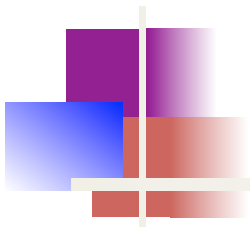


Database Application Programming

Chapter 6

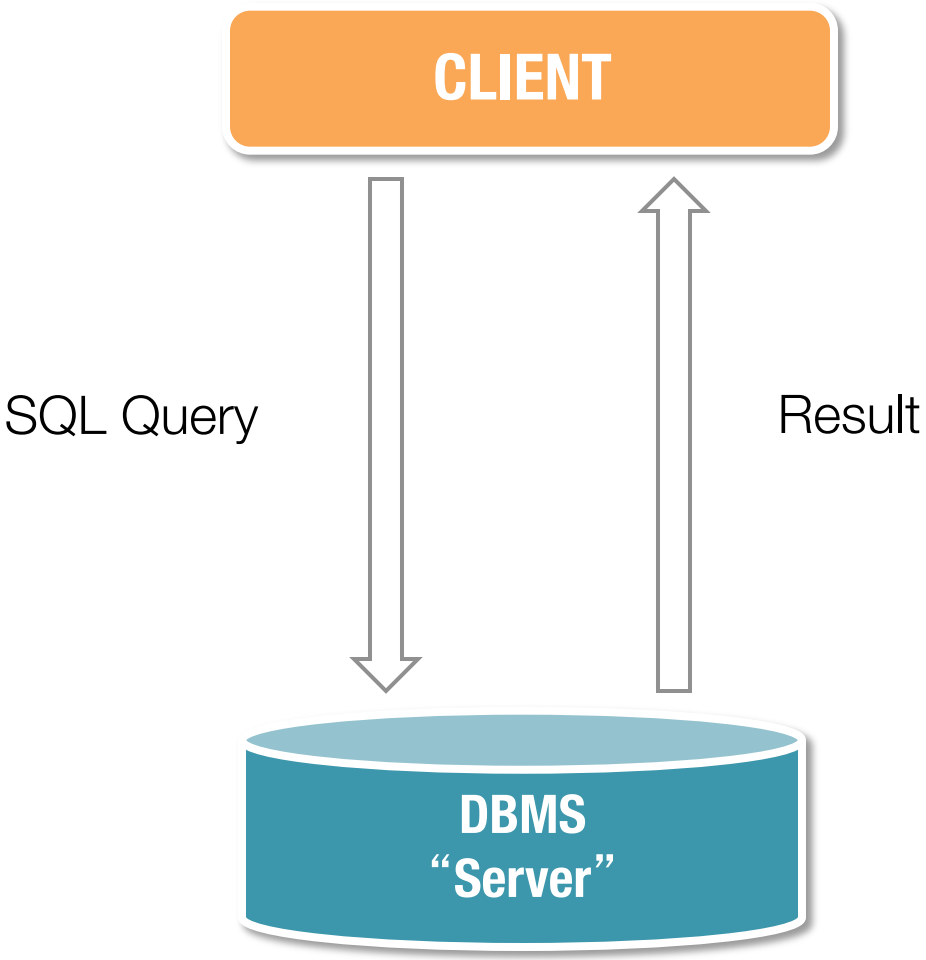
(JDBC Section)

Databases "In the Wild"

- So far, we've talked about the DBMS as a standalone system
  - Access interactively by writing SQL queries (e.g., using SQL*Plus)
- In practice, DBMS is often part of a larger software infrastructure
  - Multi-tiered system architecture
  - Access database from another program

