فصل پنجم: SQL پیشرفته

(Advanced SQL)

درس پایگاه داده
دانشگاه صنعتی نوشیروانی بابل
مهدی عمادی
m.emadi@nit.ac.ir

Outline

- Accessing SQL From a Programming Language
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features

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

- **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);
  }
}
```

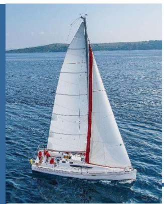
NOTE: Above syntax works with Java 7, and JDBC 4 onwards.

Resources opened in "try (….)" syntax ("try with resources") are automatically closed at the end of the try block

JDBC Code for Older Versions of Java/JDBC

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

NOTE: Classs.forName is not required from JDBC 4 onwards. The try with resources syntax in prev slide is preferred for Java 7 onwards.



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
- Other Features
- Database Access from Python

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");
```

Prepared Statement

- 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();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - NEVER create a query by concatenating strings
  - "insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + ')“
  - What if name is “D'Souza”?

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = '" + name + "'"
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = 'Y
- then the resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = 'X' or 'Y' = 'Y'
  - User could have even used
    - X'; update instructor set salary = salary + 10000; --
- Prepared stament internally uses:
  "select * from instructor where name = 'X\' or \'Y\' = \'Y'
  - Always use prepared statements, with user inputs as parameters

Metadata Features

- ResultSet metadata
- E.g.after executing query to get a ResultSet rs:
  - ResultSetMetaData rsmd = rs.getMetaData();

```
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

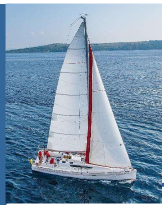
- How is this useful?

Metadata (Cont)

- **Database metadata**
- **DatabaseMetaData dbmd = conn.getMetaData();**

```
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
// The value null indicates all Catalogs/Schemas.
// The value “” indicates current catalog/schema
// The value “%” has the same meaning as SQL like clause
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
while( rs.next()) {
      System.out.println(rs.getString("COLUMN_NAME"),
                                  rs.getString("TYPE_NAME");
}
```

- **And where is this useful?**



Metadata (Cont)

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getTables: Catalog, Schema-pattern, Table-pattern,
// and Table-Type
// Returns: One row for each table; row has a number of attributes
// such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
// The value null indicates all Catalogs/Schemas.
// The value “” indicates current catalog/schema
// The value “%” has the same meaning as SQL like clause
// The last attribute is an array of types of tables to return.
//    TABLE means only regular tables
ResultSet rs = dbmd.getTables (“”, "", “%", new String[] {“TABLES”});
while( rs.next()) {
      System.out.println(rs.getString(“TABLE_NAME“));
}
```

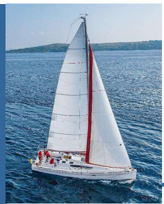
- And where is this useful?

Finding Primary Keys

- ```
  DatabaseMetaData dmd = connection.getMetaData();

  // Arguments below are:  Catalog, Schema, and Table
  // The value “”  for Catalog/Schema indicates current catalog/schema
  //  The value null indicates all catalogs/schemas
  ResultSet rs = dmd.getPrimaryKeys(“”, “”, tableName);

  while(rs.next()){
     // KEY_SEQ indicates the position of the attribute in
     // the primary key, which is required if a primary key has multiple
     // attributes
     System.out.println(rs.getString(“KEY_SEQ”),
                                    rs.getString("COLUMN_NAME");
  }
  ```
  ```



Transaction Control in JDBC

- **By default, each SQL statement is treated as a separate transaction that is committed automatically**
  - **bad idea for transactions with multiple updates**
- **Can turn off automatic commit on a connection**
  - **conn.setAutoCommit(false);**
- **Transactions must then be committed or rolled back explicitly**
  - **conn.commit();   or**
  - **conn.rollback();**
- **conn.setAutoCommit(true) turns on automatic commit.**

Other JDBC Features

- **Calling functions and procedures**
  - **CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");**
  - **CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");**
- **Handling large object types**
  - **getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively**
  - **get data from these objects by getBytes()**
  - **associate an open stream with Java Blob or Clob object to update large objects**
    - **blob.setBlob(int parameterIndex, InputStream inputStream).**

JDBC Resources

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

SQLJ

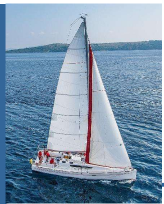
- **JDBC is overly dynamic, errors cannot be caught by compiler**
- **SQLJ: embedded SQL in Java**
  - **#sql iterator deptInfoIter (String dept name, int avgSal);**

```
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

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**



Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/1.
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement >;

  Note: this varies by language:
  - In some languages, like COBOL, the semicolon is replaced with END-EXEC
  - In Java embedding uses # SQL { …. };

Embedded SQL (Cont.)

- **Before executing any SQL statements, the program must first connect to the database. This is done using:**

  EXEC-SQL connect to *server* user *user-name* using *password*;

  Here, *server* identifies the server to which a connection is to be established.

- **Variables of the host language can be used within embedded SQL statements. They are preceded by a colon (:) to distinguish from SQL variables (e.g., :*credit_amount*)**

- **Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.**

  EXEC-SQL BEGIN DECLARE SECTION}

  int *credit-amount* ;

  EXEC-SQL END DECLARE SECTION;

Embedded SQL (Cont.)

