Advanced SQL 2

EECS 339

Lecture 8

Overview

- SQL
  - Join types
  - 3-valued logic
  - Strings and pattern matching
- Query execution
  - JDBC
  - Explain

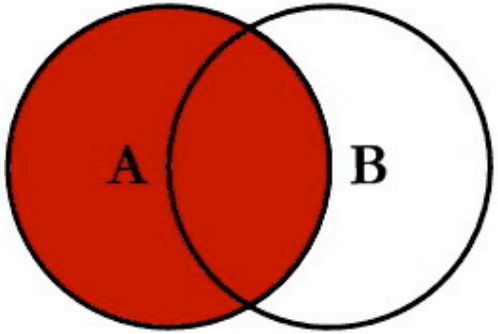
Join Types

- Joins needed to combine data from multiple tables and expressions
- Recall that join output concatenates the schemas of inputs

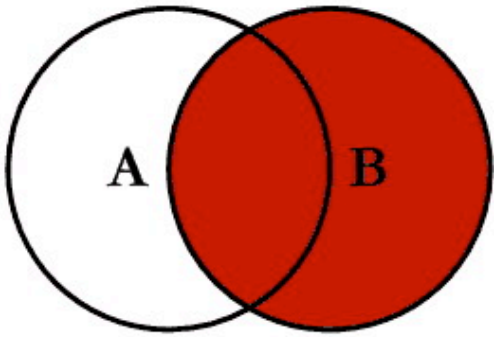
Joins

- Equi-join
  - Join on a specific attribute or set of attributes
- Natural join
  - Compare all common columns between 2 relations, select full equalities
- Theta join
  - Allows arbitrary comparisons between attributes, e.g., <, <=, etc.

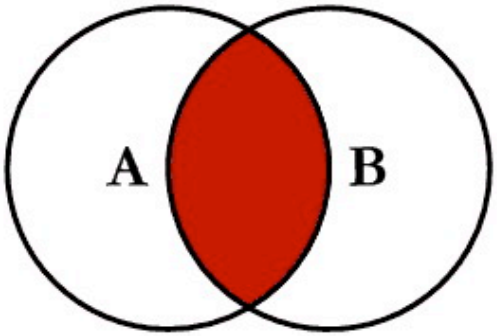
SQL JOINS



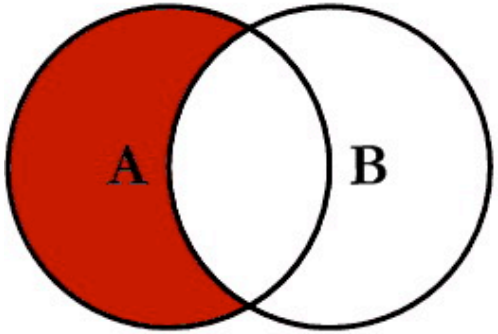

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



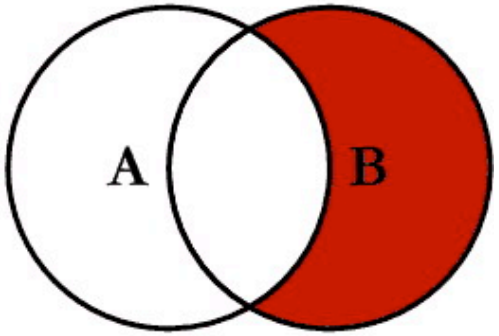

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



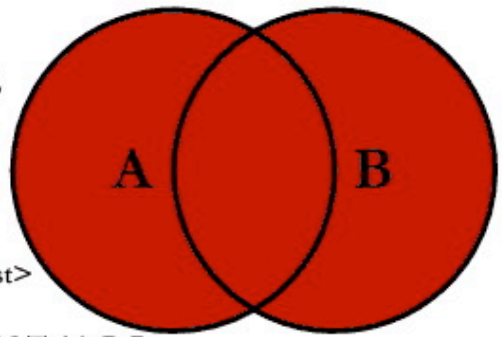

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



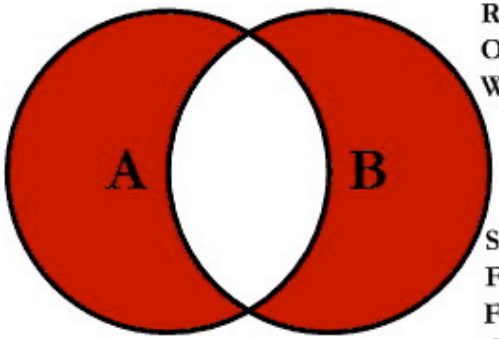

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Coalesce

- If there is a null, e.g., left join, sometimes want to replace it with a default value
- Do this with coalesce
- E.g., “find the credit card balance of all of a bank’s account holders. If no cc record exists, their debt is zero”.

```
SELECT acct_id, COALESCE(c.balance, 0)
FROM acct_holders LEFT JOIN
credit_cards c ON acct_id;
```

