MODULE – 3

SQL : ADVANCE QUERIES

DATABASE APPLICATION DEVELOPMENT

INTERNET APPLICATIONS

Mr. C. R. Belavi, Dept. of CSE, HIT, NDS

SQL : ADVANCE QUERIES

Constraints as Assertions

- General constraints: constraints that do not fit in the basic SQL categories (presented in chapter 8)
- Mechanism: `CREATE ASSERTION`
  - components include: a constraint name, followed by `CHECK`, followed by a condition

Assertions: An Example

- "The salary of an employee must not be greater than the salary of the manager of the department that the employee works for"

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS (SELECT *
 FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
 WHERE E.SALARY > M.SALARY AND
       E.DNO=D.NUMBER AND D.MGRSSN=M.SSN))
```

Using General Assertions

- Specify a query that violates the condition; include inside a `NOT EXISTS` clause
- Query result must be empty
  - if the query result is not empty, the assertion has been violated

SQL Triggers

- Objective: to monitor a database and take action when a condition occurs
- Triggers are expressed in a syntax similar to assertions and include the following:
  - event (e.g., an update operation)
  - condition
  - action (to be taken when the condition is satisfied)

SQL Triggers: An Example

- A trigger to compare an employee's salary to his/her supervisor during insert or update operations:

```
CREATE TRIGGER INFORM_SUPERVISOR
BEFORE INSERT OR UPDATE OF
   SALARY, SUPERVISOR_SSN ON EMPLOYEE
   FOR EACH ROW
        WHEN
        (NEW.SALARY> (SELECT SALARY FROM EMPLOYEE
                      WHERE SSN=NEW.SUPERVISOR_SSN))
        INFORM_SUPERVISOR (NEW.SUPERVISOR_SSN,NEW.SSN;
```

Views in SQL

- A view is a “virtual” table that is derived from other tables
- Allows for limited update operations (since the table may not physically be stored)
- Allows full query operations
- A convenience for expressing certain operations

Specification of Views

- SQL command: `CREATE VIEW`
  - a table (view) name
  - a possible list of attribute names (for example, when arithmetic operations are specified or when we want the names to be different from the attributes in the base relations)
  - a query to specify the table contents

SQL Views: An Example

- Specify a different WORKS_ON table

```
CREATE VIEW WORKS_ON_NEW AS
SELECT FNAME, LNAME, PNAME, HOURS
    FROM EMPLOYEE, PROJECT, WORKS_ON
    WHERE SSN=ESSN AND PNO=PNUMBER
    GROUP BY PNAME;
```

Using a Virtual Table

- We can specify SQL queries on a newly create table (view):

```
SELECT FNAME, LNAME FROM WORKS_ON_NEW
WHERE PNAME='Seena';
```

- When no longer needed, a view can be dropped:

```
DROP VIEW WORKS_ON_NEW;
```

Efficient View Implementation

- Query modification: present the view query in terms of a query on the underlying base tables
  - disadvantage: inefficient for views defined via complex queries (especially if additional queries are to be applied to the view within a short time period)

Efficient View Implementation

- View materialization: involves physically creating and keeping a temporary table
  - assumption: other queries on the view will follow
  - concerns: maintaining correspondence between the base table and the view when the base table is updated
  - strategy: incremental update

View Update

- Update on a single view without aggregate operations: update may map to an update on the underlying base table
- Views involving joins: an update *may* map to an update on the underlying base relations
  - not always possible

Un-updatable Views

- Views defined using groups and aggregate functions are not updateable
- Views defined on multiple tables using joins are generally not updateable
- `WITH CHECK OPTION`: must be added to the definition of a view if the view is to be updated
  - to allow check for updatability and to plan for an execution strategy

DATABASE APPLICATION DEVELOPMENT

Justification for access to databases via programming languages :

- SQL is a direct query language; as such, it has limitations.
- via programming languages :
  - Complex computational processing of the data.
  - Specialized user interfaces.
  - Access to more than one database at a time.

SQL in Application Code

- SQL commands can be called from within a host language (e.g., C++ or Java) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include a statement to *connect* to the right database.

SQL in Application Code (Contd.)

Impedance mismatch:

- SQL relations are (multi-) sets of records, with no *a priori* bound on the number of records. No such data structure exist traditionally in procedural programming languages such as C++. (Though now: STL)
  - SQL supports a mechanism called a *cursor* to handle this.

Desirable features of such systems:

- Ease of use.
- Conformance to standards for existing programming languages, database query languages, and development environments.
- Interoperability: the ability to use a common interface to diverse database systems on different operating systems

Vendor specific solutions

- Oracle PL/SQL: A proprietary PL/1-like language which supports the execution of SQL queries:
- Advantages:
  - Many Oracle-specific features, not common to other systems, are supported.
  - Performance may be optimized to Oracle based systems.
- Disadvantages:
  - Ties the applications to a specific DBMS.
  - The application programmer must depend upon the vendor for the application development environment.
  - It may not be available for all platforms.

Vendor Independent solutions based on SQL

There are three basic strategies which may be considered:

- Embed SQL in the host language (Embedded SQL, SQLJ)
- SQL modules
- SQL call level interfaces

Embedded SQL

- Approach: Embed SQL in the host language.
  - A preprocessor converts the SQL statements into special API calls.
  - Then a regular compiler is used to compile the code.

- Language constructs:
  - Connecting to a database:
    EXEC SQL CONNECT
  - Declaring variables:
    EXEC SQL BEGIN (END) DECLARE SECTION
  - Statements:
    EXEC SQL Statement;

Embedded SQL: Variables


```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

- Two special "error" variables:
  - SQLCODE (long, is negative if an error has occurred)
  - SQLSTATE (char[6], predefined codes for common errors)

Cursors

- Can declare a cursor on a relation or query statement (which generates a relation).
- Can *open* a cursor, and repeatedly *fetch* a tuple then *move* the cursor, until all tuples have been retrieved.
  - Can use a special clause, called ORDER BY, in queries that are accessed through a cursor, to control the order in which tuples are returned.
    - Fields in ORDER BY clause must also appear in SELECT clause.
  - The ORDER BY clause, which orders answer tuples, is *only* allowed in the context of a cursor.
- Can also modify/delete tuple pointed to by a cursor.

Cursor that gets names of sailors who've reserved a red boat, in alphabetical order


```
EXEC SQL DECLARE sinfo CURSOR FOR
        SELECT  S.sname
        FROM  Sailors S, Boats B, Reserves R
        WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
        ORDER BY  S.sname
```

- Note that it is illegal to replace *S.sname* by, say, *S.sid* in the ORDER BY clause! (Why?)
- Can we add *S.sid* to the SELECT clause and replace *S.sname* by *S.sid* in the ORDER BY clause?

Embedding SQL in C: An Example


```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT S.sname, S.age            FROM Sailors S
    WHERE S.rating > :c_minrating
    ORDER BY S.sname;
do {
    EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
    printf("%s is %d years old\n", c_sname, c_age);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;
```

Dynamic SQL

- SQL query strings are not always known at compile time (e.g., spreadsheet, graphical DBMS frontend): Allow construction of SQL statements on-the-fly

