Advance SQL

Accessing SQL from programming language

- Some complex queries cannot be written completely using SQL commands.
  - Such queries can be expressed in a language such as C, Java, etc.
- Non-declarative actions
  - Printing a report
  - Interacting with a user
  - Sending a results to GUI

Accessing SQL from programming language

- Two Approaches
  - Dynamic SQL
  - Embedded SQL

- Dynamic SQL
  - At runtime
  - SQL queries as character strings, submit to the database, and then retrieve the response

- Embedded SQL
  - SQL queries are identified at the compile time using a pre-processor
  - Pre-processor submits the queries to the database for precompilation and optimization
  - Replace the SQL queries with the optimized code

APIs

- Almost, each database system provides APIs to interact with the database
  - Connect to the database
  - Send SQL commands to the database
  - Fetch tuples of results one-by-one into program

- Tools
  - ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic. Other API's such as ADO.NET sit on top of ODBC
  - JDBC (Java Database Connectivity) works with Java

ODBC

- Open DataBase Connectivity
    - Standard for application programs to communicate with a database

- Applications such as spreadsheets, GUI, etc. can use ODBC

JDBC

- Java DataBase Connectivity
    - JAVA API to interact with the database supporting JDBC
- Supports metadata retrieval
    - Check whether a relation exists or not
    - Names, types of attributes

- JDBC model for communication
    - Open a connection
    - Create a statement object
    - Execute queries using the statement object to send queires and fetch results
    - Exception mechanism to handle errors

JDBC - An example

- Open connection

```
Class.forName ("oracle.jdbc.driver.OracleDriver");        //Loading database driver

Connection conn = DriverManager.getConnection ("jdbc:oracle:thin@server:port:db",
                                                                     userid, passwd);
```

- Create statement

```
      Statement stmt = conn.createStatement();
```

JDBC - An example

- Execute statement
  - executeUpdate - (insert, update, delete)

```
stmt.executeUpdate("insert into instructor values(’77987’, ’Kim’, ’Physics’, 98000)");
```

- Retrieving the Result of a Query
  - executeQuery - returns a relation   (select)

```
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 - An example

- Closing connection

```
stmt.close();
conn.close();
```

JDBC - An example

```
public static void JDBCexample(String userid, String passwd) {
try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection ("jdbc:oracle:thin@server:port:db", userid, passwd);
        Statement stmt = conn.createStatement();
        try {
                stmt.executeUpdate("insert into instructor values(’77987’, ’Kim’, ’Physics’, 98000)");
        } catch ( SQLException sqle) { System.out.println("Could not insert tuple. " + sqle); }

        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)); }
        stmt.close(); conn.close();
}
catch (Exception sqle) { System.out.println("Exception : " + sqle); }
}
```

Prepared Statement

- If some values are not known at the moment, but will be available at a later stage
  - Create statement when then values are available, ‘OR’
  - Prepare statement and supply the values when available
    - Useful, when same query with different values to be executed multiple times

```
PreparedStatement pStmt = conn.prepareStatement("insert into instructor values(?,?,?,?)");
//…..
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
//…..
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
```

SQL Injections and Prepared Statement

- Handles special character in input efficiently

  “select * from instructor where *name* = ‘ “ + name + " ’ "

  - Case 1:
    - What if user enters O‘Henry as name?
      - Syntax error
    - setString() of preparedStatement handles this scenario by inserting escape characters (\’)
  - Case 2:
    - What if user enters X‘ or ‘Y’ = ‘Y as name?

      select * from instructor where *name* = ‘X‘ or ‘Y’ = ‘Y’
    - setString()

      select * from instructor where *name* = ‘X\‘ or \‘Y\’ = \‘Y’

Metadata - Relation

- ResultSetMetaData object allows to access the metadata of a relation returned by executeQuery()
- Name and Type of the columns

