
EXEC SQL

                :
CALL drop_student(…);          /* call this stored procedure later*/
                :
```

Accessing SQL from a Programming Language

- A database programmer must have access to a general-purpose programming language for at least two reasons
  - Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
  - ***Non-declarative actions*** -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

Accessing SQL from a Programming Language (Cont.)

- There are two approaches to accessing SQL from a general-purpose programming language
  - ***A general-purpose program*** -- can connect to and communicate with a database server using a collection of functions
  - ***Embedded SQL*** -- provides a means by which a program can interact with a database server.
    - The SQL statements are translated at compile time into function calls.
    - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

JDBC

- ***JDBC*** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
        {
     try (Connection conn = DriverManager.getConnection(
          "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
            Statement stmt = conn.createStatement();
       )
     {
            … Do Actual Work ….
     }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
     }
}
```

NOTE: Above syntax works with Java 7, and JDBC 4 onwards. Resources opened in "try (….)" syntax ("try with resources") are automatically closed at the end of the try block

JDBC Code (Cont.)

- ***Update to database***

```
try{
     stmt.executeUpdate(
          "insert into instructor values('77987', 'Kim', 'Physics’, 98000)");
}
catch (SQLException sqle)
{
    System.out.println("Could not insert tuple. " + sqle);
}
```

- ***Execute query and fetch and print results***

```
ResultSet rset = stmt.executeQuery(
                                “select dept_name, avg (salary)
                                 from instructor
                                 group by dept_name”);
while (rset.next()) {
     System.out.println(rset.getString(“dept_name”) + “ ” +
                                                rset.getFloat(2));
}
```

JDBC SUBSECTIONS

- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- Callable Statements
- Metadata Features

JDBC Code Details

- ***Getting result fields:***

```
rs.getString(“dept_name”) and rs.getString(1) equivalent
if dept_name is the first argument of select result.
```

- ***Dealing with Null values***

```
int a = rs.getInt(“a”);
if (rs.wasNull()) Systems.out.println(“Got null value”);
```

JDBC Resources

- JDBC Basics Tutorial
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html

SQLJ

- **JDBC** is overly dynamic, errors cannot be caught by compiler
- ***SQLJ***: embedded SQL in Java
  - #sql iterator deptInfoIter (String dept name, int avgSal);
    deptInfoIter iter = null;
    #sql iter = { select dept_name, avg(salary) from instructor
    group by dept name };
    while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
    }
    iter.close();

ODBC

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC