SQL: Programming

CPS 116
Introduction to Database Systems

Announcements (Tue. Sep. 23)

- Homework #2 due next Thursday
  - Please start early—you can do all of it now!
- Homework #1 sample solution available
  - Only in hardcopies; see me if you did not get one in class
- Project milestone #1 due in 3½ weeks

Motivation

- Pros and cons of SQL
  - Very high-level, possible to optimize
  - Not intended for general-purpose computation
- Solutions
  - Augment SQL with constructs from general-purpose programming languages
    - E.g.: SQL/PSM
  - Use SQL together with general-purpose programming languages
    - E.g.: JDBC, embedded SQL
  - Extend general-purpose programming languages with SQL-like constructs
    - E.g.: LINQ (Language Integrated Query for .NET), HQL (Hibernate Query Language)

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
  - Some support more positioning and movement options, modification at the current position, etc.

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 );

SQL/PSM example

CREATE FUNCTION SetMaxGPA(IN newMaxGPA FLOAT)
RETURNS INT
BEGIN
  DECLARE rowsUpdated INT DEFAULT 0;
  DECLARE thisGPA FLOAT;
  DECLARE tudentCursor CURSOR FOR
    SELECT GPA FROM Student
    FOR UPDATE;
  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;
-- Loop over all result rows:
WHILE notMoreRows <> 1 DO
  IF thisGPA > newMaxGPA THEN
    -- Enforce newMaxGPA:
    UPDATE Student SET .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 DB2-specific information, search for “SQL routines” in DB2 v9.5 Information Center

| Link available from course website (under Programming Notes: DB2 SQL Notes)

---

**Interfacing SQL with another language**

- API approach
  - SQL commands are sent to the DBMS at runtime
  - Examples: JDBC, ODBC (C/C++/VB), Python DB API
  - 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 API: JDBC**

```
// JDBC (Java Database Connectivity) is an API that allows a Java program to access databases
// Use the JDBC package:
import java.sql.*;

public class ...
...
    static {
        // Load the JDBC driver:
        try {
            Class.forName("com.ibm.db2.jcc.DB2Driver");
        } catch (ClassNotFoundException e) {
            ... Not very nice since it ties your code to a particular DBMS
            ... Best if you load it from a properties file
            ... Or, for web apps, use a JNDI DataSource (see course website: Programming Notes: Tomcat Notes)
        }
    }
...
```

---

**Connections**

```
// Connection URL is a DBMS-specific string:
String url = "jdbc:db2://cps116.cod.cs.duke.edu:50000/dbcourse";
// Making a connection:
Connection con = DriverManager.getConnection(url, user, password);
...
// Closing a connection:
con.close();
```

- For clarity we are ignoring exception handling here
- Again, in practice you should avoid hard-coding DBMS-specific things (see previous slide)

---

**Statements**

```
// Create an object for sending SQL statements:
Statement stmt = con.createStatement();
// Execute a query and get its results:
ResultSet rs = stmt.executeQuery("SELECT SID, name FROM Student");
// Work on the results:
...
// Execute a modification (returns the number of rows affected):
int rowsUpdated = stmt.executeUpdate("UPDATE Student SET name = 'Barney' WHERE SID = 142");
// Close the statement:
stmt.close();
```
Query results

// Execute a query and get its results:
ResultSet rs =
    stmt.executeQuery("SELECT SID, name FROM Student");
// Loop through all result rows:
while (rs.next()) {
    // Get column values:
    int sid = rs.getInt(1);
    String name = 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, or insert a row into the database
  - Possible only when there is a clear 1-1 correspondence between the change and a row in the underlying table
  - Analogous to the view update problem
    - Covered in the lecture on SQL views
- Obtain metadata: rs.getMetaData() returns a ResultSetMetaData object describing the output table schema (number, order, names, types of columns, etc.)

Prepared statements: motivation

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:
    ...
} // Close the ResultSet:
rs.close();

Prepared statements: syntax

// 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:
    ...
} // Close the PreparedStatement:
stmt.close();

Odds and ends of JDBC

- Most methods can throw SQLException
  - Make sure your code catches them
  - Remember to close Statement, ResultSet, etc., in finally block
  - getSQLException() returns the standard SQL error code
  - getSQLException() returns the error message
  - DataSource interface for establishing connections
  - Methods for examining metadata in databases
  - Methods to retrieve the value of a column for all result rows into an array without calling ResultSet.next() in a loop
  - Methods to construct/execute a batch of SQL statements
  - For additional information and example code, see course website: Programming Notes: JDBC Notes

A note on JDBC drivers

- Type I (bridge): translate JDBC calls to standard API not native to DBMS
  - E.g.: JDBC-ODBC bridge
  - Driver is easy to build using existing standard APIs
  - Extra layer of API adds overhead
- Type II (native API, partly Java): translates JDBC calls to DBMS-specific client API
  - DBMS-specific non-Java client library needs to be installed on each client
  - Good performance
- Type III (network bridge): sends JDBC requests to a middleware server which in turn communicates with a database
  - Client JDBC driver is completely Java, easy to build, and does not need to be DBMS-specific
  - Middleware adds translation overhead
- Type IV (native protocol, full Java): converts JDBC requests directly to native network protocol of the DBMS
  - Client JDBC driver is completely Java but is also DBMS-specific
  - Good performance
  - Supported by, e.g., com.ibm.db2.jcc.DB2Driver
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;  
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;  

/* Open the cursor: */  
EXEC SQL OPEN CPS116Student;  
/* Specify exit condition: */  
EXEC SQL WHENEVER NOT FOUND DO break;  
/* Loop through 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;

Embedded C example continued

/* 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;  
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;  

/* Open the cursor: */  
EXEC SQL OPEN CPS116Student;  
/* Specify exit condition: */  
EXEC SQL WHENEVER NOT FOUND DO break;  
/* Loop through 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 run-time)

- **Cons**
  - DBMS-specific
    - Vendors have different precompilers which translate code into different native APIs
    - Application executable is not portable (although code is)
    - Application cannot talk to different DBMS at the same time

Pros and cons of augmenting SQL

- **Pros**
  - More sophisticated processing inside DBMS
  - More application logic can be pushed closer to data

- **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!

Making a language SQL-like?

- E.g.: LINQ (for C#), HQL (for Java/Hibernate)
- Example LINQ code (from Wiki)

  ```csharp
  int someValue = 5;  
  var results = from c in someCollection  
    let x = someValue * 2  
    where c.SameProperty < x  
    select new {c.SameProperty, c.OtherProperty};  
  foreach (var result in results) {  
    Console.WriteLine(result);  
  }
  ```

- Automatic data mapping and query translation
- Fad or way of the future?