```
ResultSet rs = executeQuery(“......................”);
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++)
{
      System.out.println(rsmd.getColumnName(i));
      System.out.println(rsmd.getColumnTypeName(i));
}
```

Metadata - Database

- DatabaseMetaData object allows to access the metadata of the database
- Columns

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
// getColumns(Catalog, Schema-pattern, Table-pattern, Column-Pattern)
// returns: "COLUMN_NAME", "TYPE_NAME"
```

Metadata - Database

- Tables

```
ResultSet rs = dbmd.getTables (“”, "", “%", new String[] {“TABLE”})
// getTables(Catalog, Schema, Table-name, Table-type (table, view, etc.))
// returns: "TABLE_NAME", "TABLE_TYPE"
```

- Primary Keys

```
ResultSet rs = dbmd.getPrimaryKeys( catalog, schema, tableName);
// getTables(Catalog, Schema, Table-name, Table-type (table, view, etc.))
// returns: "COLUMN_NAME", "KEY_SEQ"
```

Transactions in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically

- Turn off automatic commit on a connection
  - conn.setAutoCommit(*false*);

- Transactions must then be committed or rolled back explicitly
  - conn.commit();   or
  - conn.rollback();
- Turn off automatic commit on a connection
  - conn.setAutoCommit(*true*)

Embedded SQL

- A language in which SQL queries are embedded is referred to as a host language

  EXEC SQL <embedded SQL statement >;

  - In COBOL, the semicolon is replaced with END EXEC
  - In Java, embedding uses # SQL { …. };

Embedded SQL

- Connect to database

  EXEC SQL **connect to** *server* **user** *user_name* **using** *password*;

- Variables
  - Program variables can be used; however, they are preceded by colon (:)
    - :*credit-amount*
  - They must be declared before usage

    EXEC SQL **BEGIN DECLARE SECTION**;
    int *credit-amount*;   // Host language syntax;
    EXEC SQL **END DECLARE SECTION**;

Embedded SQL

- Relational query

  declare *c* **cursor for <***SQL query***>**;

  variable *c* identifies the query

  - Cursor: A *temporary area for work* in memory system while the execution of a statement is done.
  - Results will be computed on **open** and **fetch** commands

- Query: Find *name* and *id* of the student with credits more than *:credit-amount*

  EXEC SQL

  declare c **cursor for**

  select *ID, name* **from** *student* **where** *tot_credit* > *:credit-amount* ;

Embedded SQL

- Open
  - Evaluates the query and stores the results in temporary area

EXEC SQL **open** *c*;

- Fetch
  - From temporary area to host language variables
  - One variables for each attribute
  - Fetched one tuple at a time

EXEC SQL **fetch** *c* **into** *:si*, *:sn*;

:si and *:sn* are the program variables to store *id* and *name,* respectively.

Embedded SQL

- Close
  - The close statement causes the database system to delete the temporary relation that holds the result of the query.

    EXEC SQL **close** *c*;

- SQL communication area (SQLCA)
  - If the SQL query results in an error, the database system stores an error diagnostic in the SQLCA variables.
  - On fetch, when no further tuples remain to be processed, the character array variable SQLSTATE in the SQLCA is set to ’02000’ (meaning “no more data”)

Embedded SQL - Update/Delete

- Can update tuples fetched by cursor by declaring that the cursor is for update

```
EXEC SQL
declare c cursor for
select * from instructor where dept_name = ‘Music’
for update
```

- We then iterate through the tuples by performing fetch operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

```
EXEC SQL
update instructor set salary = salary + 1000
where current of c;
```

Function and procedure

- Procedures and functions allow “business logic” to be stored in the database and executed from SQL statements.
    - how many courses a student can take in a given semester,
    - the minimum number of courses a full-time instructor must teach in a year,
    - the maximum number of majors a student can be enrolled in,

- Rather than ensuring such business logic in the programming language, define them as the stored procedure in the database.

