Lecture 7 of 42

Embedded SQL and Dynamic SQL
Notes: Assertions, Authorization, Cursors

Wednesday, 06 February 2008

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course page: http://snipurl.com/va60
Course web site: http://www.kddresearch.org/Courses/Spring-2008/CIS560
Instructor home page: http://www.cis.ksu.edu/~bhsu

Reading for Next Class:
Sections 4.5 – 4.6, p. 137 – 151, Silberschatz *et al.*, 5th edition
MySQL Primer info (to be posted on Handouts page)

Basic Query Structure: Review

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a relation.

Update of a View: Review

- Create a view of all loan data in the *loan* relation, hiding the *amount* attribute

  create view *branch_loan* **as**
  select *branch_name, loan_number*
  from *loan*

- Add a new tuple to *branch_loan*

  insert into *branch_loan*
  values ('Perryridge', 'L-307')

  This insertion must be represented by the insertion of the tuple

  ('L-307', 'Perryridge', *null*)

  into the *loan* relation

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>

- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting

  for all *X*, *P*(*X*)

  is achieved in a round-about fashion using

  not exists *X* such that not *P*(*X*)

Assertion Example

- Every loan has at least one borrower who maintains an account with a minimum balance of $1000.00

```
create assertion balance_constraint check
    (not exists (
        select *
        from loan
        where not exists (
            select *
            from borrower, depositor, account
            where loan.loan_number = borrower.loan_number
                and borrower.customer_name =
depositor.customer_name
                and depositor.account_number =
account.account_number
                and account.balance >= 1000 ))
```


Assertion Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
create assertion sum_constraint check
    (not exists (select *
                from branch
                where (select sum(amount )
                        from loan
                        where loan.branch_name =
                                branch.branch_name )
                    >= (select sum (amount )
                        from account
                        where loan.branch_name =
                                branch.branch_name )))
```

Authorization

all

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data. row into table
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

row field of tuple

Forms of authorization to modify the database schema (covered in Chapter 8): relations (tables)

- **Index** - allows creation and deletion of indices. rel.
- **Resources** - allows creation of new relations. column into rel.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

table

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name> **to** <user list>
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this in Chapter 8)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *branch* relation:

    grant select on *branch* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges
- more in Chapter 8

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>

  on <relation name or view name> **from** <user list>
- Example:

  revoke select on *branch* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantors, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- 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/I.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement > END_EXEC

  Note: this varies by language (for example, the Java embedding uses

  # SQL { …. };)

Example Query

- From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.

- Specify the query in SQL and declare a *cursor* for it

```
EXEC SQL
  declare c cursor for
  select customer_name, customer_city
  from depositor, customer, account
  where depositor.customer_name = customer.customer_name
      and depositor account_number = account.account_number
      and account.balance > :amount
END_EXEC
```

Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated

  EXEC SQL **open** *c* END_EXEC
- 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** *:cn, :cc* END_EXEC

  Repeated calls to **fetch** get successive tuples in the query result
- 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* END_EXEC

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

Updates Through Cursors

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

  declare *c* **cursor for**
  select *
  from *account*
  where *branch_name* = 'Perryridge'
  for update
- To update tuple at the current location of cursor *c*

  update *account*
  set *balance* = *balance* + 100
  where current of *c*

Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```
char *  sqlprog = "update account
                   set balance = balance * 1.05
                   where account_number = ?"
EXEC SQL prepare dynprog  from :sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using :account;
```

"bytecode" from JIT

Python, ASP

"bytecode interpreter" & JSP

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

ODBC and JDBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java

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

ODBC (Cont.)

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.

ODBC Code

```
int ODBCexample()
{
   RETCODE error;
   HENV    env;     /* environment */
   HDBC    conn;  /* database connection */
   SQLAllocEnv(&env);
   SQLAllocConnect(env, &conn);
   SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi", SQL_NTS,
     "avipasswd", SQL_NTS);
   { .... Do actual work ... }

   SQLDisconnect(conn);
   SQLFreeConnect(conn);
   SQLFreeEnv(env);
}
```


ODBC Code (Cont.)

- Program sends SQL commands to the database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
  - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
  - Arguments to SQLBindCol()
    - ODBC stmt variable, attribute position in query result
    - The type conversion from SQL to C.
    - The address of the variable.
    - For variable-length types like character arrays,
      - The maximum length of the variable
      - Location to store actual length when a tuple is fetched.
      - Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.

ODBC Code (Cont.)

- Main body of program

```
char branchname[80];
float  balance;
int  lenOut1, lenOut2;
HSTMT   stmt;
SQLAllocStmt(conn, &stmt);
char * sqlquery = "select branch_name, sum (balance)
                   from account
                   group by branch_name";
error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (error == SQL_SUCCESS) {
    SQLBindCol(stmt, 1, SQL_C_CHAR,   branchname , 80,
&lenOut1);
    SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance,       0 ,
&lenOut2);
    while (SQLFetch(stmt) >= SQL_SUCCESS) {
        printf (" %s  %g\n", branchname, balance);
    }
}
SQLFreeStmt(stmt, SQL_DROP);
```


More ODBC Features

- **Prepared Statement**
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. insert into account values(?,?,?)
  - Repeatedly executed with actual values for the placeholders
- **Metadata features**
  - finding all the relations in the database and
  - finding the names and types of columns of a query result or a relation in the database.
- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)}
  - transactions must then be committed or rolled back explicitly by
    - SQLTransact(conn, SQL_COMMIT) or
    - SQLTransact(conn, SQL_ROLLBACK)

ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
  - Core
  - Level 1 requires support for metadata querying
  - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.

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 {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(   "jdbc:oracle:thin:@aura.bell-
          labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
          ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```


JDBC Code (Cont.)

- Update to database

```
try {
    stmt.executeUpdate(  "insert into account values
                            ('A-9732', 'Perryridge', 1200)");
} catch (SQLException sqle) {
    System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery( "select branch_name,
    avg(balance)
                                        from account
                                        group by branch_name");
while (rset.next()) {
    System.out.println(
            rset.getString("branch_name") + "  " + rset.getFloat(2));
}
```

JDBC Code Details

- Getting result fields:
  - **rs.getString("branchname") and rs.getString(1) equivalent if branchname 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");
```


Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
  - more in Chapter 9
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
- These features are covered in Chapter 9 (Object Relational Databases)

Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions are particularly useful with specialized data types such as images and geometric objects
    - Example: functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```
create function account_count (customer_name varchar(20))
returns integer
begin
    declare a_count integer;
    select count (* ) into a_count
    from depositor
    where depositor.customer_name = customer_name
    return a_count;
end
```

- Find the name and address of each customer that has more than one account.

```
select customer_name, customer_street, customer_city
from customer
where account_count (customer_name ) > 1
```