NULL Values

- Tuples in SQL relations can have NULL as a value for one or more components.
- Meaning depends on context. Two common cases:
  - *Missing value* : e.g., we know Joe's Bar has some address, but we don't know what it is.
  - *Inapplicable* : e.g., the value of attribute spouse for an unmarried person.

Comparing NULLs to Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- Comparing any value (including NULL itself) with NULL yields UNKNOWN.
- A tuple is in a query answer iff the WHERE clause is TRUE (not FALSE or UNKNOWN).

String Matching

- A condition can compare a string to a pattern by:
  - <Attribute> LIKE <pattern> or <Attribute> NOT LIKE <pattern>
- *Pattern* is a quoted string with % = “any string”; _ = “any character.”

LIKE

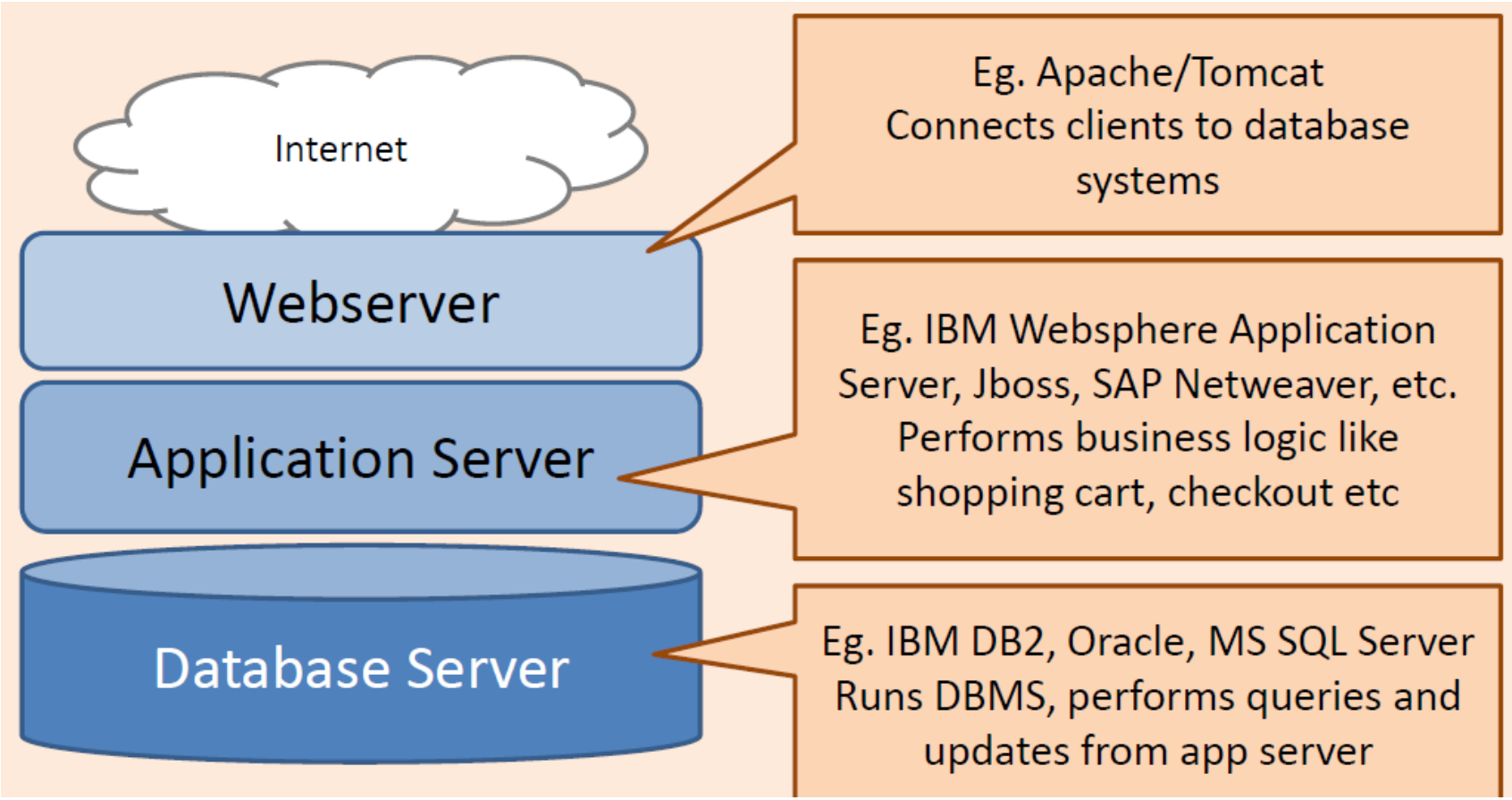
- Using Drinkers(name, addr, phone) find the drinkers with exchange 555:

```
SELECT name
FROM Drinkers
WHERE phone LIKE '%555-_ _ _ _';
```

