Database Programming: Motivation & “Host Language” Approaches

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today’s Lecture: Overview

Introduction

Context

- Since beginning this course, focus has been on SQL: the standard for accessing relational databases
  - You've used PSQL (a terminal-based front-end) to interact with POSTGRESQL
  - You've used file-based input to drive commands to POSTGRESQL
- This lecture (and the next) explore an important new topic: application programming that accesses a relational database
- Key point: you write a program in your favorite programming language: part of the code retrieves data from, stores data in, an RDB
  - As usual, we'll focus on Java ☺
- In this paradigm, SQL is still used to interact with the database
  - But the SQL is embedded (or used as the "data sub-language") inside the host language program

Some Disclaimers

- "Database programming" is a <u>very</u> broad topic
- Entire courses and books are devoted to the topic ...
  - Because at the "nitty-gritty" level, there are a plethora of details to master
- Plan: introduce you to the broad contours of what's out there
  - When you need to do this for a living, you can teach yourself the rest
- Another issue: unlike "core SQL", support by specific DBMS implementation for the "SQL standard" ranges from sketch to non-existent ☹
- My examples are biased to POSTGRESQL since that's our database of choice
- Do <u>not</u> expect that you can type the textbook's examples into your program and it will "just work"
  - This is not the usual "syntax details" issue: I'm talking about non-existent support for important features

Demystification

- IMHO, much of what's written about database programming obscures an important fact
- Although the technologies that provide support for database programming may seem to differ significantly …
  - …they share fundamental characteristics
- Because they must all solve the same issues
  - And (mostly because "it works") they use the same solutions
- Today's lecture: identify these "set of issues", and emphasize how similar the different solutions are

Some Issues

What's Wrong With SQL? (I)

- Q: SQL is such a powerful language, why not continue to use the "pure SQL" approach?
- *A_1*: even at its best, SQL can only be used to interact with a relational database
- There are many other software "entities" out there that SQL simply cannot talk to
- Implication: we must devise a way to bridge SQL code to non-SQL code
- Examples:
  - Terminal interaction with end-user
  - GUI interaction with end-user
  - External systems such as email, printers, networks

No man is an island entire of itself; every man is a piece of the continent, a part of the main;

John Donne, "Devotions upon Emergent Occasions"

What's Wrong With SQL? (II)

- A_2: equally importantly, just because a language is "technically" Turing complete doesn't imply that computation in that language is as easy as in other Turing complete languages
- SQL is a declarative, set-based language (at its core)
- Once you're not directly manipulating relations, you find yourself needing
  - Conditional statements
  - Loop constructs
  - Methods & functions
- Being able to "drop back" into your favorite programming language makes life so much easier ☺
- As we shall see, integrating the SQL approach with procedural or object-oriented approaches can be difficult
  - But: far more difficult to not integrate them
  - Google Relational Blocks: "a visual, domain-specific, dataflow language" if you don't believe me ☺

Scenario

1. You go to an ATM to withdraw some money
2. You swipe your card
3. “Something” (a program, not a relational database) reads it
4. You punch in your PIN, a program reads it
5. The program talks to a relational database to see if the PIN matches a database record
6. Assuming a match
    6.1 You ask for account balance, program reads what you punched, constructs a query to a relational database
    6.2 Program understands the database’s response
    6.3 Formats and displays response on the screen

    6.1 You want to withdraw money
    6.2 Program constructs a database request to update your account tuple
    6.3 Program understands the database’s response
    6.4 Formats and displays response on the screen

Another Scenario: Rendering Web-Pages

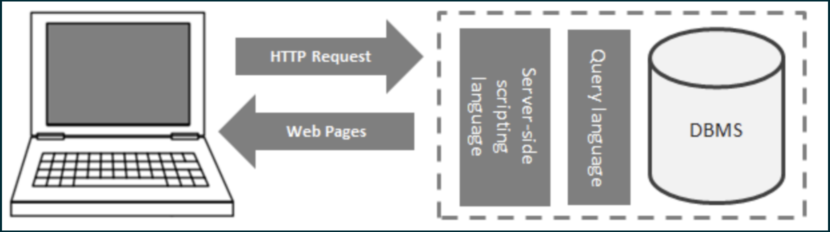
- Fundamentally, this scenario is identical to the previous one!
- I'm introducing it to show you
  1. How pervasive "database programming" is
  2. That "pure SQL" is simply not an option
- First: "client-side" scripting ...
- Client-side scripting is about enabling web-pages to render (often dynamically) without asking the server for "stuff"
  - No need for the server to tell the client how to render
  - No need for the server to give the client data to render
- Key idea: your laptop ("client device") is a pretty powerful computer
- Store & execute the code (and data) used to render a web-page on the client-device
  - Advantage: performance (both client and server ☺)

Limitations Of Client-Side Scripting?

