Web Applications Engineering: Performance and Scalability

Service Oriented Computing Group, CSE, UNSW

Week 10

Material in these Lecture Notes is derived from:

- Building Scalable and High-performance Java Web Applications by Greg Barish, Addison Wesley
- Building Scalable Web Sites by Cal Henderson, O'Reilly Books
Performance

- **Performance** - The time taken to complete a unit operation on your Web application
  - End-to-end/Round trip time - the time taken between a user presenting an input to getting a response from the application
  - Goal of any performance optimization is to minimise the round trip time by improving the performance of individual components (Web server, application logic, database)

- Performance is cumulative - depends on speed of server’s response, of the network, of database retrievals, etc.
Scalability

- **Scalability** - To keep performance constant even with increase in the number of users and requests
  - Measurement of resilience under ever-increasing load
  - Effectiveness of adding new resources to improve efficiency.
  - Highly scalable $\Rightarrow$ Requiring additional resources at low rate.
  - Corollary: If a web application is not scalable, then throwing more resources at it does not help much!

- **Scalability $\neq$ Performance** - A high-performing site need not be scalable though the reverse is generally true in practice.

- Scalability does not come from a language or platform - e.g. Java vs. PHP vs. Ruby vs. Python ..

- Scalability also does not come from an architecture - i.e. MVC or 3-tier is not inherently more scalable than others.
Case Study: Twitter

Repeated issues with scalability led to move from Ruby-on-Rails to Java.
- Implemented fully asynchronous result aggregation for search.
- Used workflows to handle dependencies between services. Workflow lifecycle is handled by a different process.

Types of Scaling

_Whatchew gonna do, when they come for you?_

**Vertical Scaling** - start with basic servers and replace them with more powerful machines when the demand increases. Rinse and repeat.

- **Good** - easy to design for, hardware upgrades maintain performance so software does not need to change.
- **Bad** - Price growth for hardware upgrades is exponential, gain in scale/performance may not match increased capacity, and there are limits to server performance.

**Horizontal Scaling** - start with basic servers and add new instances of the same when demand increases a.k.a the Google model.

- **Good** - Scalability on the cheap by using massive farms of commodity servers.
- **Bad** - Administration becomes a pain, servers fail every day, software design could hamper scalability.
Why does performance and scalability matter?

- **Wide Audience** - Millions of users, users are not only humans but also other web applications which regularly poll your application for data.

- **Interactive** - User queries drive the application logic on the server. Users also submit information via parameters that need to be processed.

- **Dynamic** - Data generated is dependent on user input/session generated. This means that pages cannot be generated in advance. Result of interactivity but not the same.

- **Always On** - Applications are always available on the Internet - there’s no shutdown or Ctrl-Al-Del. If a popular site goes down, it becomes news and impacts the perception of reliability.

- **Integrated** - Applications depend on other applications. If one goes down, a chain of applications may fail (e.g. credit-card processing applications).
Architectural Considerations - Network

- **Client-Side Network Elements**
  - Can be leveraged using HTTP features to improve client-side performance
    - *Browser Cache* - Static elements are stored in the browser cache so that client performance is increased
    - *Proxy Cache* - Implemented at the ISP end so that clients share frequently accessed pages

- **Server-Side Network Elements**
  - Determines the network environment in which your application will execute
    - *Load balancers* - Route requests among multiple machines (a.k.a Server farms) that host the same application
    - *Reverse proxy cache* - Stores items that are frequently accessed by most clients.

- **In-between**
  - *Content distribution network* - Provides strategic replication to host content close to clients.
Architectural Considerations - Server

**Tier** - A layer in your application through which the client request has to pass to be resolved. E.g. A 3-tier application consists of the interface (JSP), business logic (servlets) and database (RDBMS). Applications can have many tiers ($n$-tier).

