Database Technology
Database Applications

Heiko Paulheim
Recap: The Big Picture

• Users interact with databases *indirectly*
  – i.e., via *applications*
  – no direct usage of SQL

• Most applications today have a database under the hood, e.g.,
  – shopping portals
  – news web sites
  – games
Today’s Lecture

• Architectures for database centric applications
  – Three/Two-Layer Web Architecture
  – HTML/Session/Cookies
  – Server/Client Side Scripting

• Legacy Systems

• Performance Tuning
  – Bottlenecks
  – Database Design

• Security Issues
  – SQL Injection
  – Cross Site Scripting
  – Password Leakage
  – Application Authentication/Authorization
Application Architecture Evolution

- Three eras of application architecture
  - mainframe (1960’s and 70’s)
  - personal computer era (1980’s)
  - Web era (since 1990’s, nowadays mostly mobile Web)
Web Interface

• Web browsers
  – de-facto standard user interface to databases
  – multi-user, location agnostic interface
  – no need for downloading/installing specialized code, while providing a good graphical user interface
    • JavaScript, Flash and other scripting languages run in browser, but are downloaded transparently
  – Examples: banks, airline and rental car reservations, university course registration and grading, ...
Web based Applications in a Nutshell

- Web documents are *hypertext* documents formatted using HyperText Markup Language (HTML)
- HTML documents contain
  - text along with font specifications, and other formatting instructions
  - hypertext links to other documents
  - *forms*, enabling users to enter data which can then be sent back to the Web server
- HyperText Transfer Protocol (HTTP) used for communication with the Web server
- URL may identify a document or an executable program
  - executed by HTTP server
  - creates HTML documents, which is sent back to client
  - Web client can pass extra arguments with the name of the document
Sample HTML Source Text

<html>
<body>
<table border>
<tr> <th>ID</th> <th>Name</th> <th>Department</th> </tr>
<tr> <td>00128</td> <td>Zhang</td> <td>Comp. Sci.</td> </tr>
...
</table>
<form action="PersonQuery" method=get>
Search for:
<select name="persontype">
<option value="student" selected>Student </option>
<option value="instructor"> Instructor </option>
</select> <br>
Name: <input type=text size=20 name="name">
<input type=submit value="submit">
</form>
</body> </html>
### Display of Sample HTML Source

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>00128</td>
<td>Zhang</td>
<td>Comp. Sci.</td>
</tr>
<tr>
<td>12345</td>
<td>Shankar</td>
<td>Comp. Sci.</td>
</tr>
<tr>
<td>19991</td>
<td>Brandt</td>
<td>History</td>
</tr>
</tbody>
</table>

Search for: **Student**

Name: 

submit
Three-Layer Web Architecture
Two-Layer Web Architecture

- Multiple levels of indirection have overheads
  - Alternative: two-layer architecture
HTTP and Sessions

• The HTTP protocol is **connectionless**
  – Once the server replies to a request, the server closes the connection with the client, and forgets all about the request
  – In contrast, Unix logins, and JDBC/ODBC connections stay connected until the client disconnects
    • retaining user authentication and other information
  – Motivation: reduce load on server
    • operating systems have tight limits on number of open connections on a machine

• Information services need session information
  – E.g., user authentication should be done only once per session

• Solution: **cookies**
Sessions and Cookies

• A **cookie** is a small piece of text containing identifying information
  – Sent by server to browser
    • Sent on first interaction, to identify session
  – Sent by browser with each request
    • part of the HTTP protocol
  – Server saves information about cookies it issued, and can use it when serving a request
    • E.g., authentication information, and user preferences
• Cookies can be stored permanently or for a limited time
Programming on the Server Side

• Paradigms
  – Programming (i.e., each document is assembled by a program)
  – Scripting (embedded in HTML)

• Different programming languages can be used, e.g.
  – PHP
  – ASP.NET
  – Java (JSP, Servlets)
  – ColdFusion
  – Perl
  – Python
  – ...

https://w3techs.com/technologies/comparison/pl-aspnet,pl-php
Servlets

• Java Servlet specification
  – defines an API for communication between the Web/application server and application program running in the server
    • methods to get parameter values from Web forms
    • methods to send HTML text back to client
• Application program (also called a servlet) is loaded into the server
  – Each request spawns a new thread in the server
    • thread is closed once the request is serviced
Example Servlet Code

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class PersonQueryServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HEAD><TITLE> Query Result</TITLE></HEAD>");
        out.println("<BODY>");
        // .... BODY OF SERVLET (next slide) ...
        out.println("</BODY>");
        out.close();
    }
}
```
Example Servlet Code