- Q: can you spot major limitations with client-side scripting?
  - Especially in the context of our course?
- A: where is the data that's rendered in these web-pages supposed to come from?
  - Any interesting web-page is going to require (server-side) data!
- Note: we can use a "load once, then cache on client" strategy ... but cache invalidation is a difficult problem
  - Even more difficult: how to propagate client-side changes to the data to the "master" version on the server?
- Architecturally simpler to use server-side scripting!
  - Server gives the client a web-page, and says "render this!"

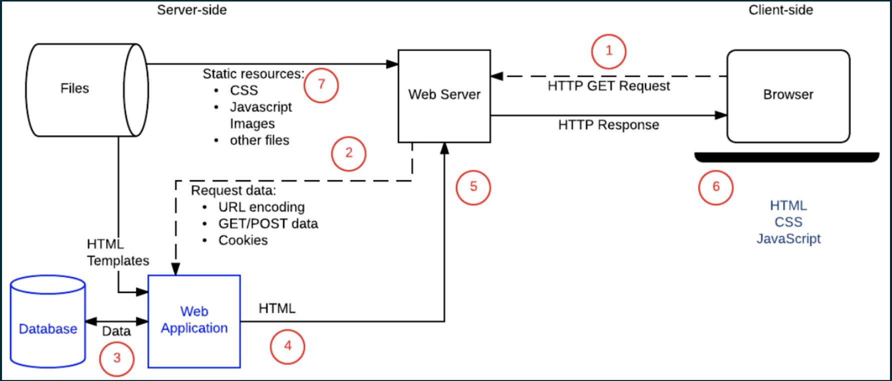
Note: in practice, a "server-side ⇒ client-side" continuum exists, I'm simplifying here for pedagogic contrast ☺

Server-Side Scripting (I)



- When a user loads a web-page in her browser, the browser asks the server to run a script that will generate the requested web-page
- Key points
  - The server is generating HTML & CSS which are meaningless on the server itself
  - The server must interleave server-side data into the web-page that it's dynamically generating for the client

Server-Side Scripting (II)

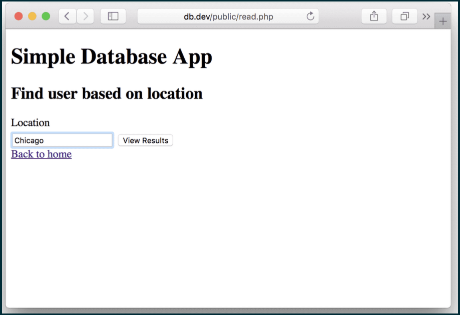


- This flow diagram makes the programming of an implementation look simple …
- In fact, it’s a big mess ☹, mostly because we’re forced to mix different data-models and programming languages in the same program!

Critique: PHP

- Google ``php language is an abomination'' and marvel at the results
- Then Google ``popularity of php 2020'' ☺
- Key point: however "ugly" PHP code, the alternatives don't fundamentally improve the situation ☹
  - Because they're all essentially offering the same solution to the same problem
- Will quickly walk through some vanilla PHP code to illustrate this problem
  - Note: I've taken this code from here

PHP Example: The Views



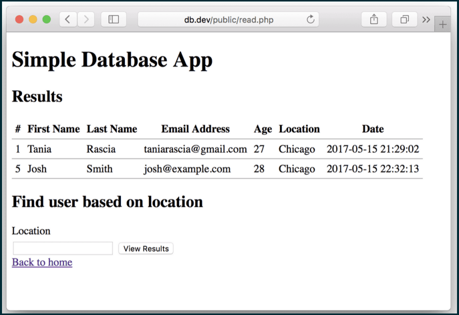
Simple Database App

Find user based on location

Location

Chicago [View Results]