- Example:

```
char c_sqlstring[]=
   {“DELETE FROM Sailors WHERE rating>5”};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

Disadvantages:

- It is a real pain to debug preprocessed programs.
- The use of a program-development environment is compromised substantially.
- The preprocessor must be vendor and platform specific.

SQL Modules

- In the module approach, invocations to SQL are made via libraries of procedures , rather than via preprocessing
- Special standardized interface: procedures/objects
- Pass SQL strings from language, presents result sets in a language-friendly way
- Supposedly DBMS-neutral
  - a “driver” traps the calls and translates them into DBMS-specific code
  - database can be across a network

Example module based

- Sun's *JDBC*: Java API
- Part of the java.sql package

- Advantages over embedded SQL:
  - Clean separation of SQL from the host programming language.
  - Debugging is much more straightforward, since no preprocessor is involved.

- Disadvantages:
  - The module libraries are specific to the programming language and environment. Thus, portability is compromised greatly.

JDBC: Architecture

- Four architectural components:
  - Application (initiates and terminates connections, submits SQL statements)
  - Driver manager (load JDBC driver)
  - Driver (connects to data source, transmits requests and returns/translates results and error codes)
  - Data source (processes SQL statements)

JDBC Architecture (Contd.)

Four types of drivers:

Bridge:

- Translates SQL commands into non-native API. Example: JDBC-ODBC bridge. Code for ODBC and JDBC driver needs to be available on each client.

Direct translation to native API, non-Java driver:

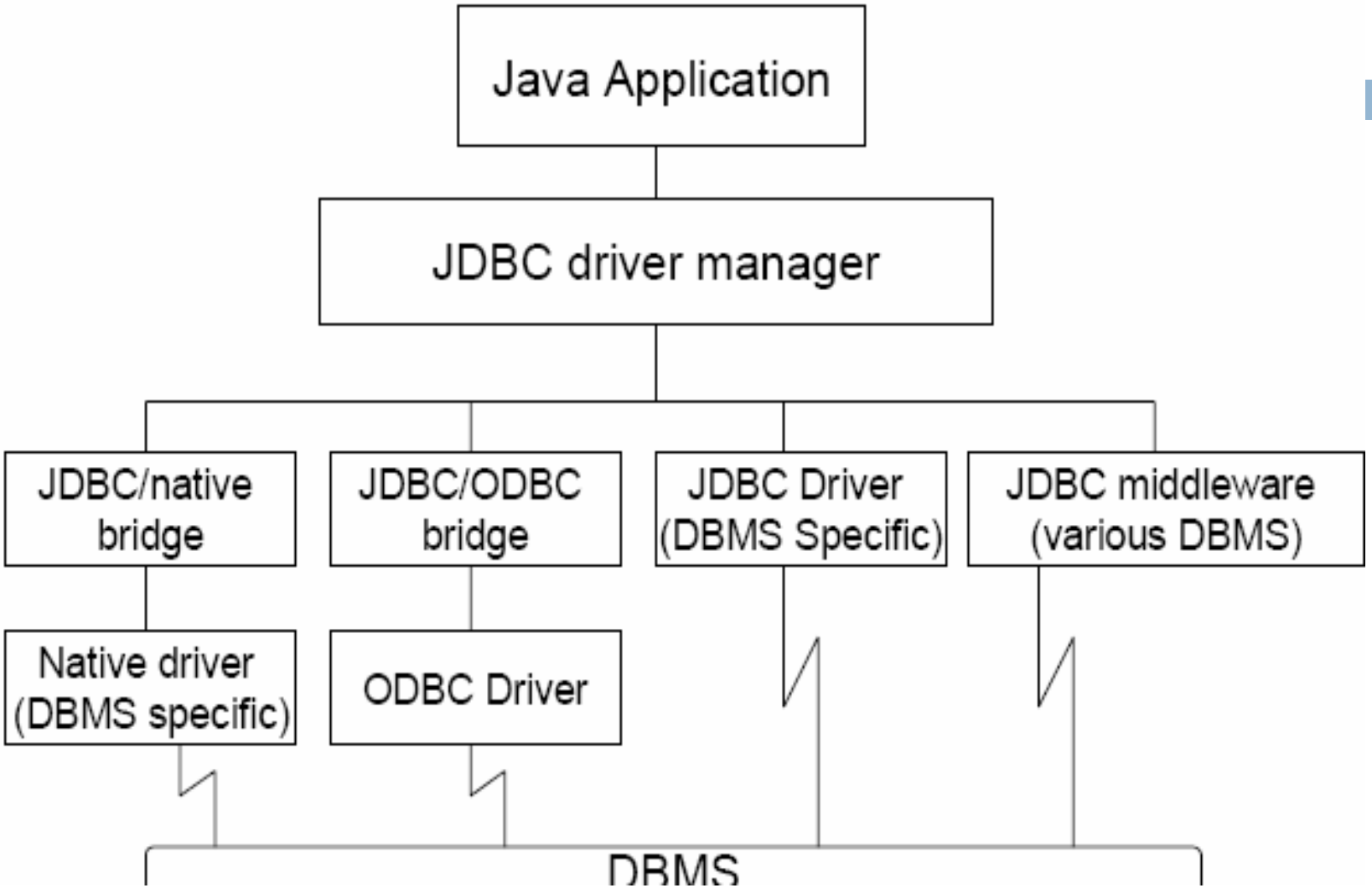
- Translates SQL commands to native API of data source. Need OS-specific binary on each client.

Network bridge:

- Send commands over the network to a middleware server that talks to the data source. Needs only small JDBC driver at each client.

Direction translation to native API via Java driver:

- Converts JDBC calls directly to network protocol used by DBMS. Needs DBMS-specific Java driver at each client.


Java Application
JDBC driver manager
JDBC/native bridge
JDBC/ODBC bridge
JDBC Driver (DBMS Specific)
JDBC middleware (various DBMS)
Native driver (DBMS specific)
ODBC Driver
DBMS

JDBC Classes and Interfaces

Steps to submit a database query:

- Load the JDBC driver
- Connect to the data source
- Execute SQL statements
- Process the results returned by DBMS
- Terminate the connection

JDBC Driver Management

- All drivers are managed by the DriverManager class
- Loading a JDBC driver:
  - In the Java code:
    Class.forName("oracle/jdbc.driver.Oracledriver");
  - When starting the Java application:
    -Djdbc.drivers=oracle/jdbc.driver

Connections in JDBC

We interact with a data source through sessions. Each connection identifies a logical session.

- JDBC URL:
  jdbc:<subprotocol>:<otherParameters>

Example:

```
String url="jdbc:oracle:www.bookstore.com:3083";
Connection con;
try{
    con = DriverManager.getConnection(url,usedId,password);
} catch SQLException excpt { …}
```

Connection Class Interface

- public int getTransactionIsolation() and
  void setTransactionIsolation(int level)
  Gets/Sets isolation level for the current connection.

- public boolean getReadOnly() and
  void setReadOnly(boolean b)
  Specifies if transactions in this connection are read-only

- public boolean getAutoCommit() and
  void setAutoCommit(boolean b)
  If autocommit is set, then each SQL statement is considered its own transaction. Otherwise, a transaction is committed using commit(), or aborted using rollback().

- public boolean isClosed()
  Checks whether connection is still open.

Executing SQL Statements

- Three different ways of executing SQL statements:
  - Statement (both static and dynamic SQL statements)
  - PreparedStatement (semi-static SQL statements)
  - CallableStatment (stored procedures)

- PreparedStatement class:
  Precompiled, parametrized SQL statements:
  - Structure is fixed
  - Values of parameters are determined at run-time

Executing SQL Statements (Contd.)


```
String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatment pstmt=con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setInt(1,sid);
pstmt.setString(2,sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4,age);

// we know that no rows are returned, thus we use
   executeUpdate()
int numRows = pstmt.executeUpdate();
```

ResultSets

- PreparedStatement.executeUpdate only returns the number of affected records
- PreparedStatement.executeQuery returns data, encapsulated in a ResultSet object (a cursor)

```
ResultSet rs=pstmt.executeQuery(sql);
// rs is now a cursor
While (rs.next()) {
  // process the data
}
```

ResultSets (Contd.)

A ResultSet is a very powerful cursor:

- previous(): moves one row back
- absolute(int num): moves to the row with the specified number
- relative (int num): moves forward or backward
- first() and last()

Matching Java and SQL Data Types

SQL Type	Java class	ResultSet get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

Examining Database Metadata

DatabaseMetaData object gives information about the database system and the catalog.

```
DatabaseMetaData md = con.getMetaData();
// print information about the driver:
System.out.println(
   "Name:" + md.getDriverName() +
   "version: " + md.getDriverVersion());
```

Database Metadata (Contd.)


```
DatabaseMetaData md=con.getMetaData();
ResultSet trs=md.getTables(null,null,null,null);
String tableName;
While(trs.next()) {
    tableName = trs.getString("TABLE_NAME");
    System.out.println("Table: " + tableName);
    //print all attributes
    ResultSet crs = md.getColumns(null,null,tableName, null);
    while (crs.next()) {
        System.out.println(crs.getString("COLUMN_NAME" + ", ");
    }
}
```

A (Semi-)Complete Example


```
import java.sql.*;

/**
 * This is a sample program with jdbc odbc  Driver
*/
public class localdemo {

  public static void main(String[] args) {
    try {

      // Register JDBC/ODBC Driver in jdbc DriverManager
      // On some platforms with some java VMs, newInstance() is necessary...
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();

      // Test with MS Access database (sailors ODBC data source)
      String url = "jdbc:odbc:mysailors";

      java.sql.Connection c = DriverManager.getConnection(url);
```

A (Semi-)Complete Example cont


```
java.sql.Statement st = c.createStatement();
    java.sql.ResultSet rs = st.executeQuery("select * from Sailors");

    java.sql.ResultSetMetaData md = rs.getMetaData();
    while(rs.next()) {
      System.out.print("\nTUPLE: | ");
      for(int i=1; i<= md.getColumnCount(); i++) {
        System.out.print(rs.getString(i) + " | ");
      }
    }
    rs.close();
  } catch(Exception e) {
    e.printStackTrace();
  }
 }
};
```

SQLJ

Complements JDBC with a (semi-)static query model: Compiler can perform syntax checks, strong type checks, consistency of the query with the schema

- All arguments always bound to the same variable:

```
#sql x = {
    SELECT name, rating INTO :name, :rating
    FROM Books WHERE sid = :sid;
```

- Compare to JDBC:

