



## Lecture 8 of 42

### Database Connectivity: ODBC & JDBC Notes: PS1 Solutions, MP2

Monday, 11 September 2006

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/Fall-2006/CIS560>  
Instructor home page: <http://www.cis.ksu.edu/~bhsu>

#### Reading for Next Class:

Rest of Chapter 4, p. 151 onward, Silberschatz *et al.*, 5<sup>th</sup> edition  
Sections 5.1 – 5.2, Silberschatz *et al.*, 5<sup>th</sup> edition  
JDBC Primer (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  $A_1, A_2, \dots, A_n$   
from  $r_1, r_2, \dots, r_m$   
where  $P$ 
```

- \*  $A_j$  represents an attribute
- \*  $R_j$  represents a relation
- \*  $P$  is a predicate.
- This query is equivalent to the relational algebra expression.

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

- 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



## 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 { ... }; )
```





## 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;
```

- 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.isNull()`) `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
```





## Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```
create function accounts_of (customer_name char(20)
returns table ( account_number char(10),
                 branch_name char(15),
                 balance numeric(12,2))
```

**return table**

```
(select account_number, branch_name, balance
from account
where exists (
  select *
from depositor
where depositor.customer_name =
accounts_of.customer_name
and depositor.account_number =
account.account_number))
```



## Table Functions (cont'd)

- Usage

```
select *
from table (accounts_of ('Smith'))
```





## SQL Procedures

- The *author\_count* function could instead be written as procedure:  
**create procedure** *account\_count\_proc* (**in** *title* **varchar**(20),  
**out** *a\_count* **integer**)

**begin**

```
select count(author) into a_count  
from depositor  
where depositor.customer_name =  
account_count_proc.customer_name  
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare a_count integer;  
call account_count_proc( 'Smith', a_count);
```

Procedures and functions can be invoked also from dynamic SQL

- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of

arguments differ, or at least the types of the arguments differ.



## Procedural Constructs

- 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:

```
declare n integer default 0;  
while n < 10 do  
  set n = n + 1  
end while
```

```
repeat  
  set n = n - 1  
until n = 0  
end repeat
```



## Procedural Constructs (Cont.)

- **For** loop
  - \* Permits iteration over all results of a query
  - \* Example: find total of all balances at the Perryridge branch

```
declare n integer default 0;
for r as
  select balance from account
  where branch_name = 'Perryridge'
do
  set n = n + r.balance
end for
```



## Procedural Constructs (cont.)

- Conditional statements (**if-then-else**)  
E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

```
if r.balance < 1000
  then set l = l + r.balance
elseif r.balance < 5000
  then set m = m + r.balance
else set h = h + r.balance
end if
```

- SQL:1999 also supports a **case** statement similar to C case statement
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
...
.. signal out-of-stock
end
```



## External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure account_count_proc(in customer_name
varchar(20),
                                out count integer)
```

```
language C
external name '/usr/avi/bin/account_count_proc'
```

```
create function account_count(customer_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/author_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
  - \* Use **sandbox** techniques
    - ⇒ that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
  - \* Or, 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



## Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)

```
with recursive empl (employee_name, manager_name) as
(
    select employee_name, manager_name
    from manager
    union
    select manager.employee_name,
    empl.manager_name
    from manager, empl
    where manager.manager_name =
    empl.employee_name)
select *
from empl
```



This example view, *empl*, is called the *transitive closure* of the



## 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 *manager* with itself
    - ⇒ This can give only a fixed number of levels of managers
    - ⇒ Given a program we can construct a database with a greater number of levels of managers on which the program will not work
  - \* The next slide shows a *manager* relation and each step of the iterative process that constructs *empl* 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 *manager* the view contains all of the tuples it contained before, plus possibly more



## Example of Fixed-Point Computation

<i>employee_name</i>	<i>manager_name</i>
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

<i>Iteration number</i>	<i>Tuples in empl</i>
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)





## Advanced SQL Features\*\*

- Create a table with the same schema as an existing table:  
**create table** *temp\_account* **like** *account*
- SQL:2003 allows subqueries to occur *anywhere* a value is required provided the subquery returns only one value. This applies to updates as well
- SQL:2003 allows subqueries in the **from** clause to access attributes of other relations in the **from** clause using the **lateral** construct:

```
select customer_name, num_accounts  
from customer, lateral (  
    select count(*)  
    from account  
    where account.customer_name = customer.customer_name  
)  
as this_customer (num_accounts)
```



## Advanced SQL Features (cont'd)

- Merge construct allows batch processing of updates.
- Example: relation *funds\_received* (*account\_number, amount*) has batch of deposits to be added to the proper account in the *account* relation

```
merge into account as A  
using (select *  
    from funds_received as F)  
on (A.account_number = F.account_number)  
when matched then  
    update set balance = balance + F.amount
```