Back to home



Simple Database App

Results

#	First Name	Last Name	Email Address	Age	Location	Date
1	Tania	Rascia	taniarascia@gmail.com	27	Chicago	2017-05-15 21:29:02
5	Josh	Smith	josh@example.com	28	Chicago	2017-05-15 22:32:13

Find user based on location

Location

[View Results]

Back to home

Database

- Already stores user profile (including *location*)
- These web-forms allow users to issue a "query-by-location" to the database, and receive a web-page populated with the result
- Really can't get much simpler than that ☺

PHP Example: The Code (I)

```php
if (isset($_POST['submit'])) {
  try {
    require "../config.php";
    require "../common.php";

    $connection = new PDO($dsn, $username, $password, $options);

    $sql = "SELECT *
    FROM users
    WHERE location = :location";

    $location = $_POST['location'];

    $statement = $connection->prepare($sql);
    $statement->bindParam(':location', $location, PDO::PARAM_STR);
    $statement->execute();

    $result = $statement->fetchAll();
  } catch(PDOException $error) {
    echo $sql . "<br>" . $error->getMessage();
  }
}
?>
```

PHP Example: The Code (II)

```
<?php
if (isset($_POST['submit'])) {
  if ($result && $statement->rowCount() > 0) { ?>
    <h2>Results</h2>

    <table>
      <thead>
<tr>
  <th>#</th>
  <th>First Name</th>
  <th>Last Name</th>
  <th>Email Address</th>
  <th>Age</th>
  <th>Location</th>
  <th>Date</th>
</tr>
      </thead>
      <tbody>
  <?php foreach ($result as $row) { ?>
      <tr>
<td><?php echo escape($row["id"]); ?></td>
<td><?php echo escape($row["firstname"]); ?></td>
<td><?php echo escape($row["lastname"]); ?></td>
<td><?php echo escape($row["email"]); ?></td>
<td><?php echo escape($row["age"]); ?></td>
<td><?php echo escape($row["location"]); ?></td>
<td><?php echo escape($row["date"]); ?> </td>
      </tr>
    <?php } ?>
      </tbody>
  </table>
  <?php } else { ?>
    > No results found for <?php echo escape($_POST['location']); ?>.
  <?php }
} ?>
```

PHP Example: The Code (III)

```
<h2>Find user based on location</h2>

<form method="post">
  <label for="location">Location</label>
  <input type="text" id="location" name="location">
  <input type="submit" name="submit" value="View Results">
</form>

<a href="index.php">Back to home</a>

<?php require "templates/footer.php"; ?>
```

“Take-Away” Conclusions (I)

- This example (and the previous ATM scenario) shows us that a very large set of application programs must inevitably:
  - Include code that accesses relational databases (using SQL)
  - Include code that converts client input from a non-relational domain into an “SQL-suitable” format
  - Include code that converts relational database output into a “non-relational” format
- Unfortunately: host programming language does not understand the concepts of relations, tuples, relational operators
- The problem isn’t only that host language primitives only correspond roughly to SQL attributes
  - That problem can be solved by a good OO language such as Java ☺

“Take-Away” Conclusions (II)

- The problem is that:
  - Host-language computation is (usually) not set-based (alternatively, not declarative): the relations need to be “unpacked” to be useful
  - SQL only understands relations so it can’t do anything useful outside of a database computation

Note: these are not “client-side” *versus* “server-side” issues!

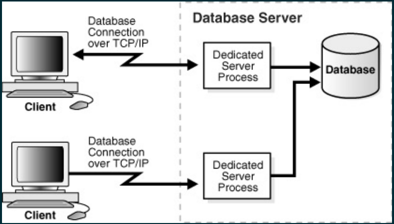
These are intrinsic “software (data) modeling” issues ☹

Impedance Mismatch

- By analogy to the notion of an impedance mismatch in the field of electronics, we can refer to a programming impedance mismatch
- The phrase captures the idea that "too large" a gap between different programming/data models makes our development lives very painful ☹
- Violent disputes break out about the existence (or non-existence) of an impedance mismatch between OO programming and the relational database model
  - We may have time to explore this later
- We've already pointed out the impedance mismatch between typical host programming languages (e.g, "Java" and "C++") and the relational database model
- Not everyone agrees, but I claim that this impedance mismatch is not removed simply by enhancing SQL with constructs such as conditional logic, loops, and procedures!