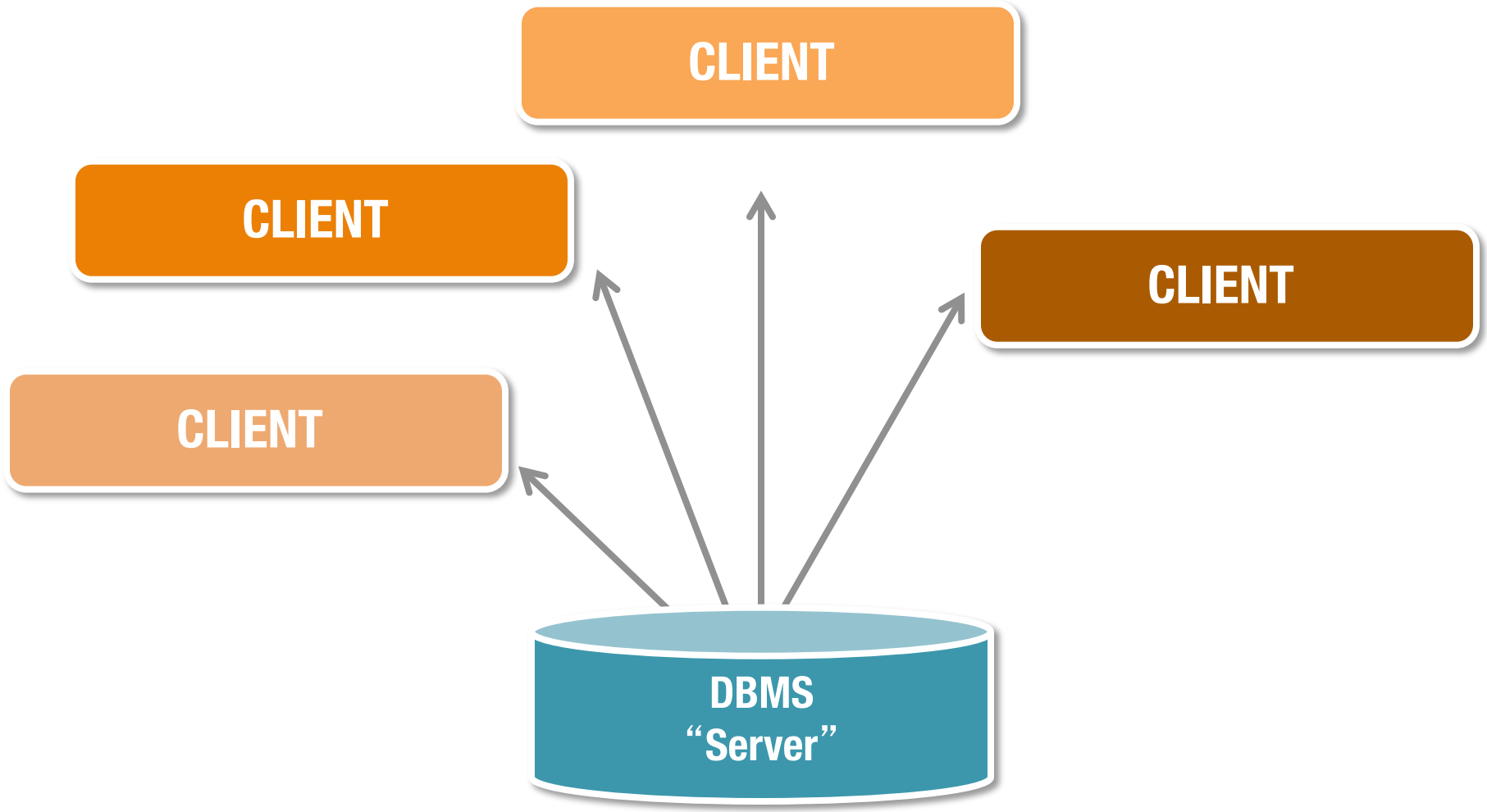
Database “Ecosystem” (1)

“Client-Server Architecture”