```
sid=rs.getInt(1);
if (sid==1) {sname=rs.getString(2);}
else { sname2=rs.getString(2);}
```

- SQLJ (part of the SQL standard) versus embedded SQL (vendor-specific)

SQLJ Code


```
Int sid; String name; Int rating;
// named iterator
#sql iterator Sailors(Int sid, String name, Int rating);
Sailors sailors;
// assume that the application sets rating
#sailors = {
    SELECT sid, sname INTO :sid, :name
     FROM Sailors WHERE rating = :rating
};
// retrieve results
while (sailors.next()) {
    System.out.println(sailors.sid + " " + sailors.sname));
}
sailors.close();
```

SQLJ Iterators

Two types of iterators ("cursors"):

- Named iterator
  - Need both variable type and name, and then allows retrieval of columns by name.
  - See example on previous slide.
- Positional iterator
  - Need only variable type, and then uses FETCH .. INTO construct:

```
#sql iterator Sailors(Int, String, Int);
Sailors sailors;
#sailors = …
while (true) {
    #sql {FETCH :sailors INTO :sid, :name} ;
    if (sailors.endFetch()) { break; }
    // process the sailor
}
```

SQL call level interfaces

- A call-level interface provides a library of functions for access to DBMS's.
- The DBMS drivers are stored separately; thus the library used by the programming language is DBMS independent.
- The programming language functions provided only an interface to the DBMS drivers.

SQL call level interfaces

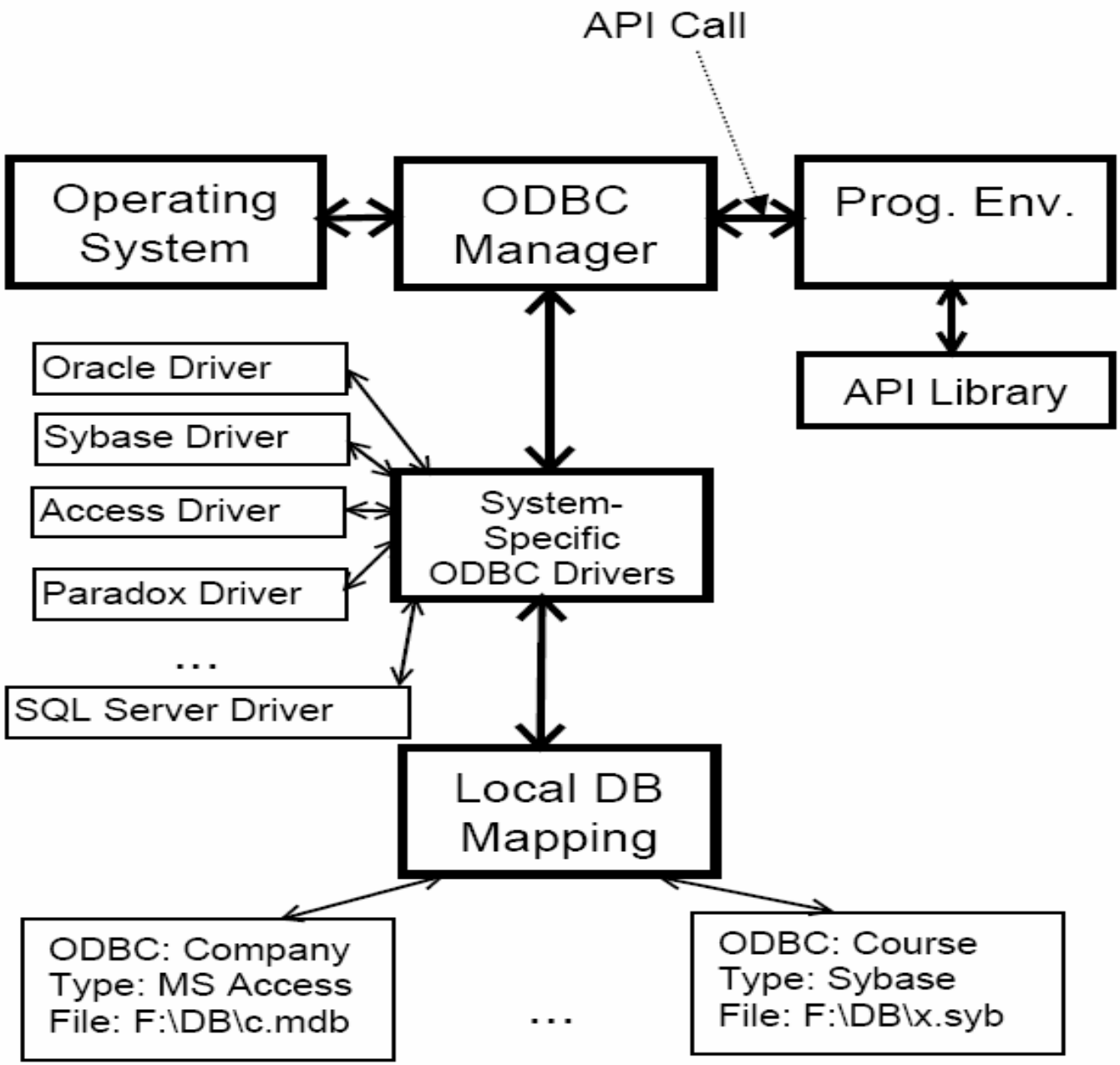
- Advantages:
  - The development environment is not tied to a particular DBMS, operating sytem, or even a particular development environment.
- Disadvantages:
  - Some low-level optimization may be more difficult or impossible to achieve.

Key example:

- The SQL CLI (X/Open CLI)
- Microsoft ODBC (Open Database Connectivity)
- · The two are closely aligned.

Open DataBase Connectivity

- Shorten to ODBC, a standard database access method
- The goal: make it possible to access any data from any application, regardless of which (DBMS).
- ODBC manages this by inserting a middle layer, called a database *driver* , between an application and the DBMS.
- The purpose of this layer is to translate the application's data queries into commands that the DBMS understands.
- For this to work, both the application and the DBMS must be ODBC-compliant -- that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.


API Call
Operating System
ODBC Manager
Prog. Env.
API Library
Oracle Driver
Sybase Driver
Access Driver
Paradox Driver
...
SQL Server Driver
System-Specific ODBC Drivers
Local DB Mapping
ODBC: Company
Type: MS Access
File: F:\DB\c.mdb
...
ODBC: Course
Type: Sybase
File: F:\DB\x.syb

Configuring a datasource (Access) under Windows

- Open the ODBC menu in the control panel.
- Click on the User DSN tab.
  - click on Add.
- From the menu in the new window,
  - select Microsoft Access Driver (sailors.mdb),
  - click on Finish.
- From the menu in the new window,
  - type in a data source name (mysailors), and optionally, a description.
  - Then click on either Select or Create, depending upon whether you want to link to an existing database, or create a new blank one.
- In the new window, give the path to the database.
- "OK" away the pile of subwindows; the new database should appear under the top-level ODBC User DSN tab.

```
// program connects to an ODBC data source called “mysailors“ then executes SQL statement
     “SELECT *  FROM Sailors ;"
#include <windows.h>
#include <sqlext.h>
#include <stdio.h>
```



```
int main(void)
{
     HENV    hEnv = NULL;                                  // Env Handle from SQLAllocEnv()
     HDBC    hDBC = NULL;                                  // Connection handle
     HSTMT   hStmt = NULL;                                 // Statement handle
     UCHAR  szDSN[SQL_MAX_DSN_LENGTH] = “mysailors";// Data Source Name buffer
     UCHAR*                szUID = NULL;                                  // User ID buffer
     UCHAR*                szPasswd = NULL;                               // Password buffer
     UCHAR  szname[255];                                   //  buffer
     SDWORD               cbname;                                        // bytes recieved
     UCHAR  szSqlStr[] = "Select * From Sailors”;          // SQL string
     RETCODE              retcode;                                       // Return code

// Allocate memory for ODBC Environment handle
SQLAllocEnv (&hEnv);

// Allocate memory for the connection handle
SQLAllocConnect (hEnv, &hDBC);
```

```
// Connect to the data source “mysailors" using userid and password.
retcode = SQLConnect (hDBC, szDSN, SQL_NTS, szUID, SQL_NTS, szPasswd, SQL_NTS);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
    {
     // Allocate memory for the statement handle
     retcode = SQLAllocStmt (hDBC, &hStmt);

     // Prepare the SQL statement by assigning it to the statement handle
     retcode = SQLPrepare (hStmt, szSqlStr, sizeof (szSqlStr));

     // Execute the SQL statement handle
     retcode = SQLExecute (hStmt);

     // Project only column 2 which is the name
     SQLBindCol (hStmt, 2, SQL_C_CHAR, szname, sizeof(szname), &cbModel);

     // Get row of data from the result set defined above in the statement
     retcode = SQLFetch (hStmt);
```