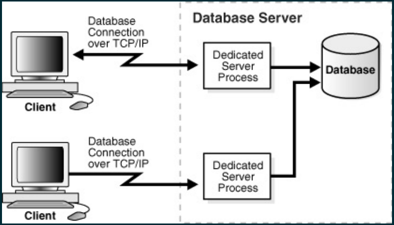
Other Fundamental Issues: Database Is Self-Contained Software (I)

- The whole point of a DBMS is to have encapsulated software that provides function to many clients
  - We refer to such software as "server-side" software ☺
- Implication: DBMS runs in its own process for robustness and security
- Implication: any client that wants to access a database must follow:
  - Some transport protocol supported by the database
  - Some security protocol supported by the database
- These protocols typically differ from the equivalent OS protocols!



The details will differ, but the basics are constant

Other Fundamental Issues: Database Is Self-Contained Software (II)



Meta-Implications:

- Given this software "structure" ...
- Any client-server database interaction will contain a database "connection" construct
- The connection encapsulates a client⇔server
  - Authentication mechanism
  - Naming mechanism (at database, schema, and table granularities)
  - Transport mechanism

An Aside: Embedded Databases Are Different

- You should be aware of the embedded database concept which take a completely different approach from what I've been describing
- With an embedded database, the database code resides in the same "code-base" as the service or application that uses it
    - Meaning: the "client" and "server" execute in the same process
    - Meaning: no network penalty ☺
- Implication: the authentication and transport protocols are much "thinner" or non-existent
    - Although Java embedded databases present clients with the same "connection-based" APIs that are used by their networked brethren
    - Q: why do you think they took this approach?
- Before we leave this topic: can you think of another embedded database used in this course?

“Inside” Or “Outside” The Database (I)

- I’ve shown that practical database programming must include a combination of both “pure SQL” and “host language” code
- Once we accept this, there basically two approaches
  - Grow the capabilities of SQL such that it can express more business logic than “pure SQL”
  - Grow the capabilities of the host programming language so it can express more relational database logic
- Key point: an end-to-end, non-trivial, application will still require a combination of both approaches
  - But there are significant differences depending on how large a role each approach plays in the overall application
- It will never(?) be possible to send an email with SQL
- It will never(?) be possible to perform a join that respects referential integrity in a host programming language
- But note: the impedance mismatch issue is orthogonal to the “inside” or “outside” the database dichotomy!

“Inside” Or “Outside” The Database (II)

- With respect to “authentication” and “transport” issues, however, there is a significant difference between “inside” and “outside” the database approaches
  - Only the “outside the database” approach has to deal with these issues
  - The “inside the database” approach doesn’t: such programs are already executing inside the database!
    - No need for the program to authenticate itself to the database
    - No need to do “inter-process” communication
- But: these issues still remain, at a different programming granularity!
  - Even when a significant chunk of programming logic is packaged as an “inside the database” function …
    - The client must still “invoke” that function using authentication over a given transport protocol

“Outside The Database”

Segue

- We're now going to drill down on the "outside the database" approach to database programming
  - Meaning: *"how do we extend a vanilla programming language to enable interactions with a relational database?"*
- Two approaches:
  - Embedded SQL approach: statically embed database commands in an explicit "SQL section"
  - Library approach: calls to a host-language "relational database" API

Dynamic *Versus* Static Dichotomy

- The "embedded SQL" *versus* "library" distinction corresponds to a static *versus* dynamic dichotomy
- With embedded SQL:
  - Database statements are identified by a special prefix
  - A pre-processor scans the source code, identifies the "database statements", and extracts them for processing by the DBMS
  - The DBMS replaces the original source code with code that can interact with the DBMS
  - This replacement is static: you can't change the SQL code without recompiling the source code
- With the "library" approach:
  - At runtime, host language strings are converted to SQL statements
  - This is dynamic: the content of these "strings" can change based on interaction with end-users or based on some internal computation

Fundamentally Same Approaches

- Sure, from a syntax perspective, the "embedded SQL" and "library" approaches are very different
  - Examples coming up soon ☺
- But: from a conceptual and runtime flow perspective, these approaches are basically identical!
- At runtime:
  1. Client program opens a connection to the database
  2. The client's program, written in some host language, uses the connection to submit DBMS-specific requests
  3. The DBMS processes these commands in the same way it processes commands submitted from a terminal "front end"
  4. Result-sets (if any) are made available to client, one tuple at a time (more on this process later)
  5. The client closes the connection
- This similarity is not a coincidence ☺