**How many tiers should your application contain?**

**Benefits**

- Applications are modular and can re-use components - reduces development time and minimises errors (if designed right)
- Distributed processing - Components can run on multiple machines
- Overloaded components can be isolated for replication

**Disadvantages**

- Message passing among components brings overheads
- Increased complexity of development
- Synchronization of distributed components
General Techniques for Improving Performance and Scalability

1. Caching/Replication

A Web cache is a store made up of key-value pairs. The keys and values can be complex objects and not just strings. One of the key challenges of caching is to design policies for retention and flush. e.g. Least-Recently-Used (LRU).

If designed well, caching can provide faster access than the original source even if the cache is located farther than the original source (bandwidth reasons).

However, caching has its disadvantages:

1. Guaranteeing data consistency
2. What data should be cached - (usernames, passwords ?)
3. Who obeys caching rules ?

We’ll look into this further in a later section.
General Techniques for Improving Performance and Scalability

2. Parallelism

Most servers today run on multi-core chip architectures and can support multiple instances of even operating systems.

Processing requests and carrying out tasks in parallel allows an application to take advantage of these architectures - generally handled by the container.

However, your application must be designed to run in a multi-threaded environment (E.g. Thread-safe controllers as requests may be processed in multiple threads).

Long-running tasks (e.g. I/O bound tasks) must be despatched to other threads/applications - Reduce response time.
General Techniques for Improving Performance and Scalability

3. Redundancy

Duplication of hardware and software so that the more resources are available for executing the application.

Not the same as replication, which is copying the data.

For example, web server farms where the same service is duplicated on multiple machines.

Disadvantages:

- Requires a load balancer - extra component at extra cost
- Data consistency - several instances connecting to the same database
General Techniques for Improving Performance and Scalability

3. Asynchrony

Components send messages to each other but go on executing without waiting for responses.

Asynchronous communication avoids serialized execution wherein the requestor is idle while waiting for response.

E.g. A search engine needn’t wait for the entire web to be crawled before returning results.

However, consistency of shared data structures must be maintained.

Also, certain implementations can result in overheads for synchronisation, serialisation and marshalling.
General Techniques for Improving Performance and Scalability

4. Resource Pooling

Share access to resources in order to reduce overhead of connections.

E.g. Database connections are shared between applications in a single container.

Disadvantages:

- May hamper scalability through artificial scarcity
- Overhead of reclaiming instances may be high
- Synchronisation of access could also be costly
Improving Performance using HTTP features

HTTP 1.1 includes several methods for optimising delivery of Web content. Includes information about the following:

- Connection management
- Caching Support
- Content negotiation

Using these features, you can tune your application to use caching effectively and tune your server for scalability and performance.
Improving Performance using HTTP features

Caching Static GET

The HTTP GET method implements queries - considered to have no ‘side effects’ on the server.

Conditional GETs requests have one of the following optional header fields: If-Modified-Since, If-Unmodified Since, If-Match, If-none-match, If-Range. E.g. the client could make a request, like so:

GET /index.html HTTP/1.1
If-Modified-Since: Mon, 17 Sep 2012 18:00:00 GMT

If the requested resource has not been changed since the given date, the server can avoid a needless data transfer by responding with:

HTTP/1.1 304 OK

Proxy servers also obey these rules and therefore, improve both client and server performance.
Improving Performance using HTTP features

Caching Dynamic GETs

GETs can also result in invoking processing logic in which case, the information produced by the process is returned. E.g.

GET search.html?name=Srikumar HTTP/1.1

In this case, the server can respond with:

HTTP/1.1 200 OK
<... other header fields ...>
Expires: Fri, 21 Sep 2012 00:00:00 GMT
<HTML>
<HEAD></HEAD>
<BODY>
Expires vs. If-Modified-Since: Responsibility of Server vs Client
Improving Performance using HTTP features

Other Caching Hints

HTTP POST changes the state of a resource on the server and therefore should not be cached. Therefore, result of a POST should only be HTTP 200 or 204.

Currently, we have used date as the means to check the freshness of a page. Assumes the existence of a globally synchronised clock $\Rightarrow$ distributed systems FAIL.