```
while (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        {       printf ("\t%s\n", szname);          // Print row (sname)
                retcode = SQLFetch (hStmt);         // Fetch next row from result set
        }

        // Free the allocated statement handle
        SQLFreeStmt (hStmt, SQL_DROP);

        // Disconnect from datasource
        SQLDisconnect (hDBC);
      }

      // Free the allocated connection handle
      SQLFreeConnect (hDBC);

      // Free the allocated ODBC environment handle
      SQLFreeEnv (hEnv);
      return 0;
}
```

Stored Procedures

- What is a stored procedure:
  - Program executed through a single SQL statement
  - Executed in the process space of the server
- Advantages:
  - Can encapsulate application logic while staying "close" to the data
  - Reuse of application logic by different users
  - Avoid tuple-at-a-time return of records through cursors

Stored Procedures: Examples


```
CREATE PROCEDURE ShowNumReservations
    SELECT S.sid, S.sname, COUNT(*)
    FROM Sailors S, Reserves R
    WHERE S.sid = R.sid
    GROUP BY S.sid, S.sname
```

Stored procedures can have parameters:

- Three different modes: IN, OUT, INOUT

```
CREATE PROCEDURE IncreaseRating(
    IN sailor_sid INTEGER, IN increase INTEGER)
UPDATE Sailors
    SET rating = rating + increase
    WHERE sid = sailor_sid
```

Stored Procedures: Examples (Contd.)

Stored procedure do not have to be written in SQL:

```
CREATE PROCEDURE TopSailors(
    IN num INTEGER)
LANGUAGE JAVA
EXTERNAL NAME “file:///c:/storedProcs/rank.jar”
```

Calling Stored Procedures


```
EXEC SQL BEGIN DECLARE SECTION
Int sid;
Int rating;
EXEC SQL END DECLARE SECTION

// now increase the rating of this sailor
EXEC CALL IncreaseRating(:sid,:rating);
```

Calling Stored Procedures (Contd.)

JDBC:

```
CallableStatement cstmt=
    con.prepareCall("{call
    ShowSailors});
ResultSet rs =
    cstmt.executeQuery();
while (rs.next()) {
   …
}
```

SQLJ:

```
#sql iterator
    ShowSailors(…);
ShowSailors showsailors;
#sql showsailors={CALL
    ShowSailors};
while (showsailors.next()) {
    …
}
```

SQL/PSM

Most DBMSs allow users to write stored procedures in a simple, general-purpose language (close to SQL) → SQL/PSM standard is a representative

Declare a stored procedure:

CREATE PROCEDURE name(p1, p2, …, pn)
    local variable declarations
    procedure code;

Declare a function:

CREATE FUNCTION name (p1, …, pn) RETURNS sqlDataType
    local variable declarations
    function code;

Main SQL/PSM Constructs


```
CREATE FUNCTION rate Sailor
        (IN sailorId INTEGER)
        RETURNS INTEGER
DECLARE rating INTEGER
DECLARE numRes INTEGER
SET numRes = (SELECT COUNT(*)
                        FROM Reserves R
                        WHERE R.sid = sailorId)
IF (numRes > 10) THEN rating =1;
ELSE rating = 0;
END IF;
RETURN rating;
```

Main SQL/PSM Constructs (Contd.)

- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:
  - IF (condition) THEN statements;
    ELSEIF (condition) statements;
    … ELSE statements; END IF;
  - LOOP statements; END LOOP
- Queries can be parts of expressions
- Can use cursors naturally without "EXEC SQL"

INTERNET APPLICATIONS

- Internet Concepts
- Web data formats
  - HTML, XML, DTDs
- Introduction to three-tier architectures
- The presentation layer
  - HTML forms; HTTP Get and POST, URL encoding; Javascript; Stylesheets. XSLT
- The middle tier
  - CGI, application servers, Servlets, JavaServerPages, passing arguments, maintaining state (cookies)

Uniform Resource Identifiers

- Uniform naming schema to identify *resources on the* Internet

- A resource can be anything:
  - Index.html
  - mysong.mp3
  - picture.jpg

- Example URIs:
  - http://www.cs.wisc.edu/~dbbook/index.html
  - mailto:webmaster@bookstore.com

Structure of URIs

- http://www.cs.wisc.edu/~dbbook/index.html

- URI has three parts:
  - Naming schema (http)
  - Name of the host computer (www.cs.wisc.edu)
  - Name of the resource (~dbbook/index.html)

- URLs are a subset of URIs

Hypertext Transfer Protocol

- What is a communication protocol?
  - Set of standards that defines the structure of messages
  - Examples: TCP, IP, HTTP
- What happens if you click on
  - www.cs.wisc.edu/~dbbook/index.html?
- Client (web browser) sends HTTP request to server
- Server receives request and replies
- Client receives reply; makes new requests

HTTP (Contd.)

Client to Server:

GET ~/index.html HTTP/1.1
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg

Server replies:

HTTP/1.1 200 OK
Date: Mon, 04 Mar 2002 12:00:00 GMT
Server: Apache/1.3.0 (Linux)
Last-Modified: Mon, 01 Mar 2002 09:23:24 GMT
Content-Length: 1024
Content-Type: text/html
<HTML> <HEAD></HEAD>
<BODY>
<h1>Barns and Nobble Internet Bookstore</h1>
Our inventory:
<h3>Science</h3>
The Character of Physical Law
...

HTTP Protocol Structure

- HTTP Requests
- Request line: GET ~/index.html HTTP/1.1
  - GET: Http method field (possible values are GET and POST
  - ~/index.html: URI field
  - HTTP/1.1: HTTP version field
- Type of client: User-agent: Mozilla/4.0
- What types of files will the client accept:
  - Accept: text/html, image/gif, image/jpeg

HTTP Protocol Structure (Contd.)

- HTTP Responses
  - Status line: HTTP/1.1 200 OK
  - HTTP version: HTTP/1.1
  - Status code: 200
  - Server message: OK
  - Common status code/server message combinations:
    - 200 OK: Request succeeded
    - 400 Bad Request: Request could not be fulfilled by the server
    - 404 Not Found: Requested object does not exist on the server
    - 505 HTTP Version not Supported
- Date when the object was created:
  - Last-Modified: Mon, 01 Mar 2002 09:23:24 GMT
- Number of bytes being sent: Content-Length: 1024
- What type is the object being sent: Content-Type: text/html
- Other information such as the server type, server time, etc.

Some Remarks About HTTP

- HTTP is stateless
  - No “sessions”
  - Every message is completely self-contained
  - No previous interaction is “remembered” by the protocol
  - Tradeoff between ease of implementation and ease of application development: Other functionality has to be built on top

- Implications for applications:
  - Any state information (shopping carts, user login-information) need to be encoded in every HTTP request and response!
  - Popular methods on how to maintain state:
    - Cookies
    - Dynamically generate unique URL’s at the server level

Web Data Formats

- HTML
  - The presentation language for the Internet
- Xml
  - A self-describing, hierarchal data model
- DTD
  - Standardizing schemas for Xml
- XSLT

HTML: An Example


```
<HTML>
<HEAD></HEAD>
<BODY>
<h1>Barns and Nobble Internet Bookstore</h1>
Our inventory:
<h3>Science</h3>
<b>The Character of Physical Law</b>
<UL>
<LI>Author: Richard Feynman</LI>
<LI>Published 1980</LI>
<LI>Hardcover</LI>
</UL>
<h3>Fiction</h3>
<b>Waiting for the Mahatma</b>
<UL>
<LI>Author: R.K. Narayan</LI>
<LI>Published 1981</LI>
</UL>
<b>The English Teacher</b>
<UL>
<LI>Author: R.K. Narayan</LI>
<LI>Published 1980</LI>
<LI>Paperback</LI>
</UL>
</BODY>
</HTML>
```

HTML: A Short Introduction

- HTML is a markup language
- Commands are tags:
  - Start tag and end tag
  - Examples:
    - <HTML> … </HTML>
    - <UL> … </UL>
- Many editors automatically generate HTML directly from your document (e.g., Microsoft Word has an “Save as html” facility)

HTML: Sample Commands

- <HTML>:
- <UL>: unordered list
- <LI>: list entry
- <h1>: largest heading
- <h2>: second-level heading, <h3>, <h4> analogous
- <B>Title</B>: Bold

XML: An Example


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<BOOKLIST>
<BOOK genre="Science" format="Hardcover">
<AUTHOR>
<FIRSTNAME>Richard</FIRSTNAME><LASTNAME>Feynman</LASTNAME>
</AUTHOR>
<TITLE>The Character of Physical Law</TITLE>
<PUBLISHED>1980</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
<AUTHOR>
<FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
</AUTHOR>
<TITLE>Waiting for the Mahatma</TITLE>
<PUBLISHED>1981</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
<AUTHOR>
<FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
</AUTHOR>
<TITLE>The English Teacher</TITLE>
<PUBLISHED>1980</PUBLISHED>
</BOOK>
</BOOKLIST>
```

XML – The Extensible Markup Language

- Language
  - A way of communicating information
- Markup
  - Notes or meta-data that describe your data or language
- Extensible
  - Limitless ability to define new languages or data sets

XML – What's The Point?

