Embedded SQL
But why?

Database Management System (DBMS)

Users/Apps

Data processing

Control Format

Integrate with application etc

Data!
SQL + a conventional language

- If we can combine SQL with code in a conventional language, we can solve these problems.
- But we have another problem:
  - SQL is based on relations, and conventional languages have no such type.
- It is solved by
  - feeding tuples from SQL to the other language one at a time, and
  - feeding each attribute value into a particular variable.
Approaches

• Three approaches for combining SQL and a general-purpose language:
  • Stored Procedures
  • Statement-level Interface
  • Call-level interface
Three Approaches
I. Stored Procedures

• The SQL standard includes a language for defining “stored procedures”, which can
  • have parameters and a return value,
  • use local variables, ifs, loops, etc.,
  • execute SQL queries.

• Stored procedures can be used in these ways:
  • called from the interpreter,
  • called from SQL queries,
  • called from another stored procedure,
  • be the action that a trigger performs.
Example (just to give you an idea)

Reference: textbook chapter 9

CREATE FUNCTION BandW(y INT, s CHAR(15)) RETURNS BOOLEAN
IF NOT EXISTS
    (SELECT *
    FROM Movies
    WHERE year = y AND studioName = s)
THEN RETURN TRUE;
ELSIF 1 <=
    (SELECT COUNT(*)
    FROM Movies
    WHERE year = y AND studioName = s AND
    genre = 'comedy')
THEN RETURN TRUE;
ELSE RETURN FALSE;
END IF;
Not very standard

• The language is called **SQL/PSM** (Persistent Stored Modules).
  • It came into the SQL standard in SQL3, 1999.
  • Reference: textbook, section 9.4

• By then, various commercial DBMSs had already defined their own proprietary languages for stored procedures
  • They have generally stuck to them.

• **PostgreSQL** has defined **PL/pgSQL**.
  • It supports some, but not all, of SQL/PSM.
  • Reference: Chapter 39 of the PostgreSQL documentation.
2. Statement-level interface (SLI)

- Embed SQL statements into code in a conventional language like C or Java.
- Use a preprocessor to replace the SQL with calls written in the host language to functions defined in an SQL library.
- Special syntax indicates which bits of code the preprocessor needs to convert.
Example (just to give you an idea)

Reference: textbook example 9.7

void printNetWorth() {
    EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50];
    int presNetWorth;
    char SQLSTATE[6]; /* Status of most recent SQL stmt
    EXEC SQL END DECLARE SECTION;

    /* OMITTED: Get value for studioName from the user. */
    EXEC SQL SELECT netWorth INTO :presNetWorth
    FROM Studio, MovieExec
    WHERE Studio.name = :studioName;

    /* OMITTED: Report back to the user */
Big picture (figure 9.5)
3. Call-level interface (CLI)

- Instead of using a pre-processor to replace embedded SQL with calls to library functions, write those calls yourself.
- Eliminates need to preprocess.
- Each language has its own set of library functions for this.
  - for C, it’s called SQL/CLI
  - for Java, it’s called JDBC
  - for PHP, it’s called PEAR DB
- We’ll look at just one: JDBC.
JDBC
JDBC Architecture

- **Java Application**
- **JDBC Driver Manager**
- **JDBC Driver (PostgreSQL)**
- **JDBC Driver (Oracle)**
- **JDBC Driver (Sybase)**

- PostgreSQL Database
- Oracle Database
- Sybase Database
JDBC - Steps

Steps:
1- Load the driver and register it with the driver manager (provided you’ve already downloaded the driver “jar” file)
2- Connect to a database
3- Create an SQL statement
4- Execute a query and retrieve the results, or, make changes to the database
5- Disconnect from the database

Steps:
1- Load the driver and register it with the driver manager (provided you’ve already downloaded the driver “jar” file)
2- Connect to a database
3- Create an SQL statement
4- Execute a query and retrieve the results, or, make changes to the database
5- Disconnect from the database
Using JDBC on cdf

• You need to run your JDBC code on dbsrv1.
• The PostgreSQL driver for JDBC is on cdf here:
  /local/packages/jdbc-postgresql
You’ll also find an example program and a how-to in that directory.
• To run JDBC code, you need this driver in your classpath.
• Example: Suppose you have a class called Jelly.java.
  javac Jelly.java
  java -cp ~/bin/postgresql-8.3-607.jdbc4.jar: Jelly
JDBC Example (see section 9.6)

Do this once in your program:

/* Get ready to execute queries. */
import java.sql.*;

/* A static method of the Class class. It loads the specified driver */
Class.forName("org.postgresql.jdbc.Driver");

Connection conn = DriverManager.getConnection(
    "jdbc:postgresql://localhost:5432/csc343h-userID,
    userID,"");

/* Continued ... */
The arguments to getConnection

```java
Connection conn = DriverManager.getConnection(
    jdbc:postgresql://localhost:5432/csc343h-userID,
    userID,""
);
```

- **jdbc:postgresql**
  We’ll use this, but it could be, e.g., **jdbc:mysql**

- **localhost:5432**
  You must use exactly this for cdf.