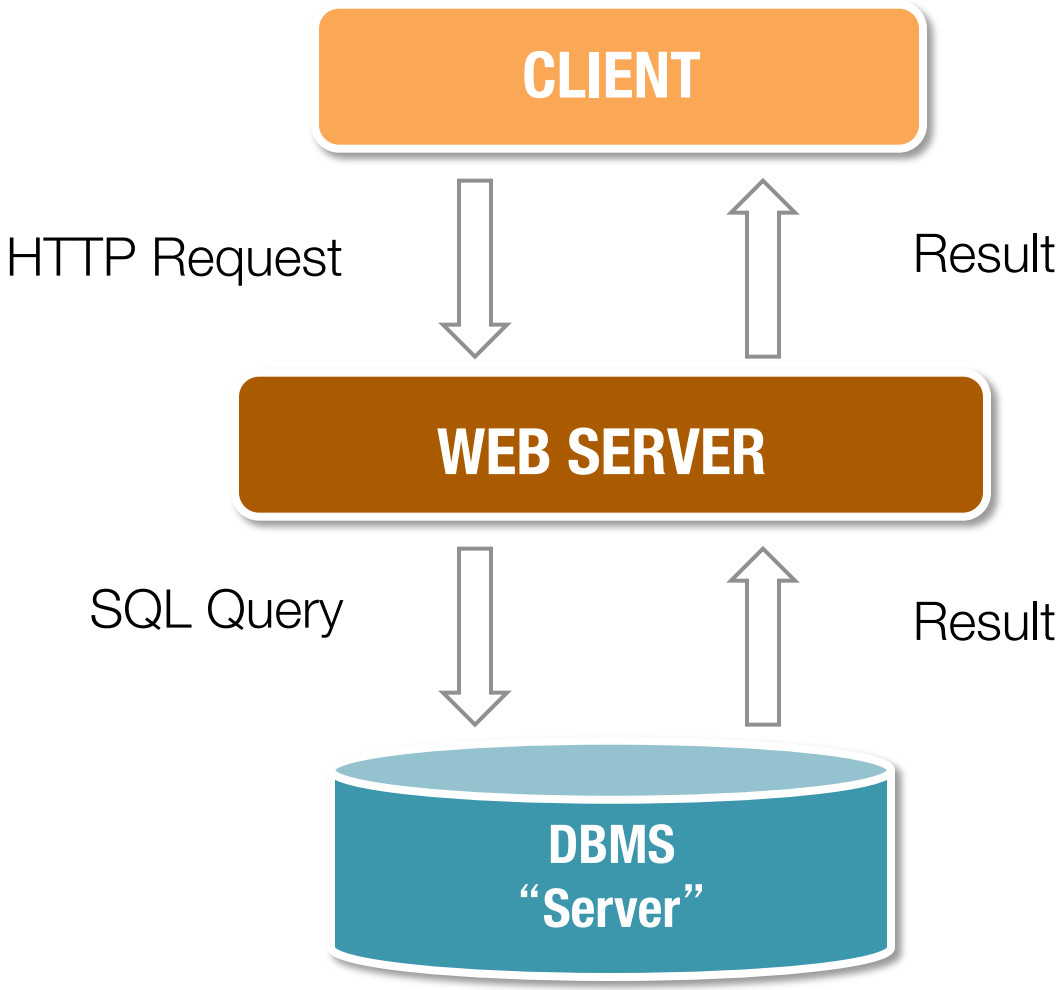
Many Clients

"Client-Server Architecture"