- You can include your data and a description of what the data represents
  - This is useful for defining your own language or protocol
- Example: Chemical Markup Language

```
<molecule>
<weight>234.5</weight>
<Spectra>…</Spectra>
<Figures>…</Figures>
</molecule>
```

- XML design goals:
  - XML should be compatible with SGML
  - It should be easy to write XML processors
  - The design should be formal and precise

XML – Structure

- XML: Confluence of SGML and HTML
- Xml looks like HTML
- Xml is a hierarchy of user-defined tags called elements with attributes and data
- Data is described by elements, elements are described by attributes

<BOOK genre="Science" format="Hardcover">…</BOOK>

Open tag
Element name

attribute

Attribute value

data

Closing Tag

XML – Elements

- Xml is case and space sensitive
- Element opening and closing tag names must be identical
- Opening tags: “<” + element name + “>”
- Closing tags: “</” + element name + “>”
- Empty Elements have no data and no closing tag:
  - They begin with a “<“ and end with a “/>”
- <BOOK/>

XML – Attributes

- Attributes provide additional information for element tags.

- There can be zero or more attributes in every element; each one has the form:

  attribute_name='attribute_value'

  - There is no space between the name and the "='"
  - Attribute values must be surrounded by " or ' characters

- Multiple attributes are separated by white space (one or more spaces or tabs).

XML – Data and Comments

- Xml data is any information between an opening and closing tag
- Xml data must not contain the ‘<‘ or ‘>’ characters
- Comments:

  <!- comment ->

XML – Nesting & Hierarchy

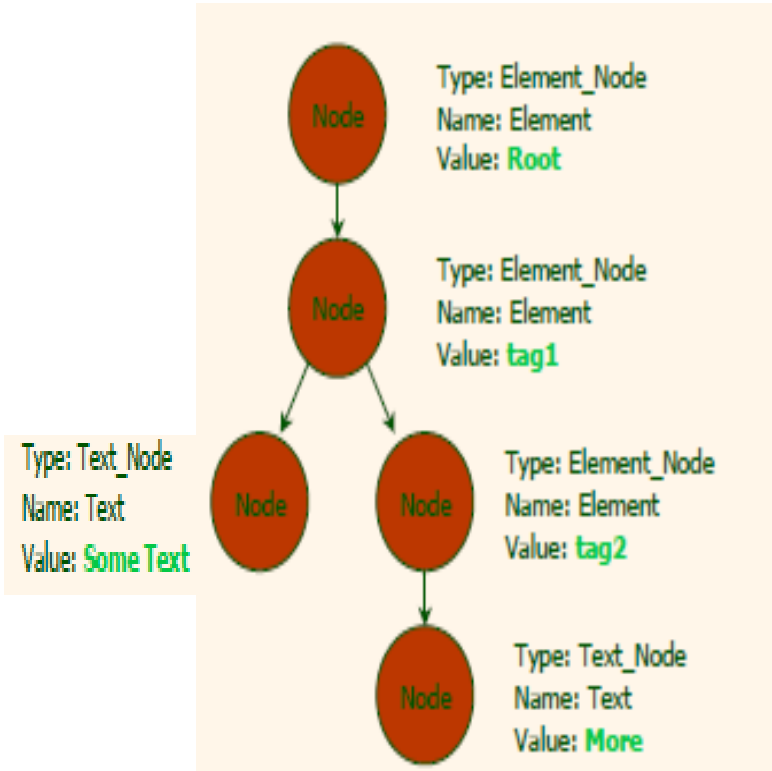
- Xml tags can be nested in a tree hierarchy
- Xml documents can have only one root tag
- Between an opening and closing tag you can insert:
  - Data
  - More Elements
  - A combination of data and elements

```
<root>
   <tag1>
            Some Text
            <tag2>More</tag2>
   </tag1>
</root>
```

Xml – Storage

- Storage is done just like an n-ary tree (DOM)

```
<root>
        <tag1>
                Some Text
                <tag2>More</tag2>
        </tag1>
</root>
```



DTD – Document Type Definition

- A DTD is a schema for Xml data
- Xml protocols and languages can be standardized with DTD files
- A DTD says what elements and attributes are required or optional
  - Defines the formal structure of the language

DTD – An Example

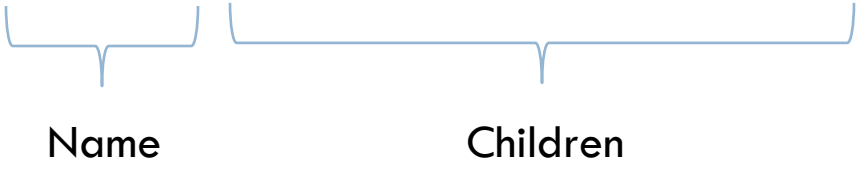

```
<?xml version='1.0'?>
<!ELEMENT Basket (Cherry+, (Apple | Orange)*) >
<!ELEMENT Cherry EMPTY>
<!ATTLIST Cherry flavor CDATA #REQUIRED>
<!ELEMENT Apple EMPTY>
<!ATTLIST Apple color CDATA #REQUIRED>
<!ELEMENT Orange EMPTY>
<!ATTLIST Orange location 'Florida'>
```

```
<Basket>
<Cherry flavor='good'/>
<Apple color='red'/>
<Apple color='green'/>
</Basket>
```

```
<Basket>
<Apple/>
<Cherry flavor='good'/>
<Orange/>
</Basket>
```

DTD - !ELEMENT

- <!ELEMENT Basket (Cherry+, (Apple | Orange)*) >



- !ELEMENT declares an element name, and what children elements it should have

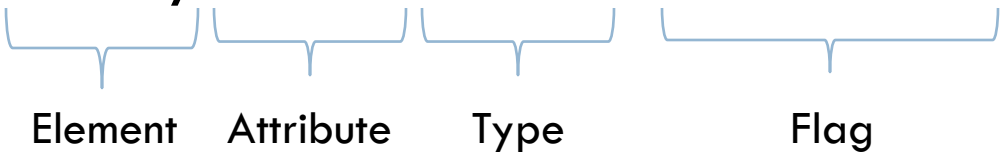
- Content types:
  - Other elements
  - #PCDATA (parsed character data)
  - EMPTY (no content)
  - ANY (no checking inside this structure)
  - A regular expression

DTD - !ELEMENT (Contd.)

- A regular expression has the following structure:
  - exp_1, exp_2, exp_3, …, exp_k: A list of regular expressions
  - exp^*: An optional expression with zero or more occurrences
  - exp^+: An optional expression with one or more occurrences
  - exp_1 | exp_2 | …| exp_k: A disjunction of expressions

DTD - !ATTLIST

<!ATTLIST Cherry flavor CDATA #REQUIRED>



<!ATTLIST Orange location CDATA #REQUIRED
color 'orange'>

- !ATTLIST defines a list of attributes for an Element
- Attributes can be of different types, can be required or not required, and they can have default values.

DTD – Well-Formed and Valid


```
<?xml version='1.0'?>
<!ELEMENT Basket (Cherry+)>
<!ELEMENT Cherry EMPTY>
<!ATTLIST Cherry flavor CDATA #REQUIRED>
```

Not Well – Formed

```
<basket>
<Cherry flavor=good>
</Basket>
```

Well – Formed but Invalid

```
<Job>
<Location>Home</Location>
</Job>
```

Well – Formed and Valid

```
<Basket>
<Cherry flavor='good'/>
</Basket>
```

XML and DTDs

- More and more standardized DTDs will be developed
  - MathML
  - Chemical Markup Language
- Allows light-weight exchange of data with the same semantics
- Sophisticated query languages for XML are available:
  - Xquery
  - XPath

Components of Data-Intensive Systems

- Three separate types of functionality:
  - Data management
  - Application logic
  - Presentation

- The system architecture determines whether these three components reside on a single system (“tier) or are distributed across several tiers

Single-Tier Architectures

- All functionality combined into a single tier, usually on a mainframe
  - User access through dumb Terminals
- Advantages:
  - Easy maintenance and administration
- Disadvantages:
  - Today, users expect graphical user interfaces.
  - Centralized computation of all of them is too much for a central system