SQL in Real Programs

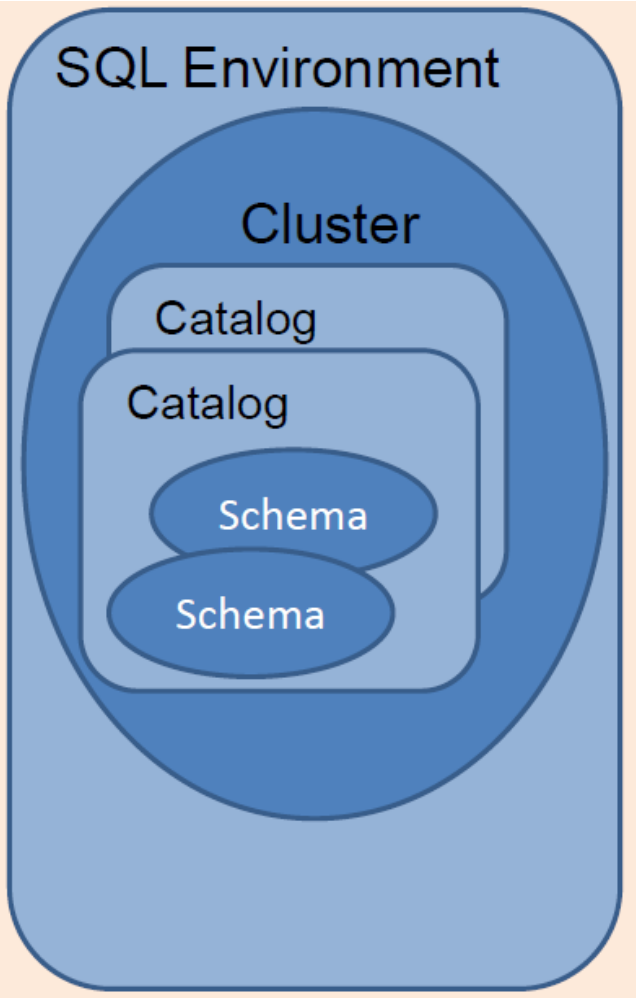
- We have seen only how SQL is used at the generic query interface --- an environment where we sit at a terminal and ask queries of a database.
- Reality is almost always different: conventional programs interacting with SQL.