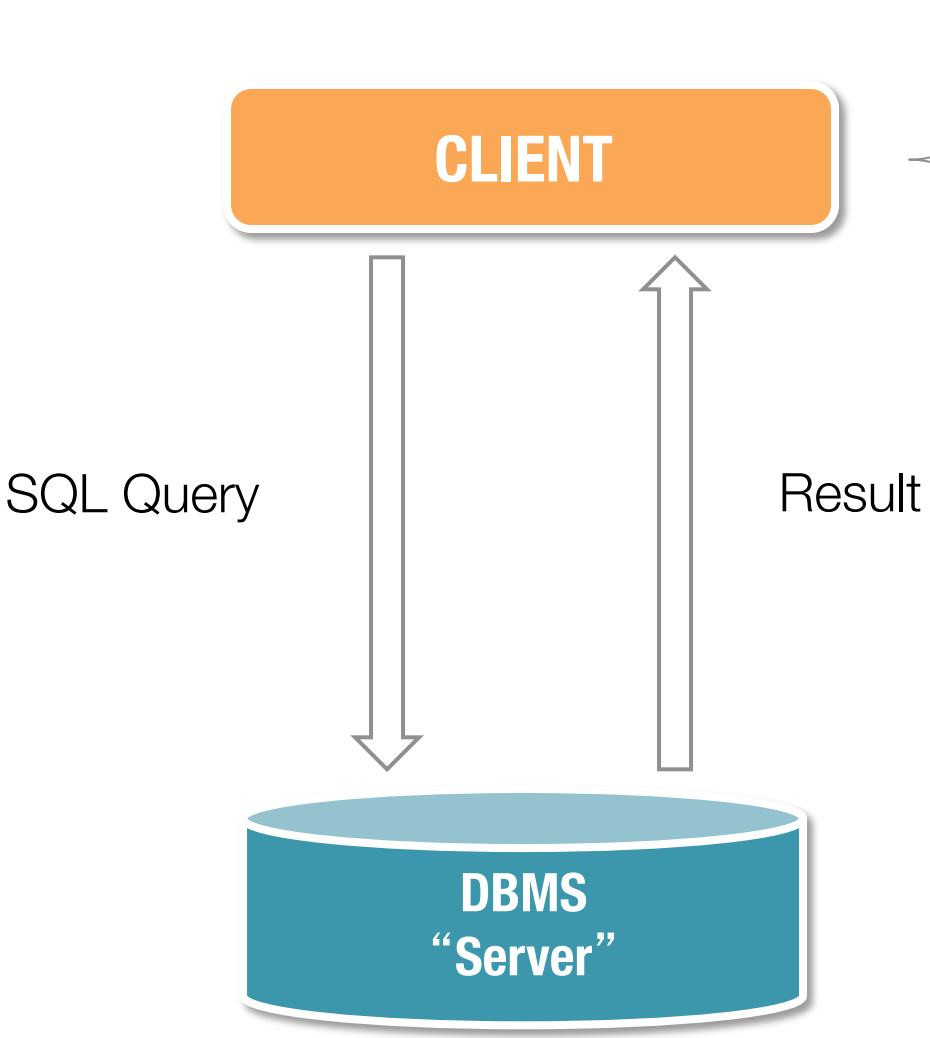


Database "Ecosystem" (2)

"3-Tier Architecture" (Common to add more tiers, too)



Embedding SQL in Application



Client is often a program, written in a language like C++ or Java

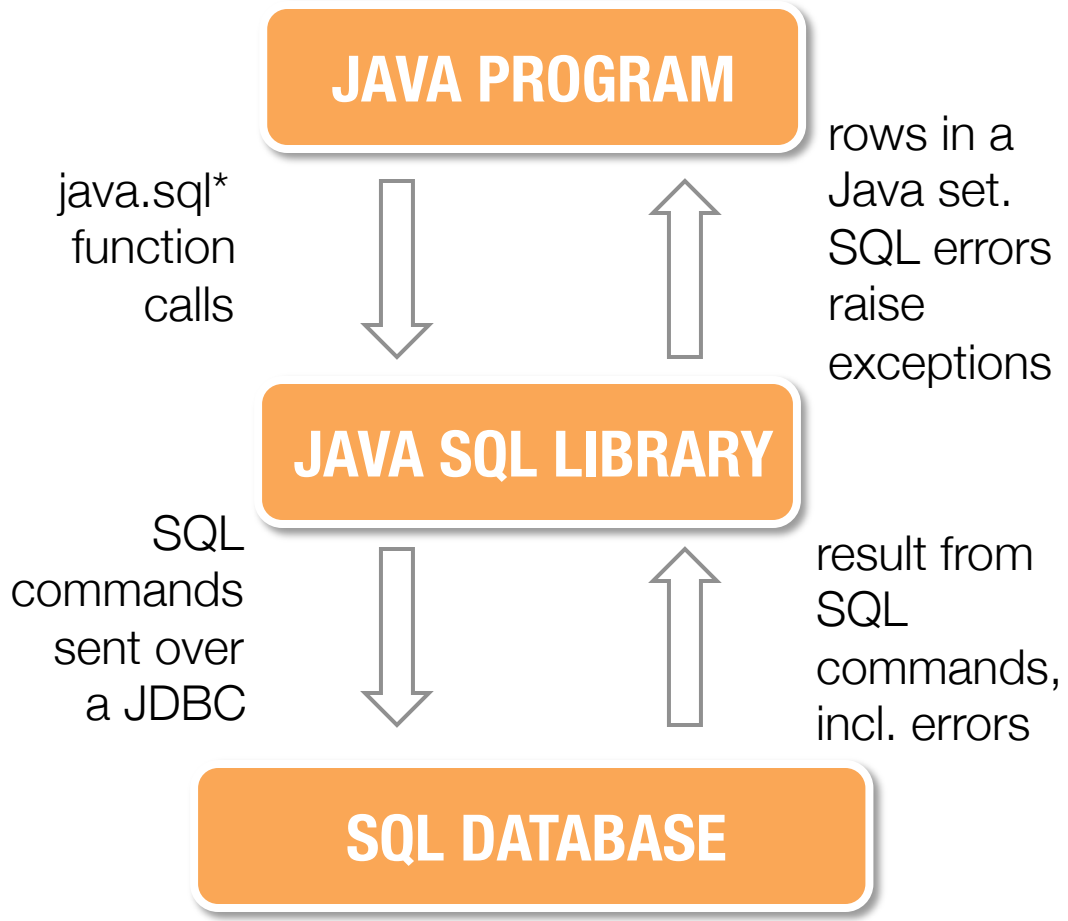
CHALLENGE:
How to access SQL from application code?