Alternately, tag a page every time it is modified and then use If-Match, If-None-Match.

E.g.

GET /foo.html
If-None-Match: "0-90-3ad8c299"
Improving Performance using HTTP features

Fine-tuning Cache
Using Cache-Control: header

- **max-age=[seconds]**, **s-maxage** Specifies the maximum age (in seconds) that the content is fresh
- **public**: Content can be cached by any element in the network path
- **private**: Content can only be cached uniquely (i.e. at the browser end) rather than in a shared cache (i.e. proxies)
- **must-revalidate**: browser must check with a server about the freshness of content.
- **no-cache**: forces client to resubmit request to server for the content, every time
- **no-store**: do not store the content under any circumstance.

HTTPS content is not cached by the proxy servers. However, using Cache-control:public can make some of that content cacheable.
Improving Performance using HTTP features

Connection Management

Persistent connections avoid the overhead of setting up a TCP/IP connection every time a client makes a request. E.g. To download images on a single page.

Advantages:
- Reduced CPU, bandwidth and memory demand on servers
- Better congestion control - longer TCP connections.

HTTP/1.1 assumes all connections are persistent. Server can close a connection by putting `Connection: close` in its response header.

Persistent connections enable request pipelining - client can make a series of requests without waiting for response to each of them.
Improving Performance using HTTP features

Connection Management

Excessive HTTP connections can overload a server and reduce performance.

Reduce the number of connections via:

- Use as few images as possible - use CSS for menus and navigation
- Reduce heavy objects - graphics, Flash animations
- Use AJAX in Client UI

Use persistent connections.
Improving Performance using HTTP features

Content Negotiation

Different representations of the same resource in different formats. Client determines which format to request

- `Accept:  text/html,application/xml;q=0.9, */*;q=0.1`
- `Accept-Charset:  iso-8859-5, unicode-1-1;q=0.8`
- `Accept-Encoding:  gzip, deflate`
- `Accept-Range:  <number of bytes>`

Tip 1: Gzip encoding. Web client sends out header. In response, server indicates compression via `Content-Encoding: gzip`

Tip 2: Support `Accept-Range` for file downloads.
Improving Database Access

For any complex Web application, database access holds the key to its performance and scalability.

**When to use the database?**

1. To store persistent data - to be retained forever.
2. To store structured data - composed of well-known attribute types that can be combined in different ways (e.g., customer data).
3. To conduct transactions - data that has to be updated and accessed in a serial manner.

**When NOT to use the database?**

1. The data is unstructured e.g., images.
2. The data is not combined with other data - storing data in-memory may be better.
3. The data is static and is not updated frequently.
Improving Database Access

Normalization - modeling data so as to optimise disk access as well as ensure correctness.

However, normalization can lead to queries becoming join operations across multiple tables → bad for performance!

Therefore, some tables may have to be denormalized - by replicating data in another table or by storing results of an intermediate query. E.g.

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>ORDER_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-Mar-2010</td>
</tr>
<tr>
<td>2</td>
<td>1-Mar-2010</td>
</tr>
<tr>
<td>3</td>
<td>1-Mar-2010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>PRODUCT_ID</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>544</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2218</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4782</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>VALID_FROM</th>
<th>VALID_TO</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>544</td>
<td>1-Jan-2010</td>
<td>1-Jun-2010</td>
<td>23.50</td>
</tr>
<tr>
<td>2728</td>
<td>1-Jan-2010</td>
<td>1-Jun-2010</td>
<td>27.50</td>
</tr>
</tbody>
</table>

How to optimise finding the value of an order?
Other techniques include using stored procedures and avoiding triggers.
Improving Database Access - PreparedStatement object

Query processing generally comes down to three phases: query parsing (syntax checking), query plan generation and plan execution. The first phase is costly, especially if we repeatedly execute the same query.