Historical Context

- Initially, clients interacted with relational databases using an SQL command-line processor
  - Just as you've been doing ...
- Very soon after (mid-1970s - early 1980s), business needs (first half of lecture) drove the creation of the embedded SQL approach
- Only much later (1990s and later) did ODBC and JDBC (examples of the "library" approach) emerge after many standardization attempts
- Key point: you may not actually use embedded SQL, but the ODBC & JDBC APIs are a (relatively) "shallow wrapper" around embedded SQL
- Plan: we're therefore going to begin by drilling down on the "embedded SQL" concepts
  - Java version: SQLJ
- Finish today's lecture with a quick discussion of ODBC, and longer discussion of JDBC in the next lecture

Example (C): Quick Comparison

```
/* select values from table into host variables using STATIC
 * SQL and print them
 */
EXEC SQL SELECT id, name, dept, salary
  INTO :id, :name, :dept, :salary
  FROM staff WHERE id = 310;
```

```
/* Update column in table using DYNAMIC SQL */
strcpy(hostVarStmtDyn,
       "UPDATE staff SET salary = salary + 1000  WHERE dept
           = ?");
EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;
EXEC SQL EXECUTE StmtDyn USING :dept;
```

Embedded SQL: Syntax Concerns

- EXEC SQL statement is used to identify embedded SQL request to the preprocessor

```
        exec sql <embedded SQL statement>;
```

- The statement identification syntax varies by language
  - Example: In COBOL, the semicolon is replaced with END-EXEC
  - SQLJ (the Java version of embedded SQL) uses #sql syntax

```
String name;
int id=37115;
float salary=20000;
#sql {select ename into :name
  from emp where empno=:id and sal>:salary};
System.out.println("Name is: " + name);
```

Embedded SQL: Declaring Variables

- Variables of the host language can be used within embedded SQL statements
- They are preceded by a colon (:) to distinguish from SQL attribute names and variables
  - Example: *:credit versus credit*
- Variables must be declared within a DECLARE section
  - Declaration syntax follows host-language rules

```
        exec sql begin declare section;
        int  credit;
        exec sql end declare section;
```

Connecting To the Database

Before executing any SQL statements, program must first connect to the database

```
    exec sql connect to server user user-name using password
```

- Q: Is this an SQL-related function?
- A: Not really
  - Nothing to do with SQL, relations, tuples
  - This is a "must somehow bridge two different systems" function

May Have to Deal with NULL Issue

- Sometimes the value inserted or retrieved will be NULL
- The host language may not know how the database is encoding NULL values
- Must use special "indicator variables" to indicate that the value is actually NULL

```
EXEC SQL SELECT profit
INTO :Plantprofit INDICATOR :Ind
WHERE C = 75;
```

- If host language variable Ind is negative, then program knows that that Plantprofit does not contain a "real" value
  - DBMS actually returned NULL

How Does DBMS Report Error Conditions To Host System? (I)

- When you interact with DBMS from a Terminal, easy for DBMS to report error condition
  - Print an error message ☺
- Q: How do you report an error condition when interaction done with code (and no humans)?
  - Especially with inter-process communication
- (Quick question: as a Java practitioner, what would you suggest?)

How Does DBMS Report Error Conditions To Host System? (II)

- A: As part of the declaration section, declare a SQLCODE variable
- DBMS will magically set the state of this variable to indicate whether the operation was successful
  - If unsuccessful, variable value indicates specific type of problem
- (Quick answer: Java wraps the SQLCODE in an *SQLException*)

```
exec sql update staff set job = 'clerk' where job = 'mgr';
if ( sqlcode < 0 )
   printf( "update error:  sqlcode = %d. \n", sqlcode);
```

Dealing With Relations

- To handle a relation in a host language, we need a "looping" mechanism that will allow us to iterate through the relation
  - One tuple at a time
- This mechanism is called a CURSOR
- DECLARE a CURSOR, in a way similar to defining a query
  - This defines – but does not compute – the result set relation
- OPEN a CURSOR
  - The relation is now computed, but is not accessible
- FETCH CURSOR is executed in order to get a tuple
  - Repeat the FETCH invocation until all tuples are processed
  - Current tuple is referred to as CURRENT
  - Use SQLCODE to check whether any unprocessed tuples
- CLOSE the CURSOR "destroys" the result set