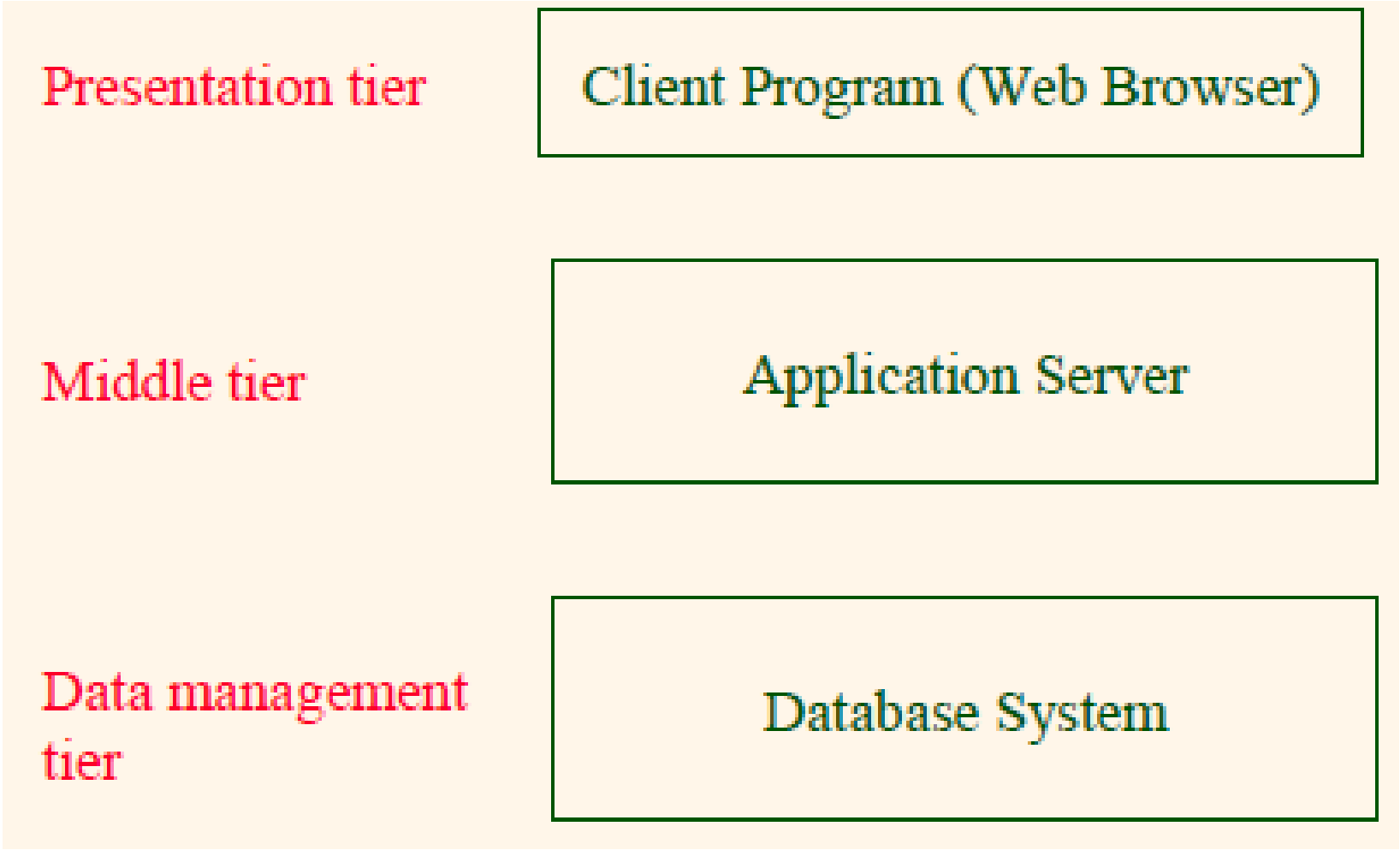
Client-Server Architectures

- Work division: Thin client
  - Client implements only the graphical user interface
  - Server implements business logic and data management
- Work division: Thick client
  - Client implements both the graphical user interface and the business logic
  - Server implements data management

Client-Server Architectures (Contd.)

- Disadvantages of thick clients
  - No central place to update the business logic
  - Security issues: Server needs to trust clients
    - Access control and authentication needs to be managed at the server
    - Clients need to leave server database in consistent state
    - One possibility: Encapsulate all database access into stored procedures
  - Does not scale to more than several 100s of clients
    - Large data transfer between server and client
    - More than one server creates a problem: x clients, y servers: x*y connections

The Three-Tier Architecture



The Three Layers

- Presentation tier
  - Primary interface to the user
  - Needs to adapt to different display devices (PC, PDA, cell phone, voice access?)
- Middle tier
  - Implements business logic (implements complex actions, maintains state between different steps of a workflow)
  - Accesses different data management systems
- Data management tier
  - One or more standard database management systems

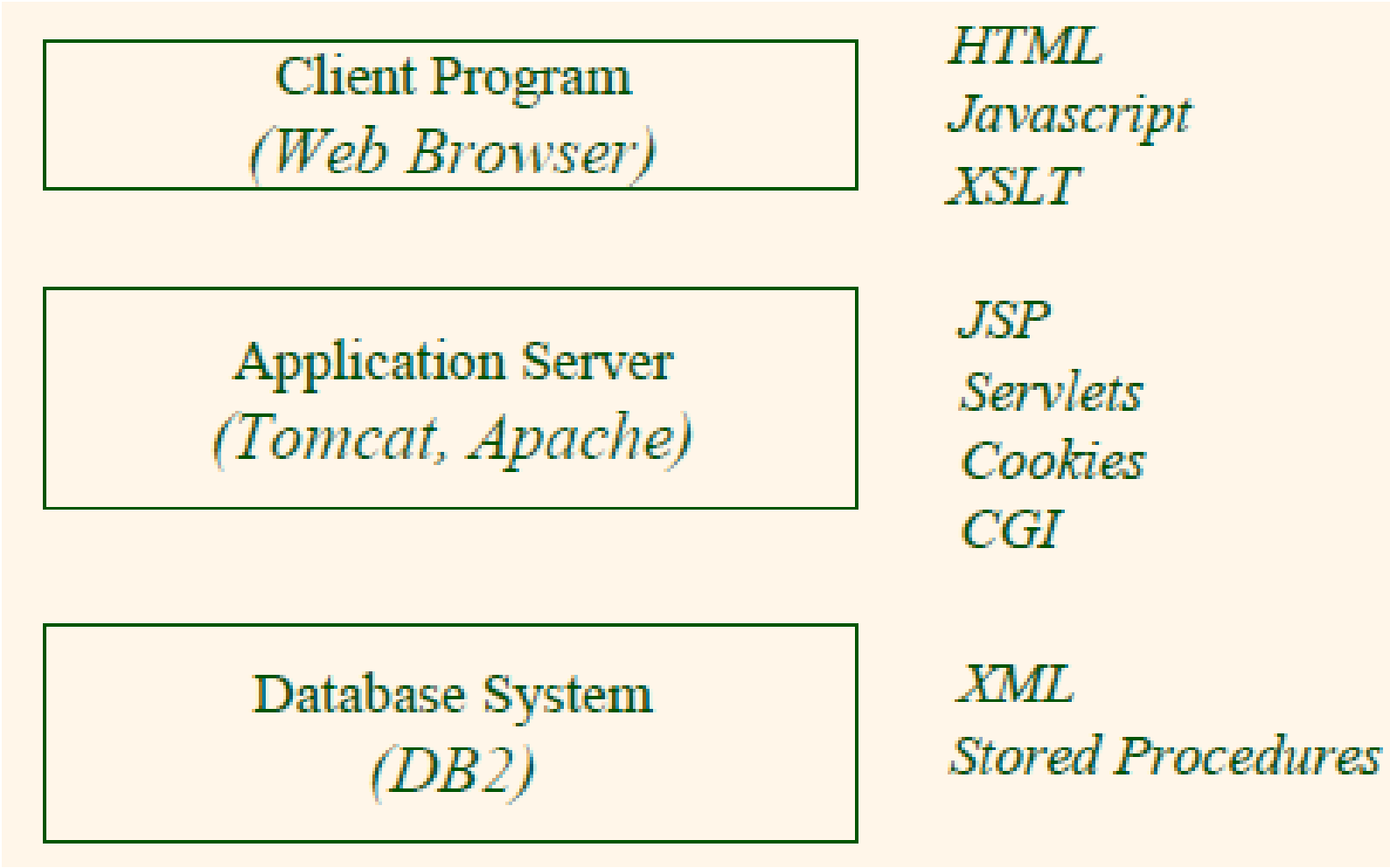
Example 1: Airline reservations

- Build a system for making airline reservations
- What is done in the different tiers?
- Database System
  - Airline info, available seats, customer info, etc.
- Application Server
  - Logic to make reservations, cancel reservations, add new airlines, etc.
- Client Program
  - Log in different users, display forms and human readable output

Example 2: Course Enrollment

- Build a system using which students can enroll in courses
- Database System
  - Student info, course info, instructor info, course availability, pre-requisites, etc.
- Application Server
  - Logic to add a course, drop a course, create a new course, etc.
- Client Program
  - Log in different users (students, staff, faculty), display forms and human-readable output