- To write an embedded SQL query, we use the

  declare *c* cursor for <SQL query>

  statement. The variable *c* is used to identify the query
- Example:
  - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount in the host langue
  - Specify the query in SQL as follows:

```
EXEC SQL
   declare c cursor for
   select ID, name
   from student
   where tot_cred > :credit_amount
END_EXEC
```

Embedded SQL (Cont.)

- The open statement for our example is as follows:

  EXEC SQL open *c* ;

  This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable *credit-amount* at the time the open statement is executed.

- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

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

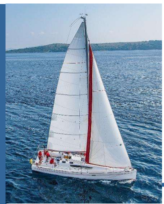
  Repeated calls to fetch get successive tuples in the query result

Embedded SQL (Cont.)

- **A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available**
- **The close statement causes the database system to delete the temporary relation that holds the result of the query.**

EXEC SQL close *c* ;

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.



Updates Through Embedded SQL

- Embedded SQL expressions for database modification (update, insert, and 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:

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

Functions and Procedures

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

Declaring SQL Functions

- 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 (dept_name varchar(20))
        returns integer
        begin
        declare d_count  integer;
                select count (* ) into d_count
                from instructor
                where instructor.dept_name = dept_name
        return d_count;
 end
```

- 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
```

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called table functions
- Example: Return all instructors in a given department

create function *instructor_of* (*dept_name* char(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*)

- Usage

    select *
    from table (*instructor_of* ('Music'))

SQL Procedures

- 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
```

- The keywords in and out are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- 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);
```

SQL Procedures (Cont.)

- Procedures and functions can be invoked also from dynamic SQL
- SQL allows more than one procedure of the so long as the number of arguments of the procedures with the same name is different.
- The name, along with the number of arguments, is used to identify the procedure.

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: begin … end,
  - May contain multiple SQL statements between begin and end.
  - Local variables can be declared within a compound statements
- While and repeat statements:
  - ```
    while boolean expression  do
              sequence of statements ;
    end while
    ```
  - ```
    repeat
              sequence of statements ;
    until boolean expression
    end repeat
    ```

Language Constructs (Cont.)

- **For loop**
  - **Permits iteration over all results of a query**
- **Example: Find the budget of all departments**

```
declare n  integer default 0;
for r  as
      select budget from department
       where dept_name = 'Music'
do
              set n = n + r.budget
end for
```

Language Constructs – if-then-else

- **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

Example procedure

- Registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - See book (page 202) for details
- Signaling of exception conditions, and declaring handlers for exceptions

declare *out_of_classroom_seats* condition
declare exit handler for *out_of_classroom_seats*
begin
…
end

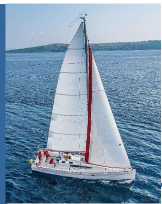
- The statements between the begin and the end can raise an exception by executing "signal *out_of_classroom_seats*"
- The handler says that if the condition arises he action to be taken is to exit the enclosing the begin end statement.

External Language Routines

- SQL allows us to define functions in a programming language such as Java, C#, C or C++.
  - Can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL\can be executed by these functions.
- Declaring external language procedures and functions

```
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 (Cont.)

- **Benefits of external language functions/procedures:**
  - **more efficient for many operations, and more expressive power.**
- **Drawbacks**
  - **Code to implement function may need to be loaded into database system and executed in the database system's address space.**
    - **risk of accidental corruption of database structures**
    - **security risk, allowing users access to unauthorized data**
  - **There are alternatives, which give good security at the cost of potentially worse performance.**
  - **Direct execution in the database system's space is used when efficiency is more important than security.**

Security with External Language Routines

- **To deal with security problems, we can do on of the following:**
  - **Use sandbox techniques**
    - **That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code.**
  - **Run external language functions/procedures in a separate process, with no access to the database process' memory.**
    - **Parameters and results communicated via inter-process communication**
- **Both have performance overheads**
- **Many database systems support both above approaches as well as direct executing in database system address space.**

Triggers

Triggers

- A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

- Triggering event can be insert, delete or update
- Triggers on update can be restricted to specific attributes
  - For example, after update of *takes* on *grade*
- Values of attributes before and after an update can be referenced
  - referencing old row as : for deletes and updates
  - referencing new row as : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
     when (nrow.grade = ' ')
     begin atomic
          set nrow.grade = null;
end;
```

Trigger to Maintain credits_earned value

- create trigger *credits_earned* after update of *takes* on (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> 'F' and *nrow.grade* is not null
  and (*orow.grade* = 'F' or *orow.grade* is null)
  begin atomic
  update *student*
  set *tot_cred*= *tot_cred* +
  (select *credits*
  from *course*
  where *course.course_id*= *nrow.course_id*)
  where *student.id* = *nrow.id*;
  end;

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use for each statement instead of for each row
  - Use referencing old table or referencing new table to refer to temporary tables (called *transition tables*) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of
    through a trigger

When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

The Power of Recursion

- **Computing transitive closure using iteration, adding successive tuples to *rec_prereq***
  - **The next slide shows a *prereq* relation**
  - **Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.**
  - **The final result is called the *fixed point* of the recursive view definition.**
- **Recursive views are required to be monotonic. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more**

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Iteration Number	*Tuples in c1*
0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done

پایان فصل پنجم