```java
String persontype = request.getParameter("persontype");
String number = request.getParameter("name");
if(persontype.equals("student")) {
    ... code to find students with the specified name ...
    ... using JDBC to communicate with the database ...
    out.println("<table BORDER COLS=3>");
    out.println(" <tr> <td>ID</td> <td>Name: </td> <td>Department</td> </tr>");
    for(... each result ...){
        ... retrieve ID, name and dept name
        ... into variables ID, name and deptname
        out.println("<tr> <td>" + ID + "</td> <td>" + name + "</td> + "<td>" + deptname + "</td></tr>");
    }
    out.println("</table>");
}
else {
    ... as above, but for instructors ...
}
```
Servlet Sessions

- Servlet API supports handling of sessions
  - Set a cookie on first interaction with browser
  - use it to identify session on further interactions
- To check if session is already active:
  - if (request.getSession(false) == true)
    - .. then use existing session
    - else .. redirect to authentication page
  - authentication page
    - check login/password

we can also check the age of the cookie here for session timeout
Servlet Support

- Servlets run inside application servers such as
  - Apache Tomcat, Glassfish, JBoss
  - BEA Weblogic, IBM WebSphere and Oracle Application Servers
- Application servers support
  - deployment and monitoring of servlets
  - Java 2 Enterprise Edition (J2EE) platform supporting objects, parallel processing across multiple application servers, etc
Server-Side Scripting

• Server-side scripting
  – HTML document with embedded executable code and/or SQL queries
  – Input values from HTML forms can be used directly
  – When the document is requested, the Web server executes the embedded code/SQL queries to generate the actual HTML document

• Numerous server-side scripting languages
  – JSP, PHP
  – General purpose scripting languages: VBScript, Perl, Python
Java Server Pages (JSP)

- A JSP page with embedded Java code
  ```html
  <html>
  <head> <title> Hello </title> </head>
  <body>
  <% if (request.getParameter("name") == null) {
    out.println("Hello World");
  } else { out.println("Hello, " + request.getParameter("name")); } %>
  </body>
  </html>
  ```

- JSP is compiled into Java + Servlets
- JSP allows new tags to be defined, in tag libraries
  - such tags are like library functions, can be used for example to build rich user interfaces such as paginated display of large datasets
PHP

- PHP is widely used for Web server scripting
- Extensive libraries including for database access using ODBC