SQL Integration with PL

- Ugly Problem:
  - Database supports SQL queries
  - Application written in programming language (e.g. Java, C++)
  - What is the interface between the two?
- A Common Solution:
  - “Embed” SQL in host language
  - Provide an API for processing query results
- Also object-relational mapping tools
  - LINQ to SQL, Ruby on Rails…

JDBC ("Java-Database Connectivity")

- Connect to a database using a JDBC driver
- Send queries over the JDBC connection
- Receive results into a Java ResultSet



Try things out with sqlite / Oracle

- Download JDBCTest.java and the latest jdbc driver for sqlite from https://bitbucket.org/xerial/sqlite-jdbc/downloads
- Alternative: Oracle driver posted with Project 2.
- Compile: javac JDBCTest.java
- Run: java –cp .:*sqlite-jdbc-3.7.2.jar* JDBCTest

Opening/Closing Connections

- Get Connection object:
  - Oracle requires passwords
  - Sqlite3 is file-based and does not
    - Connection conn = ….
- Always close connections before quitting the program
  - conn.close();
- A cool trick in recent Java/JDBC to auto-close:

  http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html

JDBC Example

```
Connection conn;
// Obtain a connection to DB, store in conn
// (Requires JDBC driver; See sample code in Project 2)

String q = "SELECT name FROM Students WHERE GPA > 3.5";
try {
      Statement st = conn.createStatement();
      ResultSet rs = st.executeQuery(q);

      while (rs.next()) {
         String name = rs.getString(”name");
         System.out.println(name);
      }
      rs.close();
      st.close();
}
catch(SQLException e){System.err.println(e.getMessage());}
```

Cursor retrieves rows from result one at a time

Full Javadoc for java.sql available online:
http://download.oracle.com/javase/6/docs/api/

Challenges

- DBMS and PL implement different types
  - “Impedance Mismatch”
- Need to match DB types with PL types

SQL Type	Java Type	ResultSet Method
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
INTEGER	Integer	getInt()
NUMBER	(depends)	(depends)

Challenges

- What computation to do in the database vs. the application program? 
- Rules of Thumb:
  - Avoid fetching more data than necessary
  - “Push” data processing to the DBMS when possible

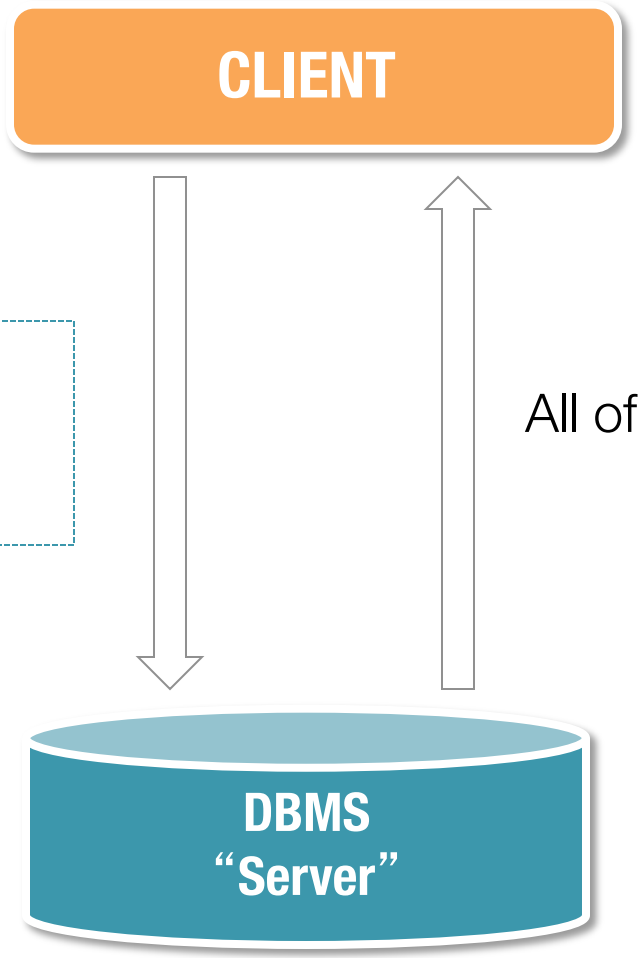
JDBC Example

What does the following code snippet do?

```
String query = "SELECT NAME, AGE FROM CITIZENS";
try {
      double sum = 0;
      double count = 0;
     Statement st = conn.createStatement();
     ResultSet rs = st.executeQuery(query);
     while (rs.next()) {
        String name = rs.getString("NAME");
        sum += rs.getDouble("AGE");
         count++;
     }
      System.out.println(sum/count);
     // be sure to close rs and st as well here
     // and in exception handlers. Omitted for brevity.
}
catch(SQLException e){System.err.println(e.getMessage());}
```

What Happens?