Cursor Usage: Example

Increase the profit of all plants in Miami by 10%, if the profit is less than 0.1

```
plantcity:='miami';
exec sql declare cursor todo as
select *
from plant
where city = :plantcity;

exec sql open cursor todo;

while sqlcode = 0 do
begin
   exec sql fetch todo
      into :plant, :plantname, :plantcity, :plantprofit;
   if :plantprofit < 0.1 then
      exec sql update plant
      set profit = profit*1.1
      where current of todo
end;

exec sql close cursor todo;
```

Example from Professor Zvi Kedem

Using Cursors To Modify Database

- Cursors are not restricted to Database → Program
- Can also go the other direction: Program → Database
  - Modify the database through an UPDATE, INSERT, DELETE statement

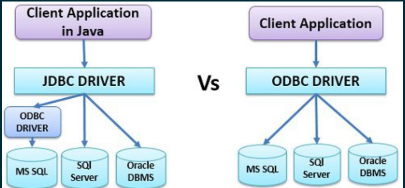
```
exec sql
declare c cursor for
select *
from instructor
where dept_name = 'music'
-- Can update tuples fetched by cursor
-- by declaring that the cursor is 'for update'
for update
```

- Now that the cursor has been declared as FOR UPDATE
- Can iterate through the tuples (via FETCH operation)
- After each "fetch", update the tuples as in example below

```
update instructor
set salary = salary + 1000
where current of c
```

“Library” Approach: ODBC & JDBC

- ODBC (*Open Database Connectivity*): (Microsoft, 1992)
- JDBC (*Java Database Connectivity*): (Sun, 1997)
- Very similar (conceptually): both provide non-embedded, dynamic SQL access
  - The “library” is usually referred to as a “driver”
  - Program: interleaves your business logic with calls to driver’s SQL functions
  - ODBC provides procedural APIs, JDBC provides object-oriented APIs (see Figure below)
- Think of JDBC as “ODBC for Java”
- Most bindings for other languages (JS, Swift) have copied this approach
- We won’t be discussing ODBC, will discuss JDBC next lecture

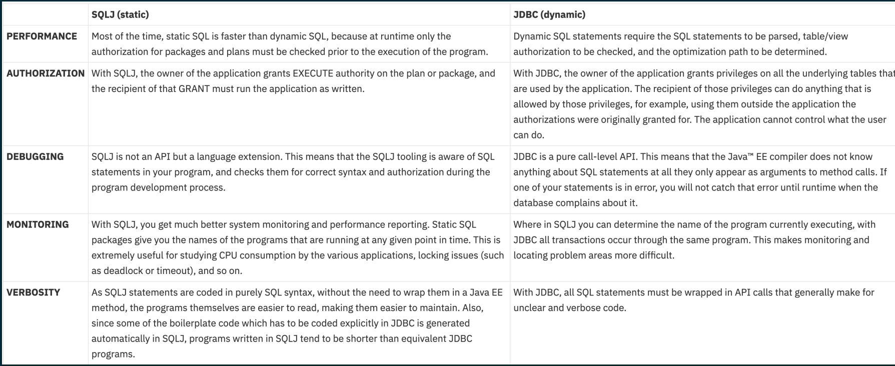


Comparing The Two Approaches (I)

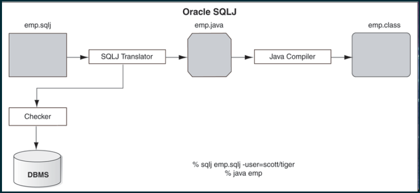
- Disadvantages relative to dynamic SQL
  - Embedded SQL requires a compilation phase and binding to specific database
    - So: not portable
  - Parameter values must be known at compilation time
    - So: cannot depend on user input
- Advantages:
  - Pay the performance penalty (parameter binding, creating an access plan for the database) up-front at compilation time
    - Faster at run-time
  - Error-checking (parsing, statement validation, and optimization) done (and only once) before you run the program
    - With dynamic SQL you lose much of the type-safety that you've become used to with Java ☹
    - Embedded SQL compilation can even validate that the attribute domain values are compatible with host language variable!