Technologies



Advantages of the Three-Tier Architecture

- Heterogeneous systems
  - Tiers can be independently maintained, modified, and replaced
- Thin clients
  - Only presentation layer at clients (web browsers)
- Integrated data access
  - Several database systems can be handled transparently at the middle tier
  - Central management of connections
- Scalability
  - Replication at middle tier permits scalability of business logic
- Software development
  - Code for business logic is centralized
  - Interaction between tiers through well-defined APIs: Can reuse standard components at each tier

Overview of the Presentation Tier

- Recall: Functionality of the presentation tier
  - Primary interface to the user
  - Needs to adapt to different display devices (PC, PDA, cell phone, voice access?)
  - Simple functionality, such as field validity checking

- We will cover:
  - HTML Forms: How to pass data to the middle tier
  - JavaScript: Simple functionality at the presentation tier
  - Style sheets: Separating data from formatting

HTML Forms

- Common way to communicate data from client to middle tier

- General format of a form:

```
<FORM ACTION="page.jsp" METHOD="GET" NAME="LoginForm">
…
</FORM>
```

- Components of an HTML FORM tag:
  - ACTION: Specifies URI that handles the content
  - METHOD: Specifies HTTP GET or POST method
  - NAME: Name of the form; can be used in client-side scripts to refer to the form

Inside HTML Forms

- INPUT tag
  - Attributes:
    - TYPE: text (text input field), password (text input field where input is, reset (resets all input fields)
    - NAME: symbolic name, used to identify field value at the middle tier
    - VALUE: default value
  - Example: <INPUT TYPE="text" Name="title">

- Example form:

```
<form method="POST" action="TableOfContents.jsp">
<input type="text" name="userid">
<input type="password" name="password">
<input type="submit" value="Login" name="submit">
<input type="reset" value="Clear">
</form>
```

Passing Arguments

- Two methods: GET and POST
  - GET
    - Form contents go into the submitted URI
    - Structure:
      - action?name1=value1&name2=value2&name3=value3
        - Action: name of the URI specified in the form
        - (name,value)-pairs come from INPUT fields in the form; empty fields have empty values ("name=")
    - Example from previous password form:
      - TableOfContents.jsp?userid=john&password=johnpw
    - Note that the page named action needs to be a program, script, or page that will process the user input

HTTP GET: Encoding Form Fields

- Form fields can contain general ASCII characters that cannot appear in an URI

- A special encoding convention converts such field values into "URI-compatible" characters:
  - Convert all "special" characters to %xyz, were xyz is the ASCII code of the character. Special characters include &, =, +, %, etc.
  - Convert all spaces to the "+" character
  - Glue (name,value)-pairs from the form INPUT tags together with "&" to form the URI

HTML Forms: A Complete Example


```
<form method="POST" action="TableOfContents.jsp">
<table align = "center" border="0" width="300">
<tr>
<td>Userid</td>
<td><input type="text" name="userid" size="20"></td>
</tr>
<tr>
<td>Password</td>
<td><input type="password" name="password" size="20"></td>
</tr>
<tr>
<td align = "center"><input type="submit" value="Login“
name="submit"></td>
</tr>
</table>
</form>
```

JavaScript

- Goal: Add functionality to the presentation tier.
- Sample applications:
  - Detect browser type and load browser-specific page
  - Form validation: Validate form input fields
  - Browser control: Open new windows, close existing windows (example: pop-up ads)
- Usually embedded directly inside the HTML with the <SCRIPT>… </SCRIPT> tag.
- <SCRIPT> tag has several attributes:
  - LANGUAGE: specifies language of the script (such as javascript)
  - SRC: external file with script code
  - Example:
    - <SCRIPT LANGUAGE="JavaScript" SRC="validate.js>
    - </SCRIPT>

JavaScript (Contd.)

- If <SCRIPT> tag does not have a SRC attribute, then the JavaScript is directly in the HTML file.

- Example:

```
<SCRIPT LANGUAGE="JavaScript">
<!-- alert("Welcome to our bookstore")
//-->
</SCRIPT>
```

- Two different commenting styles
  - <!-- comment for HTML, since the following JavaScript code should be ignored by the HTML processor
  - // comment for JavaScript in order to end the HTML comment

JavaScript (Contd.)

- JavaScript is a complete scripting language
  - Variables
  - Assignments (=, +=, …)
  - Comparison operators (<,>,…), boolean operators (&&, ||, !)
  - Statements
    - if (condition) {statements;} else {statements;}
    - for loops, do-while loops, and while-loops
  - Functions with return values
    - Create functions using the function keyword
    - f(arg1, …, argk) {statements;}

JavaScript: A Complete Example

HTML Form:

```
<form method="POST“
action="TableOfContents.jsp">
<input type="text"
name="userid">
<input type="password"
name="password">
<input type="submit"
value="Login“
name="submit">
<input type=“reset”
value=“Clear”>
</form>
```

Associated JavaScript:

```
<script language="javascript">
function testLoginEmpty()
{
loginForm = document.LoginForm
if ((loginForm.userid.value == "") ||
(loginForm.password.value == ""))
{
alert('Please enter values for userid and
password.');
return false;
}
else return true;
}
</script>
```

Stylesheets

- Idea: Separate display from contents, and adapt display to different presentation formats

- Two aspects:
  - Document transformations to decide what parts of the document to display in what order
  - Document rending to decide how each part of the document is displayed

- Why use stylesheets?
  - Reuse of the same document for different displays
  - Tailor display to user's preferences
  - Reuse of the same document in different contexts

- Two stylesheet languages
  - Cascading style sheets (CSS): For HTML documents
  - Extensible stylesheet language (XSL): For XML documents

CSS: Cascading Style Sheets

- Defines how to display HTML documents

- Many HTML documents can refer to the same CSS
  - Can change format of a website by changing a single style sheet
  - Example:

```
<LINK REL="style sheet" TYPE="text/css" HREF="books.css"/>
```

- Each line consists of three parts:
  - selector {property: value}
    - Selector: Tag whose format is defined
    - Property: Tag's attribute whose value is set
    - Value: value of the attribute

CSS: Cascading Style Sheets

Example style sheet:

```
body {background-color: yellow}
h1 {font-size: 36pt}
h3 {color: blue}
p {margin-left: 50px; color: red}
```

The first line has the same effect as:

```
<body background-color="yellow>
```

XSL

- Language for expressing style sheets

- Three components
  - XSLT: XSL Transformation language
    - Can transform one document to another
  - XPath: XML Path Language
    - Selects parts of an XML document
  - XSL Formatting Objects
    - Formats the output of an XSL transformation

Overview of the Middle Tier

- Recall: Functionality of the middle tier
  - Encodes business logic
  - Connects to database system(s)
  - Accepts form input from the presentation tier
  - Generates output for the presentation tier

- We will cover
  - CGI: Protocol for passing arguments to programs running at the middle tier
  - Application servers: Runtime environment at the middle tier
  - Servlets: Java programs at the middle tier
  - JavaServerPages: Java scripts at the middle tier
  - Maintaining state: How to maintain state at the middle tier. Main focus: Cookies.

CGI: Common Gateway Interface

- Goal: Transmit arguments from HTML forms to application programs running at the middle tier
- Details of the actual CGI protocol unimportant à libraries implement high-level interfaces
- Disadvantages:
  - The application program is invoked in a new process at every invocation (remedy: FastCGI)
  - No resource sharing between application programs (e.g., database connections)
  - Remedy: Application servers

CGI: Example

HTML form:

```
<form action="findbooks.cgi"
method=POST>
Type an author name:
<input type="text" name="authorName">
<input type="submit" value="Send it">
<input type="reset" value="Clear form">
</form>
```