Compute the average age




```
SELECT NAME, AGE
FROM CITIZENS
```

All of the citizen records

What if there are millions of citizens?

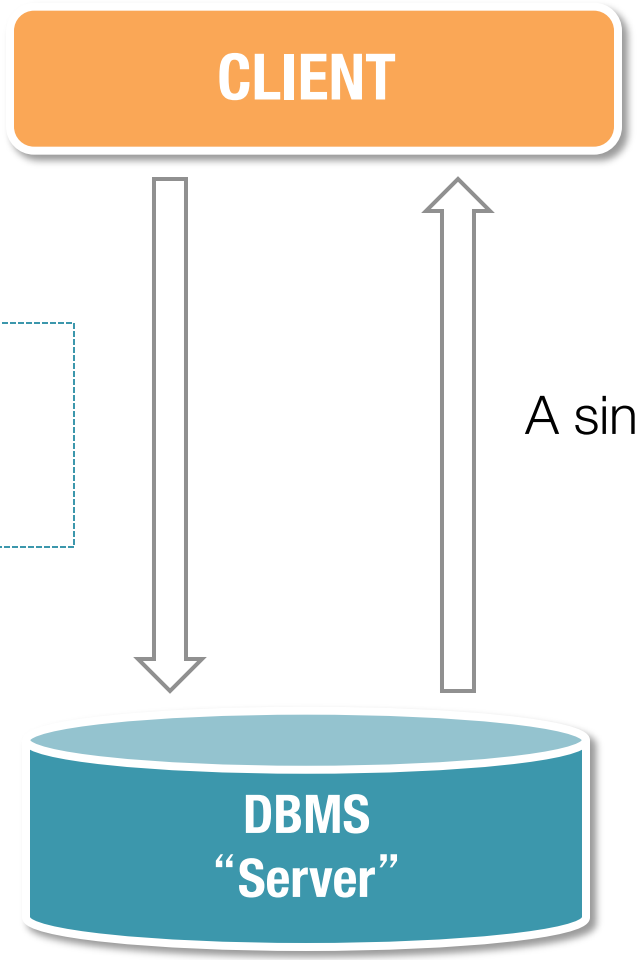
JDBC Example (Revised)

Push the computation “closer” to the data…
Make DBMS do processing it does well.

```
String query = "SELECT AVG(AGE) FROM CITIZENS";
try {
     Statement st = conn.createStatement();
     ResultSet rs = st.executeQuery(query);
     while (rs.next()) {
        Double avg = rs.getDouble(1);
         System.out.println(avg);
     }
     // close omitted for brevity
}
catch(SQLException e){System.err.println(e.getMessage());}
```

What Happens Now?

Compute the average age




```
SELECT AVG(AGE)
FROM CITIZENS
```

A single number

Statement vs. PreparedStatement

- Statement (Base class)
  - Arbitrary SQL query
  - DBMS parses and optimizes each query
- PreparedStatement
  - Parameterized SQL query
  - DBMS “pre-compiles” the query (parses, optimizes, and stores query execution plan)
  - Can be used multiple times
  - Amortizes optimization cost across multiple uses

Statement vs. PreparedStatement

```
public List<String> getNames (int age) {
   String q = "SELECT name FROM Students WHERE age = " + age;
   Statement st = conn.createStatement();
   ResultSet rs = st.executeQuery(q);
   …
}
```

```
public List<String> getNames (int age) {
   String q = "SELECT name FROM Students WHERE age = ?";
   PreparedStatement ps = conn.prepareStatement(q);
   ps.setDouble(1, age);
   ResultSet rs = ps.executeQuery();
   …
}
```

