EECS 647: Introduction to Database Systems

Instructor: Luke Huan

Spring 2009
Administrative

- Sample codes of embedded SQL programming and APIs with \{C, Perl, JAVA, PHP\} are available at class webpage
- Next Monday (March 9\textsuperscript{th}), we will have class at 1005C for a lab session
Today’s Topic

- Database Architecture
- Database programming
Centralized Architectures

- Centralized DBMS: combines everything into a single system including DBMS software, hardware, application programs, and user interface processing software.
Two Tier Client-Server Architectures

- **Server**: provides database query and transaction services to client machines.
- **Client**: provide appropriate interfaces to server.
  - Run User Interface (UI) Programs and Application Programs
  - Connect to servers via network.
Client-Server Interface

- The interface between a server and a client is commonly specified by ODBC (Open Database Connectivity)
  - Provides an Application program interface (API)
  - Allow client side programs to call the DBMS.
- For PostgreSQL, we have several API examples
  - C, PERL, PHP, and JAVA
  - `$dbconn = pg_connect(
      "host= wozniak.eecs.ku.edu port=5432
      dbname=xxxx user=yyyy password=zzzz")`
Three (n) Tier Client-Server Architecture

- **Clients**
- **Intermediate layer**
  - **Web server**
  - **Application servers**
- **Database servers**

- The intermediate layer is called Application Server or Web Server, or both:
- Stores the web connectivity software and business logic for applications
- Acts like a conduit for sending partially processed data between the database server and the client.

**Additional Features**
- Security: encrypt the data at the server and client before transmission
Database Programming: Overview

- Pros and cons of SQL
  - Very high-level, possible to optimize
  - Specifically designed for databases and is called data sublanguage
  - Not intended for general-purpose computation, which is usually done by a host language

- Solutions
  - Augment SQL with constructs from general-purpose programming languages (SQL/PSM)
  - Use SQL together with general-purpose programming languages
    - Database APIs, embedded SQL, JDBC, etc.
Clarification of Terms

- John has a mySQL database server installed in his laptop. He wrote a perl script to connect to the local mySQL database, retrieve data, and print out reports about his house innovation plan.
  - Client-server model
  - Use APIs provided by mySQL to access the database
  - Perl supports mySQL API
Clarification of Terms (cont.)

- John went to his office. He has a JAVA program, which connects to a SqlServer database in his company’s intranet. He use the program to retrieve data and print out reports for his business partner.
  - Client-server model
  - Use APIs provided by SqlServer to access the database
  - Java supports SqlServer API using JDBC
Clarification of Terms (cont.)

- After job, John went to youtube.com, searched for a video of Thomas train for his children, and downloaded one
  - Client-mediate level-sever model
  - “SQL experience a plus” from a job ad linked from youtube’s web site.
Impedance mismatch and a solution

- SQL operates on a set of records at a time
- Typical low-level general-purpose programming languages operates on one record at a time

Solution: cursor

- Open (a result table): position the cursor before the first row
- Get next: move the cursor to the next row and return that row; raise a flag if there is no such row
- Close: clean up and release DBMS resources

Found in virtually every database language/API
  - With slightly different syntaxes
A Typical Flow of Interactions

- A client (user interface, web server, application server) opens a connection to a database server
- A client interact with the database server to perform query, update, or other operations.
- A client terminate the connection
Augmenting SQL: SQL/PSM

- PSM = Persistent Stored Modules
- CREATE PROCEDURE proc_name ( parameter_declarations )
  local_declarations
  procedure_body;
- CREATE FUNCTION func_name ( parameter_declarations )
  RETURNS return_type
  local_declarations
  procedure_body;
- CALL proc_name ( parameters );
- Inside procedure body:
  SET variable = CALL func_name ( parameters );
CREATE FUNCTION SetMaxGPA(IN newMaxGPA FLOAT)
    RETURNS INT
    -- Enforce newMaxGPA; return number of rows modified.
BEGIN
    DECLARE rowsUpdated INT DEFAULT 0;
    DECLARE thisGPA FLOAT;
    -- A cursor to range over all students:
    DECLARE studentCursor CURSOR FOR
        SELECT GPA FROM Student
        FOR UPDATE;
    -- Set a flag whenever there is a “not found” exception:
    DECLARE noMoreRows INT DEFAULT 0;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET noMoreRows = 1;
    … (see next slide) …
    RETURN rowsUpdated;
END
SQL/PSM example continued