PreparedStatement object is compiled ahead of time (i.e., precompiled) and can be executed many times as needed. - as opposed to Statement object which sends a SQL string to the database each time for parsing.

```java
PreparedStatement ps = conn.prepareStatement("SELECT first_name FROM employee");
ResultSet rset;
for (int i=0; i<1000; i++) {
    rset = ps.executeQuery();
    /* ... do something more ... */
}

Statement stmt = conn.createStatement();
ResultSet rset;
for (int i=0; i<1000; i++) {
    rset = stmt.executeQuery("SELECT first_name FROM employee");
    /* ... do something more ... */
}
```
A more realistic case is that the same kind of SQL statement is processed over and over (rather than a static SQL statement).

```sql
SELECT * FROM employee WHERE id = 3;
SELECT * FROM employee WHERE id = 7;
SELECT * FROM employee WHERE id = 25;
SELECT * FROM employee WHERE id = 21;
... 
SELECT * FROM employee WHERE id = ?
```

In PreparedStatement, a place holder (?) will be bound to an incoming value before execution (no recompilation).

```java
PreparedStatement ps = conn.prepareStatement("SELECT * FROM employee where id=?");
ResultSet rset;
for (int i=0; i<1000; i++) {
    ps.setInt(1,i);
    rset = ps.executeQuery();
    /* ... do something more ... */
}
```

This results in more efficient querying.
PreparedStatement and Parameters

Delete a Car (which car?)

<HTML><HEAD><TITLE>Delete a Car</TITLE></HEAD>
<%@ page import="java.io.*, java.sql.*"%>
<BODY><CENTER><H3>Car Deletion</H3>
<% String inCarNr = request.getParameter("carnr");
    String delete_car = "DELETE FROM Cars WHERE CarNr= ?";
%>
<H2>SQL: <%= delete_car %></H2>
<% try {
    Class.forName("org.hsqldb.jdbcDriver");
    Connection connection = DriverManager.getConnection("jdbc:hsqldb: ... 
    PreparedStatement statement = connection.prepareStatement(delete_car);
    statement.setString(1, inCarNr);
    int rowsGone = statement.executeUpdate();
    if (rowsGone==1) {
        <H3>Car nr <%= inCarNr %> deleted.</H3>
        // snipped ...
    }
}</BODY>
</HTML>
Performance of PreparedStatements vs Statements

```java
ResultSet rset;
/* Using a Statement */
long elapsed = System.currentTimeMillis();
Statement stmt = conn.createStatement();
for (int i=0; i<10000; i++)
    rset = stmt.executeQuery("SELECT first_name FROM employee WHERE id=1");
elapsed = System.currentTimeMillis() - elapsed;
System.out.println("Statement approach took " + elapsed+" ms");

/* Using a PreparedStatement */
elapsed = System.currentTimeMillis();
PreparedStatement pstmt = conn.prepareStatement(
    "SELECT first_name FROM employee WHERE id=1");
for (int i=0; i<10000; i++)
    rset = pstmt.executeQuery();
elapsed = System.currentTimeMillis() - elapsed;
System.out.println("PreparedStatement approach took " + elapsed+" ms");

Statement approach took 17275 ms
PreparedStatement approach took 7881 ms

17,272/7.881 = 2.19 (A simple query - Statement is around twice as slow)
```
Performance of PreparedStatements vs Statements

Try something more complex ...

```sql
SELECT e.first_name, e.last_name, os.num, st.name
FROM employee e, session_type st,
     (SELECT employee_id, count(*) num, type_id
      FROM order_session
      GROUP BY employee_id, type_id) os
WHERE e.id = os.employee_id
 AND os.type_id = st.id
 AND e.id = 1
```

Statement approach took 99954 ms
PreparedStatement approach took 87856 ms

99954/87856 = 1.14 (hmm ... PreparedStatement is not as effective here ...)