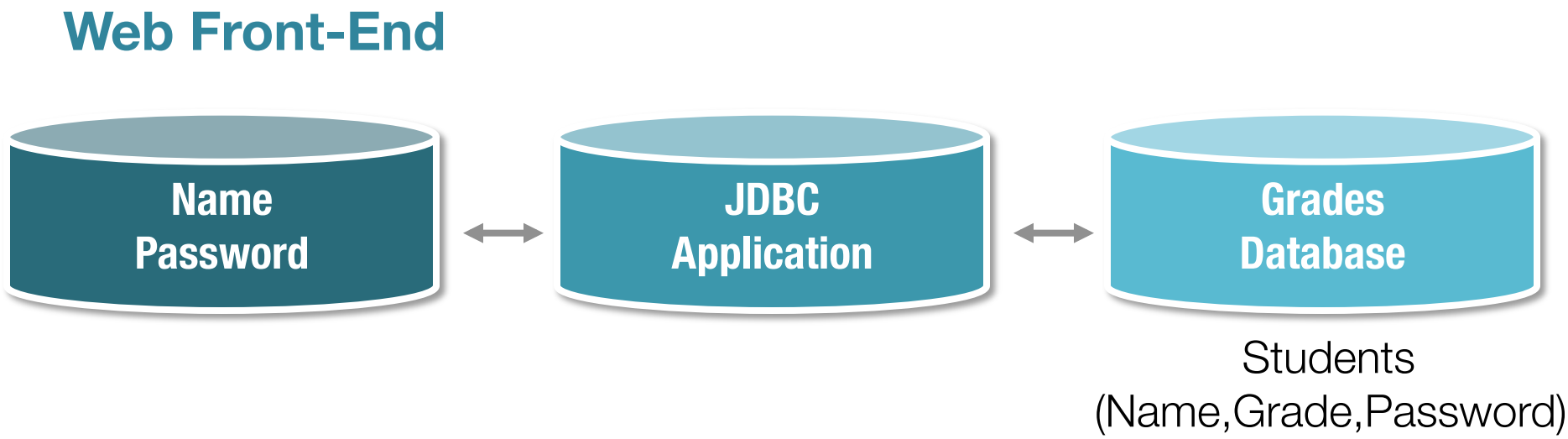
Which is more efficient if getNames() called many times?

PreparedStatement

- PreparedStatement also does some type checking on parameters and removal of escape characters
- Helpful for preventing some security problems (e.g. SQL Injection)

Security Issues / SQL Injection

- Common vulnerability in database applications
  - SQL Injection is simple, yet surprisingly common
  - Can be prevented with defensive coding



SQL Injection – Example

JDBC

```
Authenticate (String n, String p) {
…
String query =
"SELECT grade FROM students WHERE name = " + n + "’
AND password = ‘" + p + "’ ";
…
}
```

Input:

name = bart; password = mypword

SQL:

```
SELECT grade FROM students WHERE name = ‘bart’ AND
password = ‘mypword’
```

SQL Injection – Example

JDBC

```
Authenticate (String n, String p) {
…
String query =
"SELECT grade FROM students WHERE name =
‘" + n + " ’  AND password = ‘" + p + "’ ";
…
}
```

Input:

name = lisa; password = n’ OR ‘x’=‘x

SQL:

```
SELECT grade FROM students WHERE name = ‘lisa’ AND
password = ‘n’ OR ‘x’ = ‘x’
```

SQL Injection – Example

JDBC

```
Authenticate (String n, String p) {
…
String query =
"SELECT grade FROM students WHERE name =
‘" + n + " ’  AND password = ‘" + p + "’ ";
…
}
```

Input:

```
name= foo; password= n’; UPDATE students SET grade= ‘A
```

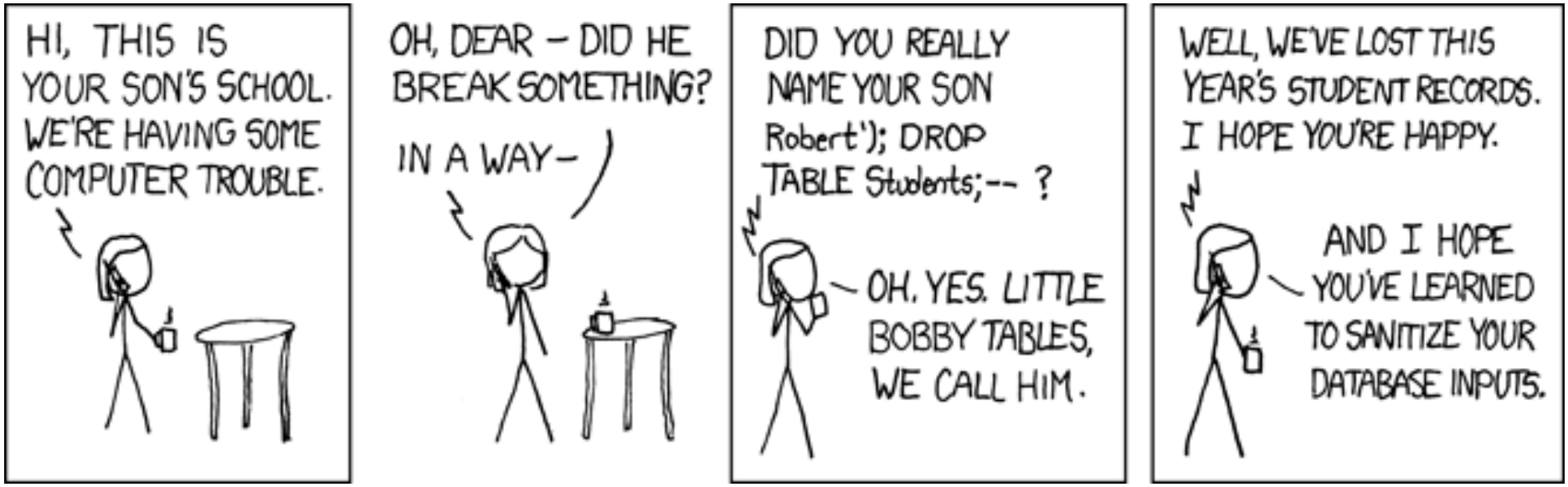
SQL:

```
SELECT grade FROM students WHERE name = ‘foo’ AND
password = ‘n’; UPDATE students SET grade = ‘A’
```