Three Tier Architecture



Large, Internet based enterprises

Basic SQL Environment view…

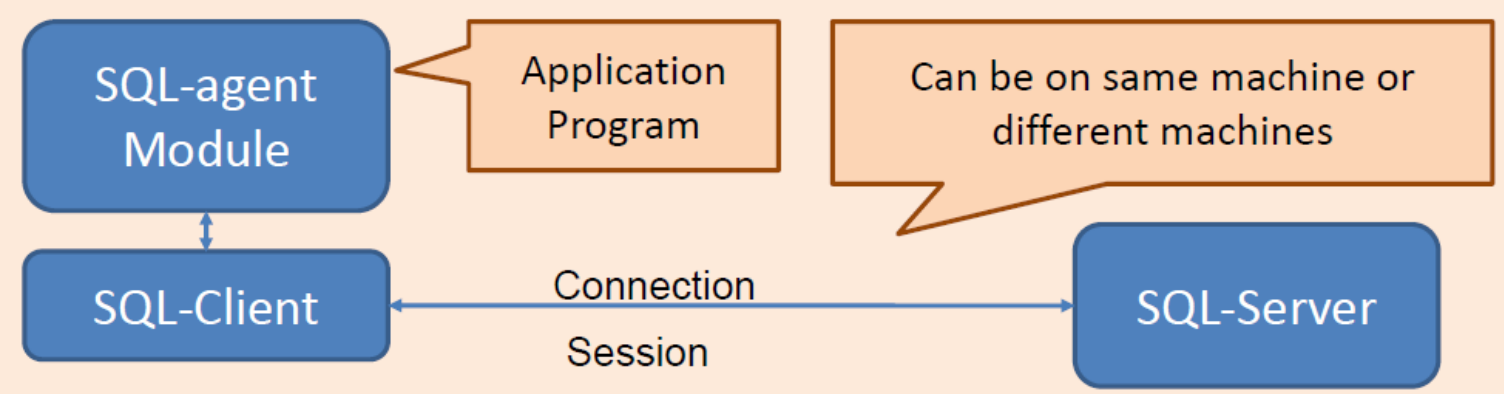


Schemas : tables, views, assertions, triggers

Catalogs : collection of schemas
(aka “databases” in DB2)

Clusters : collection of catalogs (aka “database instance” in DB2)

Typical Client-Server Model



•CONNECT TO <server> AS <connection name> AUTHORIZATION

•DISCONNECT/CONNECT RESET/TERMINATE

•**Session** – SQL operations performed while a connection is active

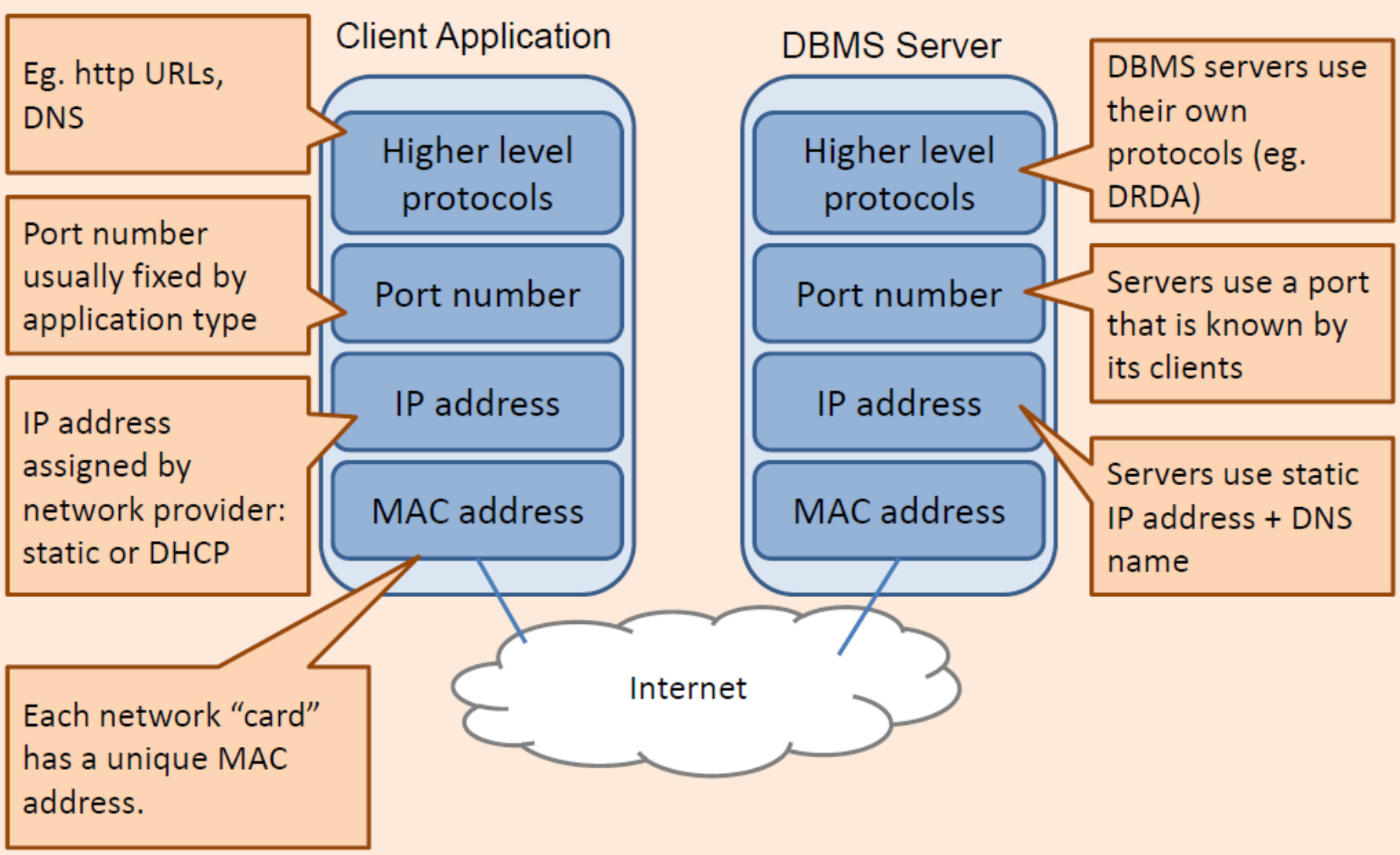
• Programming API:

–Generic SQL Interface
–Embedded SQL in a host language
–True Modules. Eg. Stored procedures.

Extremes of the integration spectrum:

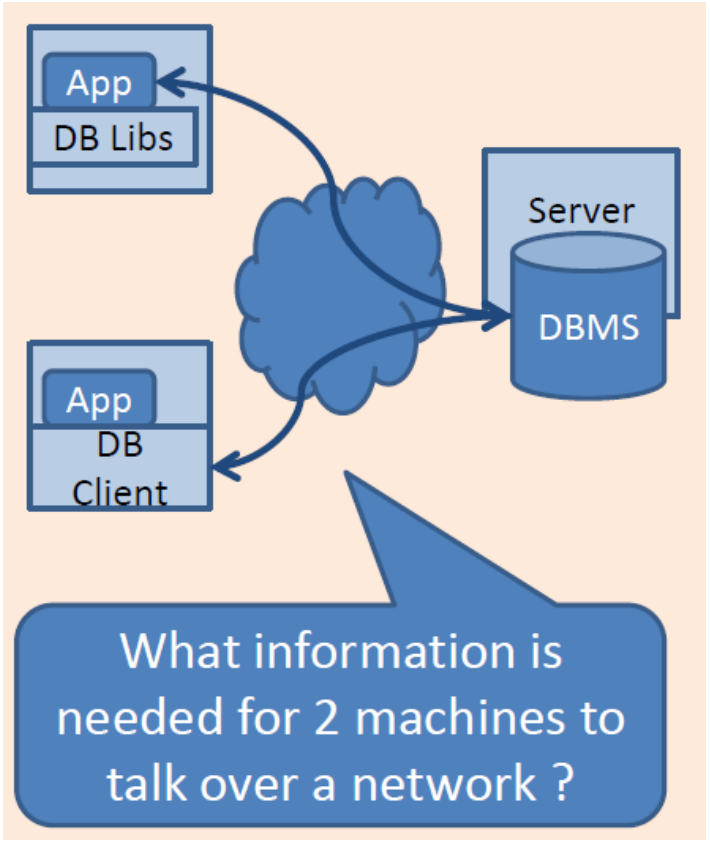
- Highly integrated (e.g. Microsoft linq)
  - Compiler checking of database operations
- Loosely integrated eg. ODBC & JDBC
  - Provides a way to call SQL from host language
  - Host language compiler doesn't understand database operations.
- Requirements:
- *Perform DB operations from host language*
- *DB operations need to access variables in host language*

In context of Networking



Remote Client Access

- Applications run on a machine that is separate from the DB server
- DBMS "thin" client
  - Libraries to which you link your app
  - App needs to know how to talk to DBMS server via network
- DBMS "full" client layer
  - Need to pre-configure the thick client layer to talk to DBMS server
  - Your app talks to a DBMS client layer as if it is talking to the server



Options

1. Code in a specialized language is stored in the database itself (e.g., PSM, PL/SQL).
2. SQL statements are embedded in a *host language* (e.g., C).
3. Connection tools are used to allow a conventional language to access a database (e.g., CLI, JDBC, PHP/DB).

Host/SQL Interfaces Via Libraries

- The third approach to connecting databases to conventional languages is to use library calls.
  1. C + CLI
  2. Java + JDBC
  3. PHP + PEAR/DB

Three-Tier Architecture

- A common environment for using a database has three tiers of processors:
  1. *Web servers* --- talk to the user.
  2. *Application servers* --- execute the business logic.
  3. *Database servers* --- get what the app servers need from the database.

