Chapter 5: Advanced SQL

Database System Concepts, 7th Ed.

Outline

- Accessing SQL From a Programming Language
- Functions
- 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
  - JDBC (Java)
  - ODBC (C, C++, Go, Python)

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 (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

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
- **Please follow the lecture 8 on CSE-302 lab.**

Functions and Procedures

Functions

- 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 OR REPLACE FUNCTION get_instructor_for_dept(d_name IN varchar2)
 RETURN NUMBER
 IS d_count NUMBER(11,2);
 BEGIN
    SELECT count(*)
    INTO d_count
    FROM instructor
    WHERE dept_name = d_name;
    RETURN d_count;
 END;

 SELECT get_instructor_for_dept('Finance') FROM dual;
```

- 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 get_instructor_for_dept (*dept_name*) > 1

PL/SQL

```
DECLARE
   dept_name varchar2(20):= 'Finance';
  item varchar2(20);
BEGIN
  FOR item IN (SELECT DISTINCT(dept_name) FROM instructor) LOOP
      IF (item.dept_name = dept_name) THEN
         dbms_output.put_line('Found ' || item.dept_name || ' department!');
      END IF;
  END LOOP;
END;
```

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

Trigger (Example)

```
-- create audit table
-- DROP TABLE instructor_insert_audit_log;
CREATE TABLE instructor_insert_audit_log(
     instructor_id varchar(5),
     user_name varchar2(64),
     created_at date
);

-- create trigger
CREATE OR REPLACE TRIGGER instructor_audit
BEFORE INSERT
   ON instructor
   FOR EACH ROW

DECLARE
   username varchar2(10);
BEGIN
   INSERT INTO instructor_insert_audit_log VALUES (:NEW.ID, user, sysdate);
END;



-- test with an insert
INSERT INTO instructor VALUES ('9876', 'me', 'Music', 50000);
SELECT * FROM instructor_insert_audit_log;
-- cleanup
-- DELETE FROM instructor WHERE name = 'me';
```

Trigger to Maintain credits_earned value

```
CREATE OR REPLACE TRIGGER credits_earned
AFTER UPDATE
   OF grade
   ON takes
   FOR EACH ROW
DECLARE
        course_credit NUMERIC(2, 0);
BEGIN
   IF ((:NEW.grade <> 'F' AND :NEW.grade IS NOT NULL) AND (:OLD.grade = 'F' OR :OLD.grade IS NULL)) THEN
      DBMS_OUTPUT.PUT_LINE('Updating total credit in student table...');
      SELECT credits INTO course_credit FROM course WHERE course_id = :NEW.course_id;
      UPDATE student SET tot_cred = (tot_cred + course_credit) WHERE id = :NEW.id;
   END IF;
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

```
CREATE TRIGGER FLIGHTS_DELETE
  AFTER DELETE ON FLIGHTS
  REFERENCING OLD_TABLE AS DELETED_FLIGHTS
  FOR EACH STATEMENT
  DELETE FROM FLIGHT_AVAILABILITY WHERE FLIGHT_ID IN
  (SELECT FLIGHT_ID FROM DELETED_FLIGHTS);
```

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

```
-- prepare our existing table
-- INSERT INTO course VALUES ('CS-401', 'Data mining', 'Comp. Sci.', 3);
-- INSERT INTO prereq VALUES ('CS-401', 'CS-347');
```

```
SELECT * FROM prereq
CONNECT BY PRIOR prereq_id = course_id
START WITH course_id = 'CS-401';
```

OR

```
WITH    required (course_id, prereq_id) AS
    (
        SELECT  course_id, prereq_id
        FROM    prereq
        WHERE   course_id = 'CS-401'
        UNION ALL
        SELECT  required.course_id, prereq.prereq_id
        FROM    required
        JOIN    prereq
        ON      required.prereq_id = prereq.course_id
    )
SELECT  *
FROM    required;
```

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

Advanced Aggregation Features

Advanced Aggregation

```sql
-- prepare table
-- create table
CREATE TABLE student_grades(
  student_id numeric(8, 0),
  student_name nvarchar2(64),
  dept_name nvarchar2(32),
  cgpa numeric(4, 2)
);

-- insert values
INSERT ALL
INTO student_grades VALUES (1, 'Tom', 'CSE', 3.4)
INTO student_grades VALUES (2, 'Leo', 'CSE', 3.3)
INTO student_grades VALUES (3, 'Chris', 'CSE', 3.6)
INTO student_grades VALUES (4, 'Michael', 'EEE', 3.8)
INTO student_grades VALUES (5, 'Quentin', 'EEE', 3.6)
INTO student_grades VALUES (6, 'Matt', 'EEE', 3.5)
INTO student_grades VALUES (7, 'Jerry', 'CE', 3.1)
INTO student_grades VALUES (8, 'David', 'CE', 3.3)
INTO student_grades VALUES (9, 'Jason', 'CE', 2.8)
INTO student_grades VALUES (10, 'Dwayne', 'CE', 3.2)
INTO student_grades VALUES (11, 'Richard', 'CSE', 3.4)
SELECT 1 FROM dual;
```

Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation

  student_grades(ID, GPA)

  giving the grade-point average of each student
- Find the rank of each student.

```
SELECT student_id,
       student_name,
       cgpa,
       dept_name,
       rank() OVER (ORDER BY cgpa desc) AS student_rank,
       dense_rank() OVER (ORDER BY cgpa desc) AS student_dense_rank
FROM student_grades;
```

- Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  - **dense_rank** does not leave gaps, so next dense rank would be 2

Ranking (Cont.)

- Ranking can be done within partition of the data.
- “Find the rank of students within each department.”

```
SELECT student_id,
       student_name,
       cgpa,
       dept_name,
       rank() OVER (PARTITION BY dept_name ORDER BY cgpa desc) AS student_rank,
       dense_rank() OVER (PARTITION BY dept_name ORDER BY cgpa desc) AS
student_dense_rank
FROM student_grades;
```

- Multiple **rank** clauses can occur in a single **select** clause.
- Ranking is done *after* applying **group by** clause/aggregation
- Can be used to find top-n results
  - More general than the **limit** *n* clause supported by many databases, since it allows top-n within each partition

Exercise

- Find the student who is ranked second in his/her department.

Exercise

- Find the student who is ranked second in his/her department.

```
WITH student_ranks AS (
    SELECT  student_id,
              student_name,
              cgpa,
              dept_name,
            rank() OVER (PARTITION BY dept_name ORDER BY cgpa desc) AS student_rank,
            dense_rank() OVER (PARTITION BY dept_name ORDER BY cgpa desc) AS student_dense_rank
    FROM student_grades
)
SELECT *
FROM student_ranks
WHERE student_rank = 2;
```

Ranking (Cont.)

- Other ranking functions:
  - **percent_rank** (within partition, if partitioning is done)
  - **cume_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  - **row_number** (non-deterministic in presence of duplicates)

SQL:1999 permits the user to specify **nulls first** or **nulls last**

```
select ID,
        rank ( ) over (order by cgpa desc nulls last) as s_rank
from student_grades
```

Ranking (Cont.)

- For a given constant *n*, the ranking the function *ntile*(*n*) takes the tuples in each partition in the specified order, and divides them into *n* buckets with equal numbers of tuples.
- E.g.,

```
select student_id,
       student_name,
       dept_name,
       cgpa,
       ntile(4) over (order by cgpa desc) as quartile
from student_grades;
```

Windowing

- Used to smooth out random variations.
- E.g., **moving average**: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”
- **Window specification** in SQL:
  - Given relation emp*(empno, ename, job, mgr, hiredate, sal, comm, deptno)*

```
SELECT ename,
       sal,
       sum(sal) OVER (PARTITION BY deptno order by sal) AS running_salary
FROM emp;

SELECT empno,
       ename,
       job,
       sal,
       sum(sal) OVER (PARTITION BY deptno ORDER BY sal DESC ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING),
       deptno
FROM   emp;

SELECT empno,
       ename,
       job,
       sal,
       sum(sal) OVER (PARTITION BY deptno ORDER BY sal DESC RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING),
       deptno
FROM   emp;
```

Windowing

- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **range between** 10 **preceding and current row**
    - All rows with values between current row value –10 to current value
  - **range interval** 10 **day preceding**
    - Not including current row

Windowing (Cont.)

- Can do windowing within partitions
- E.g., Given a relation *transaction* (*account_number, date_time, value*), where value is positive for a deposit and negative for a withdrawal
  - "Find total balance of each account after each transaction on the account"

```
select account_number, date_time,
    sum (value) over
                (partition by account_number
                order by date_time
                rows unbounded preceding)
   as balance
from transaction
order by account_number, date_time
```

Windowing (Cont.)

- Lead/Lag

```
SELECT empno,
       ename,
       job,
       sal,
       LAG(sal, 1, 0) OVER (PARTITION BY deptno ORDER BY sal desc) AS sal_prev,
       sal - LAG(sal, 1, 0) OVER (PARTITION BY deptno ORDER BY sal desc) AS sal_diff,
       deptno
FROM   emp;
```

End of Chapter 5