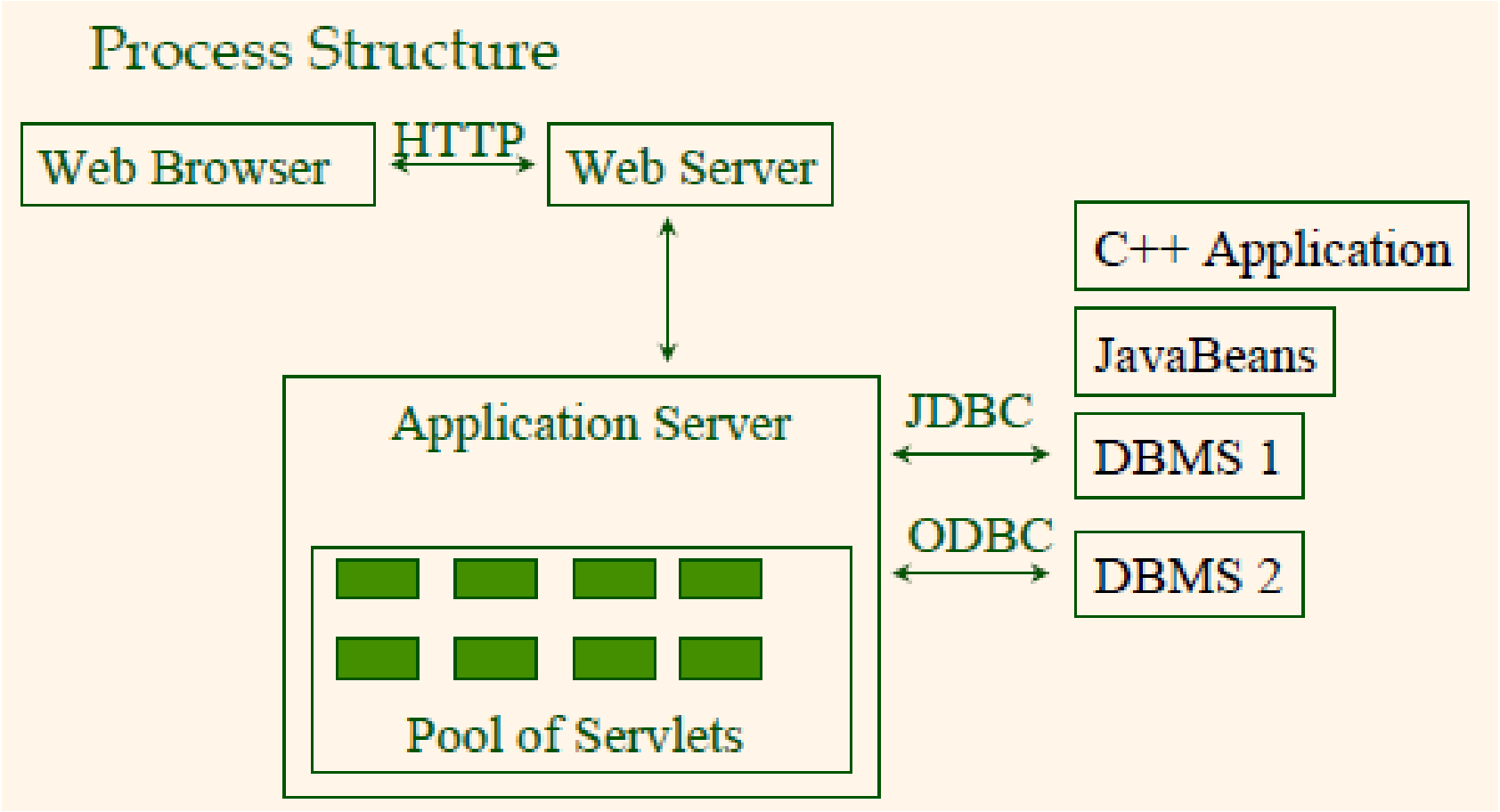
Perl code:

```
use CGI;
$dataIn=new CGI;
$dataIn->header();
$authorName=$dataIn->param('authorName');
print("<HTML><TITLE>Argument passing
test</TITLE>");
print("The author name is " + $authorName);
print("</HTML>");
exit;
```

Application Servers

- Idea: Avoid the overhead of CGI
  - Main pool of threads of processes
  - Manage connections
  - Enable access to heterogeneous data sources
  - Other functionality such as APIs for session management

Application Server: Process Structure



Servlets

- Java Servlets: Java code that runs on the middle tier
  - Platform independent
  - Complete Java API available, including JDBC

- Example:

```
import java.io.*;
import java.servlet.*;
import java.servlet.http.*;
public class ServetTemplate extends HttpServlet {
public void doGet(HTTPServletRequest request,
HTTPServletResponse response)
throws SerletExpection, IOException {
PrintWriter out=response.getWriter();
out.println("Hello World");
}
}
```

Servlets (Contd.)

- Life of a servlet?
  - Webserver forwards request to servlet container
  - Container creates servlet instance (calls init() method; deallocation time: calls destroy() method)
  - Container calls service() method
    - service() calls doGet() for HTTP GET or doPost() for HTTP POST
    - Usually, don't override service(), but override doGet() and doPost()

Servlets: A Complete Example


```
public class ReadUserName extends HttpServlet {
public void doGet( HttpServletRequest request,
HttpSevletResponse response)
throws ServletException, IOException {
reponse.setContentType("text/html");
PrintWriter out=response.getWriter();
out.println("<HTML><BODY>\n <UL> \n" + "<LI>" + request.getParameter("userid") +
"\n" + "<LI>" + request.getParameter("password") + "\n" +
"<UL>\n<BODY></HTML>");
}
public void doPost( HttpServletRequest request,
HttpSevletResponse response)
throws ServletException, IOException {
doGet(request,response);
}
}
```

Java Server Pages

- Servlets
  - Generate HTML by writing it to the “PrintWriter” object
  - Code first, webpage second

- JavaServerPages
  - Written in HTML, Servlet-like code embedded in the HTML
  - Webpage first, code second
  - They are usually compiled into a Servlet

JavaServerPages: Example


```
<html>
<head><title>Welcome to B&N</title></head>
<body>
<h1>Welcome back!</h1>
<% String name="NewUser";
if (request.getParameter("username") != null) {
name=request.getParameter("username");
}
%>
You are logged on as user <%=name%>
<p>
</body>
</html>
```

Maintaining State

- HTTP is stateless.

- Advantages
  - Easy to use: don't need anything
  - Great for static-information applications
  - Requires no extra memory space

- Disadvantages
  - No record of previous requests means
    - No shopping baskets
    - No user logins
    - No custom or dynamic content
    - Security is more difficult to implement

Application State

- Server-side state
  - Information is stored in a database, or in the application layer's local memory
- Client-side state
  - Information is stored on the client's computer in the form of a cookie
- Hidden state
  - Information is hidden within dynamically created web pages

Server-Side State

- Many types of Server side state:

- Store information in a database
  - Data will be safe in the database
  - BUT: requires a database access to query or update the information

- Use application layer's local memory
  - Can map the user's IP address to some state
  - BUT: this information is volatile and takes up lots of server main memory

5 million IPs = 20 MB

Server-Side State

- Should use Server-side state maintenance for information that needs to persist
  - Old customer orders
  - "Click trails" of a user's movement through a site
  - Permanent choices a user makes

Client-side State: Cookies

- Storing text on the client which will be passed to the application with every HTTP request.
  - Can be disabled by the client.
  - Are wrongfully perceived as "dangerous", and therefore will scare away potential site visitors if asked to enable cookies
- Are a collection of (Name, Value) pairs

Client State: Cookies

- Advantages
  - Easy to use in Java Servlets / JSP
  - Provide a simple way to persist non-essential data on the client even when the browser has closed
- Disadvantages
  - Limit of 4 kilobytes of information
  - Users can (and often will) disable them
- Should use cookies to store interactive state
  - The current user's login information
  - The current shopping basket
  - Any non-permanent choices the user has made

Creating A Cookie


```
Cookie myCookie = new Cookie(“username", “jeffd");
response.addCookie(userCookie);
```

- You can create a cookie at any time

Accessing A Cookie


```
Cookie[] cookies = request.getCookies();
String theUser;
for(int i=0; i<cookies.length; i++)
{
   Cookie cookie = cookies[i];
   if(cookie.getName().equals("username"))
                theUser = cookie.getValue();
}
```

// at this point theUser == "username"

- Cookies need to be accessed BEFORE you set your response header:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
```

Cookie Features

- Cookies can have
  - A duration (expire right away or persist even after the browser has closed)
  - Filters for which domains/directory paths the cookie is sent to

Hidden State

- Often users will disable cookies
- You can "hide" data in two places:
  - Hidden fields within a form
  - Using the path information
- Requires no "storage" of information because the state information is passed inside of each web page

Hidden State: Hidden Fields

- Declare hidden fields within a form:

<input type='hidden' name='user' value='username'/>

- Users will not see this information (unless they view the HTML source)

- If used prolifically, it's a killer for performance since EVERY page must be contained within a form.

Hidden State: Path Information

- Path information is stored in the URL request:

  http://server.com/index.htm?user=jeffd

- Can separate 'fields' with an & character:

  index.htm?user=jeffd&preference=pepsi

- There are mechanisms to parse this field in Java. Check out the

  javax.servlet.http.HttpUtils parserQueryString() method.

Multiple state methods

- Typically all methods of state maintenance are used:
  - User logs in and this information is stored in a cookie
  - User issues a query which is stored in the path information
  - User places an item in a shopping basket cookie
  - User purchases items and credit-card information is stored/retrieved from a database
  - User leaves a click-stream which is kept in a log on the web server (which can later be analyzed)

Questions

1. Define view. Explain the problems related to updating the view.
2. What is trigger? Explain with an example.
3. What are the various methods of accessing the databases? Explain.
4. Differentiate between Embedded SQL and SQLJ.
5. What are the different statement objects? Explain.
6. Explain three-tier application architecture.