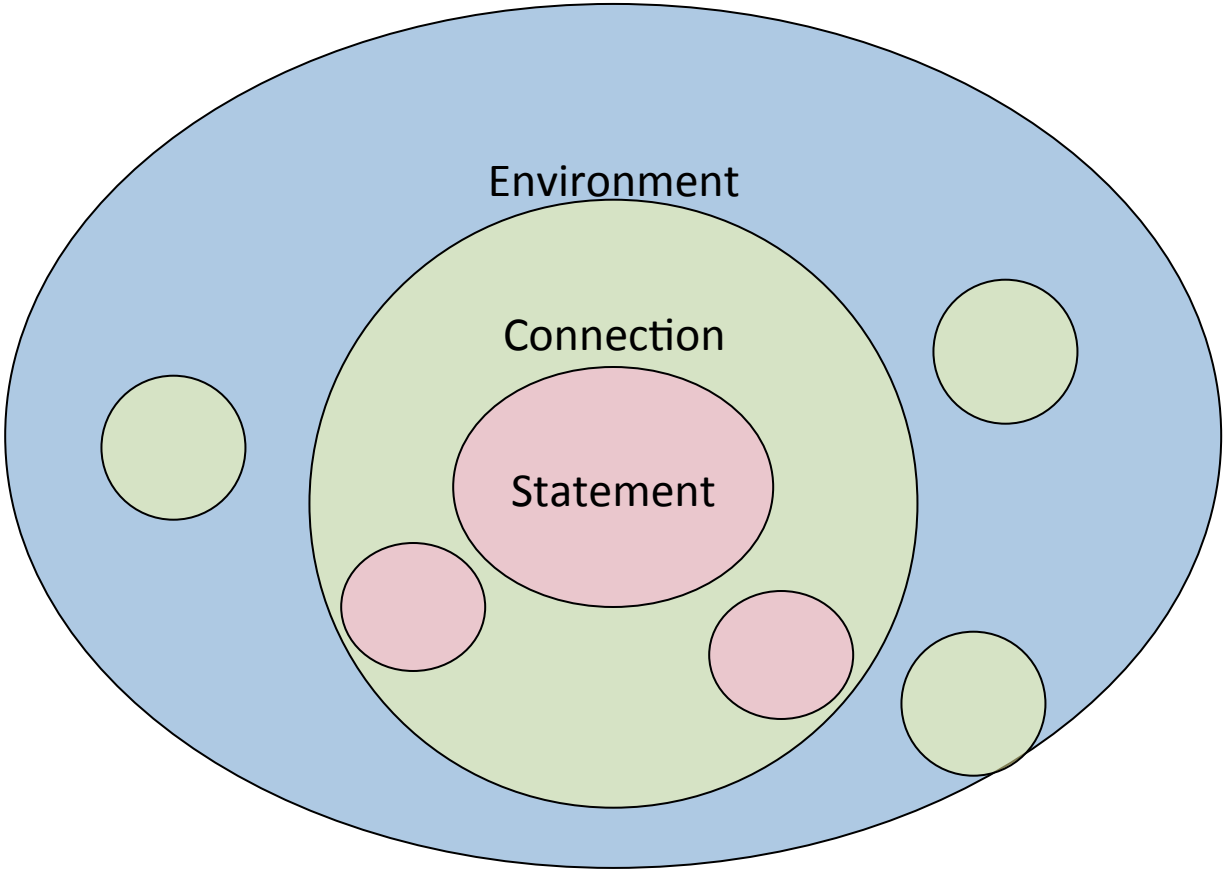
Example: Amazon

- Database holds the information about products, customers, etc.
- Business logic includes things like “what do I do after someone clicks ‘checkout’?”
  - Answer: Show the “how will you pay for this?” screen.

Environments, Connections, Queries

- The database is, in many DB-access languages, an *environment*.
- Database servers maintain some number of *connections*, so app servers can ask queries or perform modifications.
- The app server issues *statements* : queries and modifications, usually.

Diagram to Remember



SQL/CLI

- Instead of using a preprocessor (as in embedded SQL), we can use a library of functions.
  - The library for C is called SQL/CLI = "*Call-Level Interface*."
  - Embedded SQL's preprocessor will translate the EXEC SQL … statements into CLI or similar calls, anyway.

Data Structures

- C connects to the database by structs of the following types:

  1. ***Environments*** : represent the DBMS installation.
  2. ***Connections*** : logins to the database.
  3. ***Statements*** : SQL statements to be passed to a connection.
  4. ***Descriptions*** : records about tuples from a query, or parameters of a statement.

Handles

- Function SQLAllocHandle(T,I,O) is used to create these structs, which are called environment, connection, and statement *handles*.
  - *T* = type, e.g., SQL_HANDLE_STMT.
  - *I* = input handle = struct at next higher level (statement < connection < environment).
  - *O* = (address of) output handle.