-- Fetch the first result row:
OPEN studentCursor;
FETCH FROM studentCursor INTO thisGPA;
-- Loop over all result rows:
WHILE noMoreRows <> 1 DO
    IF thisGPA > newMaxGPA THEN
        -- Enforce newMaxGPA:
        UPDATE Student SET Student.GPA = newMaxGPA
        WHERE CURRENT OF studentCursor;
        -- Update count:
        SET rowsUpdated = rowsUpdated + 1;
    END IF;
    -- Fetch the next result row:
    FETCH FROM studentCursor INTO thisGPA;
END WHILE;
CLOSE studentCursor;
Other SQL/PSM features

- Assignment using scalar query results
  - SELECT INTO

- Other loop constructs
  - FOR, REPEAT UNTIL, LOOP

- Flow control
  - GOTO

- Exceptions
  - SIGNAL, RESIGNAL

... 

- For more pgSQL-specific information, check out its manual at
  http://www.postgresql.org/docs/8.2/interactive/plpgsql.html
Interfacing SQL with another language

- **API approach**
  - SQL commands are sent to the DBMS at runtime
  - Examples: JDBC, ODBC (for C/C++/VB), Perl DBI
  - These API’s are all based on the SQL/CLI (Call-Level Interface) standard

- **Embedded SQL approach**
  - SQL commands are embedded in application code
  - A precompiler checks these commands at compile-time and converts them into DBMS-specific API calls
  - Examples: embedded SQL for C/C++, SQLJ (for Java)
Example PHP

```php
// Connect to the database
$dbconn = pg_connect("host= wozniak.eecs.ku.edu port=5432
dbname=jhuan user=jhuan password=zzzzz") or
die('Could not connect: ' . pg_last_error());

//create a table
$query = "CREATE TABLE regiusers ( name varchar(50),
passwd varchar(50) )";

// Execute the Query
$query = pg_query($query);

... (see next page)

// Closing connection
pg_close($dbconn);

?>
```
Example PHP (cont.)

- //performing update
  $ldata['name'] = 'a2'; $ldata['passwd'] = 'b2';
  $res = pg_insert($dbconn, "regiusers", $ldata);

- // Performing SQL query
  $query = 'SELECT * FROM regiusers';
  $result = pg_query($query) or die('Query failed: ' . pg_last_error());

- // analyzing results
  while ($line = pg_fetch_array($result, null, PGSQL_ASSOC)) {
      foreach ($line as $col_value) {
          statement;
      }
  }

- // Free resultset
  pg_free_result($result);
Example API: JDBC

- JDBC (Java DataBase Connectivity) is an API that allows a Java program to access databases

```java
// Use the JDBC package:
import java.sql.*;
...
public class ... {
  ...
  static {
    // Load the JDBC driver:
    try {
      Class.forName("org.postgresql.Driver");
    } catch (ClassNotFoundException e) {
      ...
    }
  }
  ...
```
Connections

// Connection URL is a DBMS-specific string:
String url =
    "jdbc:postgresql://wozniak.eecs.ku.edu/jhuan";

// Making a connection:
conn
    = DriverManager.getConnection(url, username, password)
...

// Closing a connection:
con.close();

For clarity we are ignoring exception handling for now
Create an object for sending SQL statements:
Statement stmt = con.createStatement();

Execute a query and get its results:
ResultSet rs =
    stmt.executeQuery("SELECT name, passwd FROM regiusers");

Work on the results:
...