- **csc343h-userID and userID**
  Substitute your cdf userid.

- **""**
  Password (unrelated to your cdf password).
  Literally use the empty string.
Do this once per query in your program:

/* Execute a query and iterate through the resulting tuples. */

`PreparedStatement` `execStat = conn.prepareStatement(
    "SELECT netWorth FROM MovieExec");`

`ResultSet` `worths = execStat.executeQuery();`

while (worths.next()) {
    int worth = worths.getInt(1);

    /* If the tuple also had a float and another int attribute, you’d get them by calling 
    worths.getFloat(2) and worths.getInt(3). Or you can look up values by attribute name. 
    Example: `worths.getInt(netWorth)` */

    /* OMITTED: Process this net worth */
Exceptions can occur

• Any of these calls can generate an exception.
• Therefore, they should be inside try/catch blocks.

```java
try {
    /* OMITTED: JDBC code */
} catch (SQLException ex) {
    /* OMITTED: Handle the exception */
}
```

• The class `SQLException` has methods to return the `SQLSTATE`, etc.
What is “preparation”?

- Preparing a statement includes parsing the SQL, compiling and optimizing it.
- The resulting `PreparedStatement` can be executed any number of times without having to repeat these steps.
If the query isn’t known until run time

• You may need input and computation to determine the query.
• You can hard-code in the parts you know, and use “?” as a placeholder for the values you don’t know.
• This is enough to allow a `PreparedStatement` to be constructed.
• Once you know values for the placeholders, methods `setString`, `setInt`, etc. let you fill in those values.
Example (figure 9.22)

```java
PreparedStatement studioStat =
    conn.prepareStatement(
        "INSERT INTO Studio(name, address) VALUES(?, ?)"
    );

/* OMITTED: Get values for studioName and studioAddr */
studioStat.setString(1, studioName);
studioStat.setString(2, studioAddr);
studioStat.executeUpdate();
```
Why not just build the query in a string?

• We constructed an incomplete `preparedStatement` and filled in the missing values using method calls.
• Instead, we could just build up the query in an ordinary string at run time, and ask to execute that.
• There are classes and methods that will do this in JDBC.
• But never use that approach because it is vulnerable to injections: insertion of strings into a query with malicious intent.
• Always use a `preparedStatement` instead.
Example with `createStatement`

Statement `stat = conn.createStatement();`

String `query =
    "SELECT networth
     FROM MovieExec
     WHERE execName like '%%Spielberg%';"

`ResultSet worths = stat.executeQuery(query);`
Example: Some vulnerable code

Suppose we want the user to provide the string to compare to
You can do this rather than hard-coding Spielberg into the query:

```java
Statement stat = conn.createStatement();
String who = /* get a string from the user */
String query =
    "SELECT networth
     FROM MovieExec
     WHERE execName like '%%' + who + '%%';"
ResultSet worths = stat.executeQuery(query);
```
A gentle user does no harm

If a user enters Cameron, the SQL code we execute is this:

```
SELECT networth
FROM MovieExec
WHERE execName like ‘%Cameron%’;
```

Nothing bad happens.
An injection can exploit the vulnerability

What could a malicious user enter?

```sql
SELECT networth
FROM MovieExec
WHERE execName like '??????????????????%';
```
An injection can exploit the vulnerability

But if a malicious user enters

```
Milch%'; drop table Contracts; --
```

the code we execute is this:

```
SELECT networth
FROM MovieExec
WHERE execName like '%Milch%'; DROP TABLE Contracts; --%';
```

In other words:
```
SELECT networth
FROM MovieExec
WHERE execName like '%Milch%';
```
```
DROP TABLE Contracts; --%';
```

“Did you really name your son Robert’); DROP TABLE Students; -- ??”
Queries vs updates in JDBC

• The previous examples used `executeQuery`.
• This method is only for pure queries.
• For SQL statements that change the database (insert, delete or modify tuples, or change the schema), use the analogous method `executeUpdate`.