Example: SQLAllocHandle

```
SQLAllocHandle(SQL_HANDLE_STMT,
  myCon, &myStat);
```

- `myCon` is a previously created connection handle.
- `myStat` is the name of the statement handle that will be created.

Preparing and Executing

- SQLPrepare(H, S, L) causes the string *S*, of length *L*, to be interpreted as a SQL statement and optimized; the executable statement is placed in statement handle *H*.
- SQLExecute(H) causes the SQL statement represented by statement handle *H* to be executed.

Example: Prepare and Execute

```
SQLPrepare(myStat, "SELECT beer,
  price FROM Sells
  WHERE bar = 'Joe''s Bar'",
  SQL_NTS);
SQLExecute(myStat);
```

This constant says the second argument is a "null-terminated string"; i.e., figure out the length by counting characters.

Direct Execution

- If we shall execute a statement *S* only once, we can combine PREPARE and EXECUTE with:

SQLExecuteDirect(H,S,L);

  - As before, *H* is a statement handle and *L* is the length of string *S*.

Fetching Tuples

- When the SQL statement executed is a query, we need to fetch the tuples of the result.
  - A cursor is implied by the fact we executed a query; the cursor need not be declared.
- SQLFetch(H) gets the next tuple from the result of the statement with handle *H*.

Accessing Query Results

- When we fetch a tuple, we need to put the components somewhere.
- Each component is bound to a variable by the function SQLBindCol.
  - This function has 6 arguments, of which we shall show only 1, 2, and 4:

    1 = handle of the query statement.

    2 = column number.

    4 = address of the variable.

Example: Binding

- Suppose we have just done SQLExecute(myStat), where myStat is the handle for query

```
SELECT beer, price FROM Sells
WHERE bar = 'Joe''s Bar'
```

- Bind the result to theBeer and thePrice:

```
SQLBindCol(myStat, 1, , &theBeer, , );
SQLBindCol(myStat, 2, , &thePrice, , );
```

Example: Fetching

- Now, we can fetch all the tuples of the answer by:

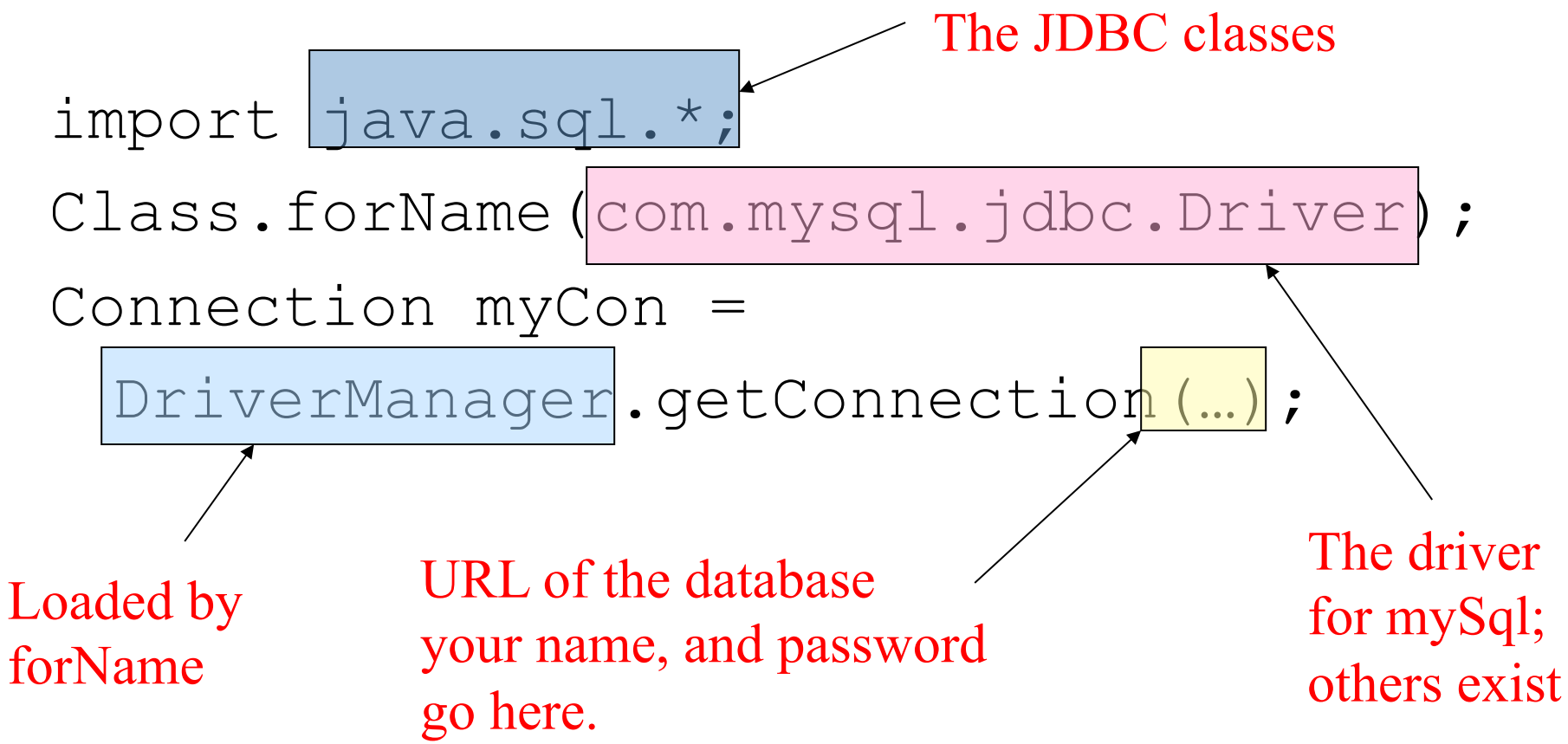
```
while ( SQLFetch(myStat) != SQL_NO_DATA)
   {
    /* do something with theBeer and
       thePrice */
   }
```