```html
<html>
<head>
<title>Hello</title>
</head>
<body>
<?php if (!isset($_REQUEST['name']))
{
    echo "Hello World";
}
else {
    echo "Hello, " + $_REQUEST['name'];
}
?>
</body>
</html>
```
Client Side Scripting

• Browsers can fetch certain scripts (client-side scripts) or programs along with documents, and execute them in “safe mode” at the client site
  – Javascript
  – Macromedia Flash and Shockwave for animation/games
  – VRML
  – Applets

• Client-side scripts/programs allow documents to be active
  – E.g., animation by executing programs at the local site
  – E.g., ensure that values entered by users satisfy some correctness checks
  – Permit flexible interaction with the user
  • Executing programs at the client site speeds up interaction by avoiding many round trips to server

https://de.wikipedia.org/wiki/Virtual_Reality_Modeling_Language
Client Side Scripting and Security

• Security mechanisms needed to ensure that malicious scripts do not cause damage to the client machine
  – easy for limited capability scripting languages
  – harder for general purpose programming languages like Java
• E.g., Java’s security system ensures that the Java applet code does not make any system calls directly
  – Disallows dangerous actions such as file writes
  – Notifies the user about potentially dangerous actions
    • allow the option to abort the program or to continue execution
Javascript

• Javascript very widely used
  – forms basis of new generation of Web applications (called Web 2.0 applications) offering rich user interfaces

• Javascript functions can
  – check input for validity
  – modify the displayed Web page
    • by altering the underlying **document object model (DOM)** tree
  – communicate with a Web server to fetch data and modify the current page using fetched data, without needing to reload/refresh the page
    • forms basis of AJAX technology used widely in Web 2.0 applications
    • e.g., loading further content upon scrolling down a Web page
    • e.g. on selecting a country in a drop-down menu, the list of states in that country is automatically populated in a linked drop-down menu
Legacy Systems

• Older-generation systems that are incompatible with current generation standards and systems but still in production use
  – E.g., applications written in COBOL that run on mainframes
    • Today’s hot new system is tomorrow’s legacy system!
• Porting legacy system applications to a more modern environment is problematic
  – Legacy system may involve millions of LoC, written over decades
    • Original programmers usually no longer available
  – Switching over from old system to new system is a problem
    • more on this later
• One approach: build a **wrapper** layer on top of legacy application to allow interoperation between newer systems and legacy application
  – E.g., use ODBC or OLE-DB as wrapper
Legacy Systems (Cont.)

• Rewriting legacy application: understanding what it does (and how)
  – Legacy code often has no/little documentation
  – **reverse engineering**: process of going over legacy code to
    • Come up with schema designs in ER or OO model
    • Get a high level view of system
• **Re-engineering**: reverse engineering followed by design of new system
  – Improvements are made on existing system design in this process
Legacy Systems (Cont.)

• Switching over from old to new system is a major problem
  – Production systems are in every day, generating new data
  – Stopping the system may bring all of a company’s activities to a halt, causing enormous losses

• Big-bang approach:
  – Implement complete new system
  – Populate it with data from old system
    • No transactions while this step is executed
    • scripts are created to do this quickly
  – Shut down old system and start using new system
  – Danger with this approach: what if new code has bugs or performance problems, or missing features
    • Company may be brought to a halt
Legacy Systems (Cont.)

- **Chicken-little approach:**
  - Replace legacy system one piece at a time
  - Use wrappers to interoperate between legacy and new code
    - E.g., replace front end first, with wrappers on legacy backend
      - Old front end can continue working in this phase in case of problems with new front end
    - Replace back end, one functional unit at a time
      - All parts that share a database may have to be replaced together, or wrapper is needed on database as well
  - Drawback: significant extra development effort to build wrappers and ensure smooth interoperation
    - Still worth it if company’s life depends on system
Performance Tuning

• Adjusting various parameters and design choices
  – to improve system performance for a specific application
  – notion: continuous improvement rather than waterfall model

• Tuning is best done by
  1) identifying bottlenecks, and
  2) eliminating them

• Three levels of tuning
  – Hardware, e.g., add disks, memory, use faster processor
  – Database system parameters, e.g., buffer size, checkpointing intervals