- The bulk of the time isn’t spent in query parsing, but in execution
- The query joins three tables and performs a nested query in addition to grouping, etc.
- Query parsing is only one of three phases involved in query processing
Performance of PreparedStatements vs Statements

/* Using a Statement */
long elapsed = System.currentTimeMillis();
Statement stmt = conn.createStatement();
for (int i=0; i<10000; i++)
    stmt.executeUpdate("INSERT INTO numbers VALUES (" + i + ")");
elapsed = System.currentTimeMillis() - elapsed;
System.out.println("Statement approach took " + elapsed+" ms");

/* Using a PreparedStatement */
elapsed = System.currentTimeMillis();
PreparedStatement pstmt = conn.prepareStatement(
    "INSERT INTO numbers VALUES (?)");
for (int i=0; i<10000; i++) {
    pstmt.setInt(1, i);
pstmt.executeUpdate();
}
elapsed = System.currentTimeMillis() - elapsed;
System.out.println("PreparedStatement approach took " + elapsed+" ms");

Statement approach took 36263 ms
PreparedStatement approach took 27930 ms

36263/27930 = 1.3 (inserting data itself is expensive and dwarfs the impact of using pre-compiled queries.)
long elapsed = System.currentTimeMillis();
PreparedStatement ps = conn.prepareStatement("INSERT INTO numbers VALUES (?)");
for (int i=0; i<10000; i++) {
    ps.setInt(1, i);
    ps.executeUpdate(); /* auto commit */
}
elapsed = System.currentTimeMillis() - elapsed;
System.out.println("Always-commit approach took " + elapsed + " ms");

conn.setAutoCommit(false);
elapsed = System.currentTimeMillis();
PreparedStatement p = conn.prepareStatement("INSERT INTO numbers VALUES (?)");
for (int i=0; i<10000; i++) {
    p.setInt(1, i);
    p.executeUpdate();
}
conn.commit(); /* single commit */
elapsed = System.currentTimeMillis() - elapsed;
System.out.println("Single-commit approach took " + elapsed + " ms");

Always-commit approach took 28761 ms
Single-commit approach took 9905 ms

28761/9905 = 2.9 (do not overcommit!)
Transaction Management

By default, JDBC commits each update when you call `executeUpdate()`. Committing after each update can be suboptimal in terms of performance. It is also not suitable if you want to manage a series of operations as a logical single operation (i.e., transaction).

```java
Class.forName("... // the driver
Connection conn = DriverManager.getConnection("... // the DB url
/* Do not commit each update immediately */
conn.setAutoCommit(false); // configuring connection

Now you could:

conn.setAutoCommit(false); // this marks START_TRANSACTION
Statement stmt = conn.createStatement();
try {
    stmt.executeUpdate("UPDATE ACCOUNTS SET bal=bal - 100 WHERE id = 101");
    stmt.executeUpdate("UPDATE ACCOUNTS SET bal=bal + 100 WHERE id = 102");

    conn.commit();

} catch (Exception e) {
    conn.rollback();
}
```
Scaling Databases - Replication

Data is replicated between different machines so that it is available from multiple points. These techniques are discussed specifically w.r.t MySQL.

Master-Slave Replication

- Master copies changes periodically to slaves - slave has a up-to-date copy of the master.
- Data on the slave cannot be modified in any other way - therefore it is consistent.
- Transactions are only copied when they are committed.
- Read throughput increases - write operations also increase.
Scaling Databases

Tree Replication

- Slaves can be structured in a hierarchy where a slave acts as a master to other slaves.
- Only a portion of the data can be replicated to slaves beneath
- Adv.- slaves can be specialised
- Disadv. - if a slave goes down, an entire branch can turn inconsistent.
Scaling Databases

Master-Master Replication

- Two masters that are slaves to each other.
- A record can only be inserted into one master, the other updates from it. If records are added to both masters - violates PK constraints.
- Consistency cannot be guaranteed.
- Good for redundancy, both reads and writes can be performed even with a single master.
Scaling Databases

**Partitioning** - Split the DB into chunks that can be read and written independently

**Clustering**

- Splitting the DB into multiple chunks or clusters
- Should ensure that joined tables are not split across clusters - difficult to manage
- Each cluster is a master - requires extra connection, extra overhead
Scaling Databases

Federation

- Slice the data into arbitrarily sized chunks called shards stored on different machines.
- Queries should be limited to single shard. Cross-shard queries should be avoided.
- Therefore, splitting should be based on logical grouping of data (by users, say) and denormalization should be employed.
- Logic is much more complicated and consistency cannot be guaranteed.
Request Load Balancing

Horizontal scaling is a cheap way to add capacity for your web application. However, the question remains, who routes the requests to the new servers when they appear on the network?

