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 (SQL/PSM)
  - Use SQL together with general-purpose programming languages (JDBC, embedded SQL, etc.)

JDBC

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

  ```java
  import java.sql.*;
  public class … {
    static {
      // Load the JDBC driver:
      Class.forName("COM.ibm.db2.jdbc.net.DB2Driver");
    }
    …
  }
  ...
  ...
  ...
  ```

Connections

- Connection URL is a DBMS-specific string:
  ```java
  String url = "jdbc:db2://rack40.cs.duke.edu/cps116";
  // Making a connection:
  Connection con = DriverManager.getConnection(url);
  ...
  // Closing a connection:
  con.close();
  ...
  ```

Statements

- Create an object for sending SQL statements:
  ```java
  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:
  ```java
  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
  - 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, even though all strings are essentially the same query (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();`
- 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();`

Odds and ends of JDBC

- Most methods can throw `SQLException`
  - Make sure your code catches them
- `getSQLState()` returns the standard SQL error code
- `getMessage()` returns the error message
- 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 and execute a batch of SQL statements together
  - `...`

JDBC drivers – Types I, II

- Type I (bridge): translate JDBC calls to a standard API not native to the DBMS (e.g., JDBC-ODBC bridge)
  - Driver is easy to build using existing standard API’s
  - Extra layer of API adds overhead
- Type II (native API, partly Java): translates JDBC calls to DBMS-specific client API calls
  - DBMS-specific client library needs to be installed on each client
  - Good performance
JDBC drivers – Types III, IV

♦ 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

Other database programming methods

♦ API approach
  • SQL commands are sent to the DBMS at runtime
  • Examples: JDBC, ODBC (for C/C++/VB), Perl DBI
  • These APIs 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 convert them into DBMS-specific API calls
  • Examples: embedded SQL for C/C++, SQLJ (for Java)

Embedded C example

```c
/* 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 CPS196Student CURSOR FOR
SELECT SID, GPA FROM Student
WHERE SID IN (SELECT SID FROM Enroll WHERE CID = 'CPS196')
FOR UPDATE;
/* Open the cursor: */
EXEC SQL OPEN CPS196Student;
/* 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 CPS196Student INTO :thisSID, :thisGPA;
  printf("SID %d: current GPA is %f\n", thisSID, thisGPA);
  printf("Enter new GPA: ");
  scanf("%f", &thisGPA);
  EXEC SQL UPDATE Student SET GPA = :thisGPA
WHERE CURRENT OF CPS196Student;
}
/* Close the cursor: */
EXEC SQL CLOSE CPS196Student;
```

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;
/* Declare a cursor: */
EXEC SQL DECLARE CPS196Student CURSOR FOR
SELECT SID, GPA FROM Student
WHERE SID IN (SELECT SID FROM Enroll WHERE CID = 'CPS196')
FOR UPDATE;
/* Open the cursor: */
EXEC SQL OPEN CPS196Student;
/* 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 CPS196Student INTO :thisSID, :thisGPA;
  printf("SID %d: current GPA is %f\n", thisSID, thisGPA);
  printf("Enter new GPA: ");
  scanf("%f", &thisGPA);
  EXEC SQL UPDATE Student SET GPA = :thisGPA
WHERE CURRENT OF CPS196Student;
}
/* Close the cursor: */
EXEC SQL CLOSE CPS196Student;
```

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 APIs
    • Application executable is not portable (although code is)
    • Application cannot talk to different DBMS at the same time

SQL/PSM stored procedures/functions

♦ 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
-- 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 GPA > 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