Comparing The Two Approaches (II)

(Table taken from here)



	SQLJ (static)	JDBC (dynamic)
PERFORMANCE	Most of the time, static SQL is faster than dynamic SQL, because at runtime only the authorization for packages and plans must be checked prior to the execution of the program.	Dynamic SQL statements require the SQL statements to be parsed, table/view authorization to be checked, and the optimization path to be determined.
AUTHORIZATION	With SQLJ, the owner of the application grants EXECUTE authority on the plan or package, and the recipient of that GRANT must run the application as written.	With JDBC, the owner of the application grants privileges on all the underlying tables that are used by the application. The recipient of those privileges can do anything that is allowed by those privileges, for example, using them outside the application the authorizations were originally granted for. The application cannot control what the user can do.
DEBUGGING	SQLJ is not an API but a language extension. This means that the SQLJ tooling is aware of SQL statements in your program, and checks them for correct syntax and authorization during the program development process.	JDBC is a pure call-level API. This means that the Java™ EE compiler does not know anything about SQL statements at all they only appear as arguments to method calls. If one of your statements is in error, you will not catch that error until runtime when the database complains about it.
MONITORING	With SQLJ, you get much better system monitoring and performance reporting. Static SQL packages give you the names of the programs that are running at any given point in time. This is extremely useful for studying CPU consumption by the various applications, locking issues (such as deadlock or timeout), and so on.	Where in SQLJ you can determine the name of the program currently executing, with JDBC all transactions occur through the same program. This makes monitoring and locating problem areas more difficult.
VERBOSITY	As SQLJ statements are coded in purely SQL syntax, without the need to wrap them in a Java EE method, the programs themselves are easier to read, making them easier to maintain. Also, since some of the boilerplate code which has to be coded explicitly in JDBC is generated automatically in SQLJ, programs written in SQLJ tend to be shorter than equivalent JDBC programs.	With JDBC, all SQL statements must be wrapped in API calls that generally make for unclear and verbose code.



```
% sqlj emp.sqlj -user=scott/tiger
% java emp
```

Note: pre-processing does add complexity

Bonus: Code Comparison

Code Makes It Real

- A code comparison can be helpful, so am including an “apples-to-apples” comparison of the same function
- Specifically: the code provides methods to:
    1. Retrieve an employee’s address from the database
    2. Update the address
    3. Store the modified address in the database
- Note especially the “verbosity” difference

SQLJ Implementation

```java
public class SimpleDemoSQLJ {
  public Address getEmployeeAddress(int empno)
    throws SQLException {
    Address addr;
    #sql { SELECT office_addr INTO :addr FROM employees
        WHERE empnumber = :empno };
    return addr;
  }
  public Address updateAddress(Address addr)
    throws SQLException
  {
    #sql addr = { VALUES(UPDATE_ADDRESS(:addr)) };
    return addr;
  }
}
```

JDBC Implementation

```java
public class SimpleDemoJDBC {
  public Address getEmployeeAddress(int empno, Connection conn)
    throws SQLException {
    Address addr;
    PreparedStatement pstmt =
      conn.prepareStatement("SELECT office_addr FROM employees" +
       " WHERE empnumber = ?");
    pstmt.setInt(1, empno);
    OracleResultSet rs = (OracleResultSet)pstmt.executeQuery();
    rs.next();
    addr = (Address)rs.getORAData(1, Address.getORADataFactory());
    rs.close();
    pstmt.close();
    return addr;
  }
  public Address updateAddress(Address addr, Connection conn)
    throws SQLException {
    OracleCallableStatement cstmt = (OracleCallableStatement)
      conn.prepareCall("{ ? = call UPDATE_ADDRESS(?) }");
    cstmt.registerOutParameter(1, Address._SQL_TYPECODE, Address._SQL_NAME);
    if (addr == null) {
      cstmt.setNull(2, Address._SQL_TYPECODE, Address._SQL_NAME);
    }
    else {
      cstmt.setORAData(2, addr);
    }

    cstmt.executeUpdate();
    addr = (Address)cstmt.getORAData(1, Address.getORADataFactory());
    cstmt.close();
    return addr;
  }
}
```

Today’s Lecture: Wrapping it Up

Readings

- Textbook discussion: Chapter 5.1, somewhat limited IMNSHO