CLI macro representing SQLSTATE = 02000 = "failed to find a tuple."

JDBC

- *Java Database Connectivity* (JDBC) is a library similar to SQL/CLI, but with Java as the host language.
- Like CLI, but with a few notable differences, to be mentioned…

Making a Connection



Statements

- JDBC provides two classes:

  1. Statement = an object that can accept a string that is a SQL statement and can execute such a string.

  2. PreparedStatement = an object that has an associated SQL statement ready to execute.

Creating Statements

- The Connection class has methods to create Statements and PreparedStatements.

```
Statement stat1 = myCon.createStatement();
PreparedStatement stat2 =
   myCon.createStatement(
    "SELECT beer, price FROM Sells " +
    "WHERE bar = 'Joe' 's Bar' "
   );
```

createStatement with no argument returns a Statement; with one argument it returns a PreparedStatement.

Executing SQL Statements

- JDBC distinguishes queries from modifications, which it calls "updates."
- Statement and PreparedStatement each have methods executeQuery and executeUpdate.
  - For Statements: one argument: the query or modification to be executed.
  - For PreparedStatements: no argument.

Example: Update

- stat1 is a Statement.
- We can use it to insert a tuple as:

```
stat1.executeUpdate(
  ”INSERT INTO Sells ” +
  ”VALUES(’Brass Rail’,’Bud’,3.00)”
);
```

Example: Query

- stat2 is a PreparedStatement holding the query ”SELECT beer, price FROM Sells WHERE bar = ’Joe’’s Bar’ ”.
- executeQuery returns an object of class ResultSet – we’ll examine it later.
- The query:

```
ResultSet menu = stat2.executeQuery();
```

Accessing the ResultSet

- An object of type ResultSet is something like a cursor.
- Method next() advances the "cursor" to the next tuple.
  - The first time next() is applied, it gets the first tuple.
  - If there are no more tuples, next() returns the value false.

Accessing Components of Tuples

- When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet.
- Method get*X* (*i*), where *X* is some type, and *i* is the component number, returns the value of that component.
  - The value must have type *X*.

Example: Accessing Components

- Menu = ResultSet for query

  “SELECT beer, price FROM Sells WHERE bar = ’Joe’ ’s Bar’ ”.

- Access beer and price from each tuple by:

```
while ( menu.next() ) {
  theBeer = Menu.getString(1);
  thePrice = Menu.getFloat(2);
   /*something with theBeer and
  thePrice*/
}
```

Debugging Query Performance

- Sometimes a query does not perform as we expect
- Debug it without running by putting EXPLAIN in front of statement
- Debug it with runtime statistics using EXPLAIN ANALYZE

Conclusions

- SQL has advanced functionality for handling joins, nulls, and strings
- Connection tools are useful for embedded database calls into a program
- EXPLAIN enables us to debug query performance issues