  – Higher level database design, e.g., schema, indices, and transactions
Bottlenecks

• Performance of most systems (at least before they are tuned) usually limited by performance of one or a few components
  – these are called **bottlenecks**
  – 80/20 rule: 20% of code consume 80% of execution time
    • spend more time on those 20%
• Bottlenecks may be in hardware (e.g., disks are very busy, CPU is idle), or in software
• Removing one bottleneck often exposes another
• De-bottlenecking consists of repeatedly finding bottlenecks, and removing them
Identifying Bottlenecks

• Transactions request a sequence of services
  – E.g., CPU, Disk I/O, locks
• Concurrent transactions wait for a requested service while others are being served
• Notion: database as a **queueing system** with a queue for each service
  – Transactions repeatedly do the following
    • request a service, wait in queue for the service, and get serviced
• Bottlenecks in a database system typically show up as very high utilizations (very long queues) of a particular service
  – e.g., disk vs. CPU utilization
  – 100% utilization leads to very long waiting time:
    • Rule of thumb: design system for about 70% utilization at peak load
    • utilization over 90% should be avoided
Queues in a Database System
Tuning of Hardware

• Even well-tuned transactions typically require a few I/O operations
  – Typical disk supports about 100 random I/O operations per second
  – Suppose each transaction requires just 2 random I/O operations
    • to support $n$ transactions per second, we need to distribute data across $n/50$ disks (ignoring skew)

• Number of I/O operations per transaction can be reduced by keeping more data in memory
  – If all data is in memory, I/O needed only for writes
  – Keeping frequently used data in memory reduces disk accesses, reducing number of disks required, but has a memory cost

• Five minute rule:
  – if a page that is randomly accessed is used more frequently than once in five minutes, it should be kept in memory
Tuning the Database Design

• Schema tuning
  – Vertically partition relations to isolate the data that is accessed most often
    • e.g., split account into two, (account-number, branch-name) and (account-number, balance).
        – branch name need not be fetched unless required
        – More rows per block → less block transfers
  – Improve performance by storing a denormalized relation
    • E.g., store join of account and depositor; branch-name and balance information is repeated for each holder of an account
        – join need not be computed repeatedly
        – trade-off: more space and more work for programmer to keep relation consistent on updates
        – Better to use materialized views (see later)
Tuning the Database Design (Cont.)

• Incidental violations of normal forms
  – e.g., storing join tables that would be split by normalization

• Incidental violations of domain model
  – Example: each person can have many phone numbers (1:n)
  – Theoretically sound solution: two tables (person, phone)
  – Practical observation: not more than four in 1M persons
    • rather introduce attributes phone1, phone2, phone3, phone4
    • avoids joins with long tables
Materialized Views

- Materialized views can help speed up certain queries
  - Particularly aggregate queries

- Overheads
  - Space
  - Time for view maintenance
    - Immediate view maintenance: done as part of update transaction
      - time overhead paid by update transaction
    - Deferred view maintenance: done only when required
      - update transaction is not affected, but system time is spent on view maintenance
      - until updated, the view may be out-of-date

- Preferable to denormalized schema since view maintenance is systems responsibility, not programmers
  - Avoids inconsistencies caused by errors in update programs
Tuning the Database Design (Cont.)

• How to choose set of materialized views
  – Helping one transaction type by introducing a materialized view may hurt others
    • selections including aggregates will be speed up
    • updates are slowed down
  – Choice of materialized views depends on costs
    • Users often have no idea of actual cost of operations
  – Overall, manual selection of materialized views is tedious

• Some database systems provide tools to help DBA choose views to materialize
  – “Materialized view selection wizards”
Tuning the Database Design (Cont.)

• **Index tuning**
  – Create appropriate indices to speed up slow queries/updates
  – Speed up slow updates by removing excess indices (tradeoff between queries and updates)
  – Choose type of index (B-tree/hash) appropriate for most frequent types of queries
  – Choose which index to make clustered

• Index tuning wizards look at past history of queries and updates (the **workload**) and recommend which indices would be best for the workload
Application Security

WARNING

DON’T TRY THIS AT HOME
SQL Injection

• In an application, users enter data
  – this is a possible entry point for hackers!

• Consider the following code:

```java
String user = request.getParameter("username");
String password = request.getParameter("password");
String query = "SELECT * FROM users
  WHERE username = " + user + ""
  AND password = " + password "";

// execute query

// if there is a result, the login attempt was successful
```
SQL Injection

• Good user:
  – username “John”, password “test123”

• Bad user:
  – username “Jack”, password “test123’ OR 1=1”

• Consider the following code:

```java
String user = request.getParameter("username");
String password = request.getParameter("password");
String query = "SELECT * FROM users
  WHERE username = " + user + ""
  AND password = " + password "";

// execute query

// if there is a result, the login attempt was successful
```
SQL Injection

- Variant 1: Manual
  - Check input for and mask/replace/remove special characters

- Variant 2: Using prepared statements in Java
  
  ```java
  PreparedStatement stmt = connection.prepareStatement
    ("SELECT * FROM users WHERE username=? AND password=?");
  stmt.setString(1, user);
  stmt.setString(2, password);
  ResultSet rs = stmt.executeQuery();
  ```

https://xkcd.com/327
Cross Site Scripting

- HTML code on one page executes action on another page
  - E.g. `<img src='http://mybank.com/transfermoney?amount=1000&toaccount=14523'>`
  - Risk: if user viewing page with above code is currently logged into mybank, the transfer may succeed
  - Above example simplistic, since GET method is normally not used for updates, but if the code were instead a script, it could execute POST methods

- Above vulnerability called cross-site scripting (XSS) or cross-site request forgery (XSRF or CSRF)

- Prevent your web site from being used to launch XSS or XSRF attacks
  - Disallow HTML tags in text input provided by users, using functions to detect and strip such tags

- Protect your web site from XSS/XSRF attacks launched from other sites
  - ..next slide
Cross Site Scripting

- Protect your web site from XSS/XSRF attacks launched from other sites
  - Use `referer` value (URL of page from where a link was clicked) provided by the HTTP protocol, to check that the link was followed from a valid page served from same site, not another site
  - Ensure IP of request is same as IP from where the user was authenticated
    - prevents hijacking of cookie by malicious user
  - Never use a GET method to perform any updates
    - This is actually recommended by HTTP standard
Password Leakage

• **Never** store passwords, such as database passwords, in clear text in scripts that may be accessible to users
  – E.g. in files in a directory accessible to a web server
  • Normally, web server will execute, but not provide source of script files such as file.jsp or file.php, but source of editor backup files such as file.jsp~, or .file.jsp.swp may be served

• Restrict access to database server from IPs of machines running application servers
  – Most databases allow restriction of access by source IP address
Password Leakage

• **Never** store user passwords as plain text in a database!
• Hackers may get access to the database and read them
  – e.g., username “Jack”, password “test123; SELECT * FROM users”

• Typical best practice: store password hashes, e.g., md5
  – hashing is fast in one direction, hard in the other
  – Query:
    • `SELECT * FROM users WHERE user=? and password=md5(?)`
  – Changing passwords
    • `UPDATE users SET password=md5(?) WHERE user=?`
  – This way, passwords are never stored in plain text anywhere
Password Leakage

- **Attacks for hashed passwords: dictionary and brute force attacks**
  
  **Dictionary Attack**
  
  Trying apple : failed
  Trying blueberry : failed
  Trying justinbieber : failed
  ...
  Trying letmein : failed
  Trying s3cr3t : success!

  **Brute Force Attack**
  
  Trying aaaa : failed
  Trying aaab : failed
  Trying aaac : failed
  ...
  Trying acdb : failed
  Trying acdc : success!

- **Lookup Tables**

  \[
  2cf24dba5fb0a30e26e8043362938b9824 = \text{hash}(“hello”) \\
  232bc6bd9ec38f616560b120fda8e90f383 = \text{hash}(“hbllo”) 
  \]

- **Adding Salt to the password (appending a random string)**
  
  - Lookup tables won’t work

- **Do not implement your own crypto algorithm (use e.g. phpass)**
Application Authentication

- Single factor authentication such as passwords too risky for critical applications
  - guessing of passwords, sniffing of packets if passwords are not encrypted
  - passwords reused by user across sites
  - spyware which captures password
- Two-factor authentication
  - e.g. password plus one-time password sent by SMS
  - e.g. password plus one-time password devices
    - device generates a new pseudo-random number every minute, and displays to user
    - user enters the current number as password
    - application server generates same sequence of pseudo-random numbers to check that the number is correct
Application Authentication

• **Man-in-the-middle** attack
  – E.g. web site that pretends to be mybank.com, and passes on requests from user to mybank.com, and passes results back to user
  – Even two-factor authentication cannot prevent such attacks

• Solution: authenticate Web site to user, using digital certificates, along with secure http protocol

• **Central authentication** within an organization
  – application redirects to central authentication service for authentication
  – avoids multiplicity of sites having access to user’s password
  – LDAP or Active Directory used for authentication
Single Sign-On

- **Single sign-on** allows user to be authenticated once, and applications can communicate with authentication service to verify user’s identity without repeatedly entering passwords.

- **Security Assertion Markup Language (SAML)** standard for exchanging authentication and authorization information across security domains.
  - e.g. user from Yale signs on to external application such as acm.org using userid `joe@yale.edu`
  - application communicates with Web-based authentication service at Yale to authenticate user, and find what the user is authorized to do by Yale (e.g. access certain journals).

- **OpenID** standard allows sharing of authentication across organizations.
  - e.g. application allows user to choose Yahoo! as OpenID authentication provider, and redirects user to Yahoo! for authentication.
Application-Level Authorization

- Current SQL standard does not allow fine-grained authorization such as “students can see their own grades, but not other’s grades”
  - Problem 1: Database has no idea who are application users
  - Problem 2: SQL authorization is at the level of tables, or columns of tables, but not to specific rows of a table

- One workaround: use views such as

```sql
create view studentTakes as
select *
from takes
where takes.ID = USER()
```

  - where USER() provides end user identity
    - end user identity must be provided to the database by the application
  - Having multiple such views is cumbersome
Audit Trails

- Applications must log actions to an audit trail, to detect who carried out an update, or accessed some sensitive data
- Audit trails used after-the-fact to
  - detect security breaches
  - repair damage caused by security breach
  - trace who carried out the breach
- Audit trails needed at
  - Database level, and at
  - Application level
Summary

• Databases do not run by themselves, but in context
  – applications work on top
• A good database design is essential, but there’s also
  – security
  – performance,
  – …
• There’s quite a few trade offs
  – storage vs. velocity
  – update vs. read time
  – …
  → there’s no once size fits all solution!
What’s Next?

• Database Systems II (FSS, Moerkotte)
  – e.g., distributed DBMS, object-relational DBs, deductive DBs
• Query Optimization (HWS, Moerkotte)
  – more sophisticated query optimization
• Transaction Systems (FSS, Moerkotte)
  – more on multi-user synchronization protocols and methods
• Large-Scale Data Management (HWS, Gemulla)
  – e.g., parallel & distributed databases, MapReduce, SPARQL, NoSQL
What’s Next?

• Data Security (FSS, Armknecht)
  – also covers encryption, privacy, etc.
• Web Data Integration (HWS, Bizer)
  – dealing with multiple databases
  – automatically integrating them into a single one
• Data Mining (FSS/HWS, Bizer/Paulheim)
  – finding patterns in data
  – entry point to more specific lectures in the data analytics field
Questions?