Here, the public IP is termed as *Virtual IP (VIP)* and is the address of the load balancer (*virtual server*). The IPs behind it are the *real* servers that serve the actual content.
Request Load Balancing

Load Balancing can be performed at two levels:

- **Transport-level**: Routing requests at the network-level, either using TCP/IP or DNS-based approaches.
  - **Adv.**: Fast, relatively easy to setup, is invariant of application logic
  - **Disadv.**: Can lead to huge overheads in managing requests at the application level

- **Application-level**: Using application logic to distribute workload among instances.
  - **Adv.**: Session management is easier, Data overhead is lower.
  - **Disadv.**: Requires knowledge of dist. systems, slower

Furthermore, load balancing logic can be implemented at two levels - hardware and software.
Request Load Balancing

Using Hardware Load Balancers

1. Using a dedicated hardware appliance for request routing
2. Adding and removing servers behind VIP takes effect instantly
3. Load is balanced according to the content provider’s wish
4. Provides automatic failover in case of server failure.

Disadvantages:

- Expensive
- Configuration is usually troublesome especially if there are many VIPs
Request Load Balancing

Using Software Load Balancers

1. Using software installed on a commodity server running a commodity OS
2. E.g. Perlbal, Pound[http://www.apsis.ch/pound], Linux Virtual Server
3. Load balancing responsibility can be switched around with other servers

Disadvantages:

- Lower performance and more failure-prone than dedicated load balancing
Request Load Balancing

Three blind mice, three blind mice, see how they run

Using DNS - Easiest way of load balancing

1. "A" record is created for each server in an application’s domain, which is then retrieved by the DNS server.
2. When a user enters a URL, the browser asks the DNS server to return a list of records for that domain.
3. The DNS server replies with a list of IP addresses in random order and the browser tries each one of them in turn.
4. Also can be used for geographical load balancing.

However, there are problems with this approach:

- DNS servers cache server records, so if a server is down, it is not removed from the records immediately.
- User browser may or may not try other addresses in the list.
- Traffic is directed at random and not as per load - if a large ISP caches the DNS records, most of the users from this ISP will be sent en masse to the first server in the list.
Request Load Balancing

TCP/IP-based Load Balancing

1. Request is sent to the Virtual IP by the client.
2. The load balancer at the Virtual IP picks a target server with a real IP and rewrites the packet header.
3. Request is forwarded to the real IP.
4. Response can either go directly from the target server to the client or can be sent via the load balancer real IP vs virtual IP.
5. Failover is ensured by having a backup load-balancer.

However:

- There is immense workload on the load balancer (computation + txn). The best ones are usually implemented in h/w ($$$$
- Also, needs implementation of a distributed cache to avoid redundant data txn.
- Application-level session tracking is required for smooth operation.
Request Load Balancing

Application-Level Load Balancing

- **Technique 1: sticky sessions** - Use session-id/cookie-id as a key for request partitioning - Ensures every request in a session goes to the same server.
- **Technique 2: cookie injection** - The id of the server is stored in a temporary cookie on the client-side.
- **Technique 3: HTTP redirects** - The load balancer redirects the client to the real server via HTTP 301 response.
- **Technique 4: stateless logic** - Store the session-id in a cache and allow any server to recreate session data from the DB.

Drawbacks:

- Additional delay is introduced into the request chain.
- HTTP redirects: expose the server-side infrastructure to client
- HTTP redirects: no failover is possible if server collapses.
- Generally, application-level LBs are coupled with transport-level LBs.