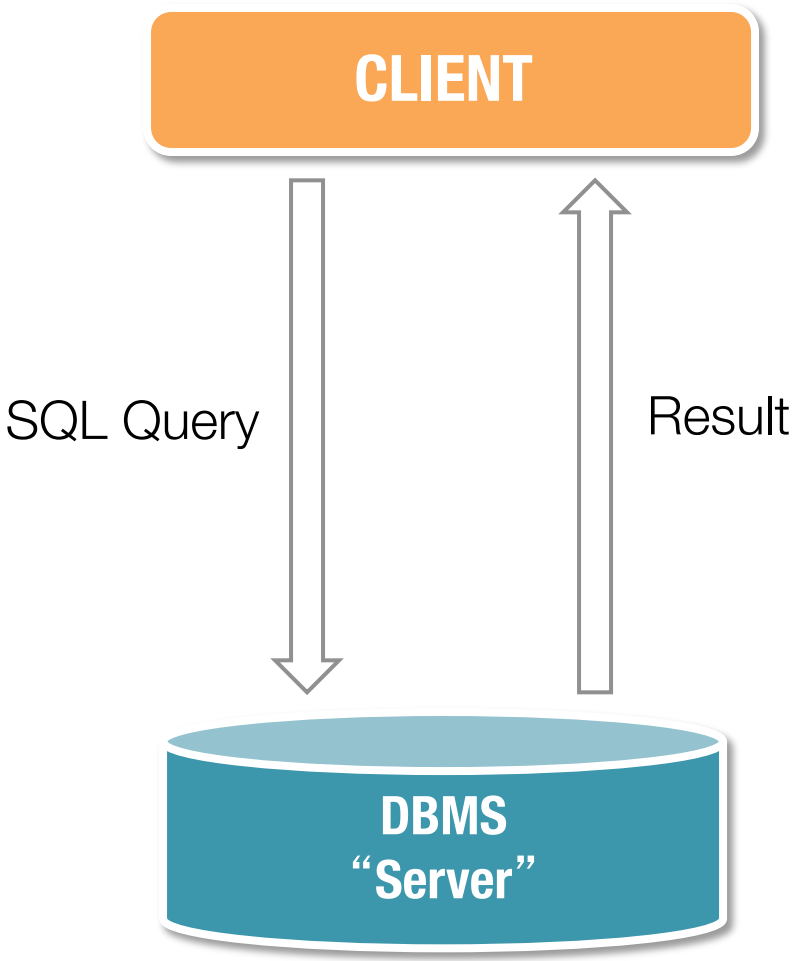
SQL Injection

What is a simple way to prevent this kind of attack?

Validate input!
Use a prepared statement for user input parameters!



Stored Procedures and UDFs



Database applications you write (e.g., using JDBC) are usually located outside the DBMS.

Exceptions:

- Stored Procedures
- User-Defined Functions (UDFs)
- Stored and executed within the DBMS

Summary

- DBMS is often part of a larger software infrastructure
  - E.g., Database-backed web applications
- Integrating SQL with application code is a messy problem
  - Efficiency: Push computation “close” to data
  - Security: Validate input
  - Testing/Debugging