Lecture 13:
• Views, Stored Procedures
• Dynamic SQL Intro

Announcements:
• HW 4 due (due by end of week)
• HW 5 out (moved to next week)
Views and Stored Procedures

A **view** in SQL is a “virtual” table ...

- provide a way to name a query (as if it were a table)
- but the query result is not stored ... not **materialized**

Consider the schema: `pepper(id, common_name, heat, heat_unit)`

- The following view creates a “named query” (hot_pepper)

  ```sql
  CREATE VIEW hot_pepper AS
  SELECT id, common_name, heat_unit
  FROM pepper
  WHERE heat='hot' OR heat='very hot';
  ```

- The view can then be queried: `SELECT * FROM hot_pepper`

The view table changes with the underlying tables

<table>
<thead>
<tr>
<th>id</th>
<th>common_name</th>
<th>heat</th>
<th>heat_unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bell</td>
<td>mild</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>jalapeno</td>
<td>medium</td>
<td>4750</td>
</tr>
<tr>
<td>3</td>
<td>anaheim</td>
<td>mild</td>
<td>250</td>
</tr>
<tr>
<td>4</td>
<td>habanero</td>
<td>hot</td>
<td>225000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(select the pepper table)

```
SELECT id, common_name FROM hot_pepper
```

<table>
<thead>
<tr>
<th>id</th>
<th>common_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>habanero</td>
</tr>
<tr>
<td>7</td>
<td>ghost</td>
</tr>
</tbody>
</table>

```
INSERT INTO pepper VALUES (8, 'carolina reaper, 'very hot', 1800000);
```

```
SELECT id, common_name FROM hot_pepper
```

<table>
<thead>
<tr>
<th>id</th>
<th>common_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>jalapeno</td>
</tr>
<tr>
<td>4</td>
<td>habanero</td>
</tr>
<tr>
<td>8</td>
<td>carolina reaper</td>
</tr>
</tbody>
</table>
Can also store query results as tables in MySQL ...

```sql
CREATE TABLE hot_pepper_table AS
    SELECT id, common_name, unit
    FROM   pepper
    WHERE  heat = 'hot' OR heat = 'very hot';
```

The new table is just a plain-old SQL table ...

- if `pepper` table is updated, no changes made to `hot_pepper_table`
- unlike with a view (which would also be “updated”)

Some systems support **materialized views**, which are updated as tables change

- *Note*: some (virtual and materialized) views can also be updated
- updates made to underlying tables “through” the view

**Some Advantages of Views**: help provide additional layers of abstraction

- can be created that specifically support users/applications
- can hide complexity from users (e.g., for complex queries)
- allow table schemas to change while keeping view schemas consistent
- help restrict access for security (limit access to views)
**Prepared statements**

- name and store a parameterized query on the server

```sql
PREPARE get_pepper_name FROM 'SELECT common_name FROM pepper WHERE id=2';

SET @my_var = 2;

EXECUTE get_pepper_name USING @my_var; -- or just USING 3

+-------------+
| common_name |
+-------------+
| jalapeno    |
```

- prepared statements are “compiled” (parsed, analyzed, optimized)

**Stored procedures are similar, but with more features ...**

- allow more programming-language-like expressions

- simple example:

```sql
CREATE PROCEDURE get_heat_unit(IN p_id INT UNSIGNED, OUT p_unit INT)
BEGIN
    SELECT heat_unit INTO p_unit
    FROM pepper
    WHERE id = p_id;
END

CALL get_heat_unit(3, @heat_unit);
SELECT @heat_unit AS 'heat_unit';
```

```
+-----------+
| heat_unit |
+-----------+
| 4750      |
```

**Some Advantages:**

- efficiency in that pre-compiled

- security and permissions (abstracts above base relations)

- still have to watch out for SQL injection! (but both can help ... more later)
Using SQL within programs

Most SQL is generated from software applications!

- via libraries or frameworks
- SQL statements reference "host variables" ... program vars passed as args

SQL query results can be large

- don’t want programs to “hold” large query results
- so DBMSs support “result sets”
- similar to I/O streams and “iterators” ... e.g., hasNext(), next()

Different approaches for using SQL within a program:

1. “Dynamic” SQL ... what we’ll discuss
   - SQL statements as strings, passed to API calls

2. Object-Relational Mappings (ORMs) ... (Hibernate, web frameworks)
   - automatic mapping to/from tables and class objects

3. “Embedded” SQL ... not as common today
   - preprocessor for SQL “embedded” in the programming language
   - modern example is Microsoft’s LINQ (Language Integrated Query)
Our plan ...

- Briefly look at basic dynamic SQL APIs in three languages
- Start with Java and JDBC
- Then Python and C++
- Then basic web-based approaches

We’ll only cover the basics ...

- enough to give you a feel for how the API’s work
- which can be helpful for building simple apps
- and also even when using frameworks

Java Database Connectivity (JDBC) API

- A Java API for accessing RDBMSs (RDBMS independent)
- Each specific DBMS implements a JDBC “driver” (i.e., the API)
- Similar to MS ODBC
- Many languages today have DBMS-specific libraries
- ... but most follow JDBC/ODBC style

Must install the specific JDBC driver for your system

- Connector/J for MySQL (mariadb.com/kb/en/about-mariadb-connector-j/)
- This is just a jar file ...
- The JDBC API comes standard in Java (but not the Driver)
- Other bindings: mariadb.com/kb/en/connectors/
import java.sql.*;

public class MySQLQuery {
    
    public static void main(String[] args) throws Exception {
        // create a connection to the DBMS
        String url = "jdbc:mariadb://cps-database.gonzaga.edu/bowersDB";
        Connection cn = DriverManager.getConnection(url, "user", "password");

        // create a (non-prepared) statement
        Statement st = cn.createStatement();
        String query = "SELECT * FROM pet ORDER BY name";

        // execute the query and print result set
        ResultSet rs = st.executeQuery(query);
        while(rs.next()) {
            String name = rs.getString("name");
            System.out.println("name = " + name);
        }

        // clean up connections
        rs.close();
        st.close();
        cn.close();
    }
}

Note: examples take some shortcuts for exposition ...  
- exception handling (try-catch blocks), error checking, etc.

To compile and run the program (with installation of connector):

$ javac MySQLQuery.java
$ java MySQLQuery
name = babe
...

To compile and run the program (just downloading the connector):

javac MySQLQuery.java
java -cp .:lib/mariadb-java-client-3.2.0.jar MySQLTest1
name = babe
...