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 operate 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
-- 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

```sql
-- 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 PostgreSQL-specific information, look for “PL/pgSQL” in PostgreSQL documentation
  - Link available from course website (under Programming Notes: PostgreSQL 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

```java
import java.sql.*;

public class ...
{
    static {
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            // ClassNotFound
        }
    }
    ...
```

- 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

```java
String url = "jdbc:postgresql://localhost/ubuntu";

Properties props = new Properties();
props.setProperty("user", "ubuntu");
props.setProperty("password", "reverse");
Connection con = DriverManager.getConnection(url, user, password);
...
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

```java
ResultSet rs = stmt.executeQuery("SELECT SID, name FROM Student");
...
```

- Create an object for sending SQL statements:
- Execute a query and get its results:
- Work on the results:
- Execute a modification (returns the number of rows affected):
- Close the statement:
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
- 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 = conn.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 result:...
}

- 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
```

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
  - getSQLState() returns the standard SQL error code
  - getMessage() 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., org.postgresql.Driver
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 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 result rows: */
while (1) {
  /* Get column values for the current row: */
  EXEC SQL FETCH CPS116Student INTO :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 (vs. API)

- **Pros**

- **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

- **Cons**
  - Already too many programming languages
  - SQL is already too big

- **Pros**

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.SomeProperty < x
  select new { c.SomeProperty, c.OtherProperty};
foreach (var result in results) {
  Console.WriteLine(result);
}
```

- Automatic data mapping and query translation
- But a different syntax for each host language?