Execute a modification (returns the number of rows affected):
int rowsUpdated =
    stmt.executeUpdate("UPDATE regiusers SET passwd = '1234' WHERE name = 'sjohn'");

Close the statement:
stmt.close();
/\ Execute a query and get its results:
ResultSet rs =
    stmt.executeQuery("SELECT name, passwd FROM regiusers");

/\ Loop through all result rows:
while (rs.next()) {
    /\ Get column values:
    String name = rs.getString(1);
    String passwd = rs.getString(2);
    /\ Work on sid and name:
    ...
}

/\ Close the ResultSet:
rs.close();
Other ResultSet features

- Move the cursor (pointing to the current row) backwards and forwards, or position it anywhere within the ResultSet
- Update/delete the database row corresponding to the current result row
  - Analogous to the view update problem
- Insert a row into the database
  - Analogous to the view update problem
Prepared statements: motivation

```java
Statement stmt = con.createStatement();
for (int age=0; age<100; age+=10) {
    ResultSet rs = stmt.executeQuery
        ("SELECT AVG(GPA) FROM Student" + 
         " WHERE age >= " + age + " AND age < " + (age+10));
    // Work on the results:
    ...
}
```

- Every time an SQL string is sent to the DBMS, the DBMS must perform parsing, semantic analysis, optimization, compilation, and then finally execution.
- These costs are incurred 10 times in the above example.
- A typical application issues many queries with a small number of patterns (with different parameter values).
Prepared statements: syntax

```java
// Prepare the statement, using ? as placeholders for actual parameters:
PreparedStatement stmt = con.prepareStatement
    ("SELECT AVG(GPA) FROM Student WHERE age >= ? AND age < ?");
for (int age=0; age<100; age+=10) {
    // Set actual parameter values:
    stmt.setInt(1, age);
    stmt.setInt(2, age+10);
    ResultSet rs = stmt.executeQuery();
    // Work on the results:
    ...
}
```

- The DBMS performs parsing, semantic analysis, optimization, and compilation only once, when it prepares the statement
- At execution time, the DBMS only needs to check parameter types and validate the compiled execution plan
Transaction processing

- Set isolation level for the current transaction
  - `con.setTransactionIsolationLevel(l);`
  - Where `l` is one of `TRANSACTION_SERIALIZABLE` (default), `TRANSACTION_REPEATABLE_READ`, `TRANSACTION_READ_COMMITTED`, and `TRANSACTION_READ_UNCOMMITTED`

- Set the transaction to be read-only or read/write (default)
  - `con.setReadOnly(true|false);`

- Turn on/off `AUTOCOMMIT` (commits every single statement)
  - `con.setAutoCommit(true|false);`

- Commit/rollback the current transaction (when `AUTOCOMMIT` is off)
  - `con.commit();`
  - `con.rollback();`
Embedded C example

...  /* Declare variables to be “shared” between the application and the DBMS: */
EXEC SQL BEGIN DECLARE SECTION;
  int thisSID; float thisGPA;
EXEC SQL END DECLARE SECTION;
/* Declare a cursor: */
EXEC SQL DECLARE CPS116Student CURSOR FOR
  SELECT SID, GPA FROM Student
  WHERE SID IN
    (SELECT SID FROM Enroll WHERE CID = 'CPS116')
  FOR UPDATE;
...

Embedded C example continued

/* Open the cursor: */
EXEC SQL OPEN CPS116Student;

/* Specify exit condition: */
EXEC SQL WHENEVER NOT FOUND DO break;

/* Loop through result rows: */
while (1) {
    /* Get column values for the current row: */
    EXEC SQL FETCH CPS116Student INTO :thisSID, :thisGPA;
    printf("SID %d: current GPA is %f\n", thisSID, thisGPA);

    /* Update GPA: */
    printf("Enter new GPA: ");
    scanf("%f", &thisGPA);
    EXEC SQL UPDATE Student SET GPA = :thisGPA
        WHERE CURRENT OF CPS116Student;
}

/* Close the cursor: */
EXEC SQL CLOSE CPS116Student;
Pros and cons of embedded SQL

- **Pros**
  - More compile-time checking (syntax, type, schema, …)
  - Code could be more efficient (if the embedded SQL statements do not need to be checked and recompiled at runtime)

- **Cons**
  - DBMS-specific
    - Vendors have different precompilers which translate code into different native API’s
    - Application executable is not portable (although code is)
    - Application cannot talk to different DBMS at the same time
Pros and cons of augmenting SQL

- **Cons**
  - Already too many programming languages
  - SQL is already too big
  - General-purpose programming constructs complicate optimization, and make it difficult to tell if code running inside the DBMS is safe
  - At some point, one must recognize that SQL and the DBMS engine are not for everything!

- **Pros**
  - More sophisticated stored procedures and triggers
  - More application logic can be pushed closer to data