SQL Function

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (d_name varchar(20))
returns integer
begin
    declare d_count integer;
    select count(*) into d_count from instructor
    where instructure.dept_name = d_name
    return d_count;
end
```

SQL Function

- The function dept_count can be used to find the department names and budget of all departments with more that 12 instructors.

  select *dept_name*, *budget* **from** *department* **where** *dept_count*(*dept_name*) > 12

- Compound statement: begin … end
  - May contain multiple SQL statements between begin and end.

- **returns:** indicates the variable-type that is returned (e.g., integer)
- **return:** specifies the values that are to be returned as result of invoking the function

Table function

- The SQL standard supports functions that can return tables as results

```
create function instructors_of (dept_name varchar(20))
returns table (
        ID varchar(5), name varchar(20), dept_name varchar(20), salary numeric (8,2))
return table (
        select ID, name, dept_name, salary
from instructor where instructor.dept_name = instructor_of.dept_name);
```

- Get the instructors’ detail from finance department

```
select * from table(instructor_of(’Finance’));
```

SQL Procedure

- The dept_count function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20), out d_count integer)
begin
        select count(*) into d_count from instructor
        where instructor.dept_name = dept_count_proc.dept_name
end
```

- Keywords **in** and **out**
    - Parameter expected and parameters to return
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the call statement.

```
        declare d_count integer;
        call dept_count_proc( ‘Physics’, d_count);
```

Language constructs for function and procedure

- SQL also supports other programming language constructs, that gives it almost all the power of general-purpose language
  - while, for, etc.
- Variables can be declared using **declare** statement and assigned using **set**
- Compound statement:
  - May contains multiple SQL commands between **begin** and **end** statements
  - Local variables can also be declared
- Compound statements as transaction
  - **begin atomic** ….. **end**

Loops - While/Repeat/For

```
while boolean_expression do
        sequence of statements;
end while
```

```
repeat
        sequence of statements;
until boolean_expression
end repeat
```

```
declare n integer default 0;
for r as
        select budget from department where dept name = ‘Music‘
do
        set n = n − r.budget
end for
```

Conditional statements: if-then-else

```
if boolean expression
then
    statement or compound statement
elseif boolean expression
then
    statement or compound statement
else
    statement or compound statement
end if
```

Registers a student to a course after ensuring classroom capacity is not exceeded

```
create function registerStudent(in s_id varchar(5), …, out errorMsg varchar(100))
returns integer
begin
    declare currEnrol int;
        select count(*) into currEnrol from takes where ... // get current count
    declare limit int;
        select capacity into limit from classroom … // get capacity of course
    if (currEnrol < limit)
        begin
            insert into takes values (s_id, ……., );
            return(0); // success
        end
    set errorMsg = ’Enrollment limit reached for course’;
    return(-1); // failure
end;
```

Other constructs

- **leave** - to exit the loop (break)
- **iterate** - starts the next tuple from the beginning of the loop (continue)

- Exception handling

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
    …
    signal out_of_classroom_seats
    …
end
```

The handler here is **exit** -- causes enclosing begin..end to be exited

External language routines

- Procedural extensions to SQL have some drawbacks
  - Efficiency
  - Different database have different formats
- SQL permits the use of functions and procedures written in other languages such as C, C++, Java, etc.
  - Can be more efficient than function defined in SQL
  - Computations that cannot be carried out in SQL can be executed by these functions.

External language routines

```
create procedure dept_count_proc(in dept_name varchar(20), out count integer)
language C
external name ’ /usr/avi/bin/dept_count_proc’


create function dept_count(dept_name varchar(20))
returns integer
language C
external name ‘/usr/avi/bin/dept_count’
```

External language routines

- Drawback
  - Functions defined and compiled outside the database system may be loaded and executed with the database-system code.
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data

- Direct execution in the database system’s space is used when efficiency is more important than security.

External language routines

- Solution to security risk, **sandbox**
    - Use a safe language like Java, C#, etc. which cannot be used to access/damage other parts of the database code.
    - Sandbox allows Java, C# code to access its own memory area, and prevents them to access the memory of query execution process
        - Parameters and results communicated via inter-process communication