Chapter

1

Web Application Scalability

CHAPTER AUTHORS

Hoang Duc

Juliana Ung Bee Chin

Nguyen Van Quang Huy

CONTENTS

1 INTRODUCTION

Unlike functional requirements which describe what the system *should do*, non-functional requirements are system-wide attributes that specify how a system *should be*. Non-functional requirements are often referred to as the qualities of a system, and are described by their attributes.

There are three classes of qualities, namely system, business and architectural qualities (Bass, Clements and Kazman 2003).

Category	Example attributes
System qualities	Availability, performance, security, testability, portability, scalability
Business qualities	Time to market, cost and benefit, integration, product lifetime
Architectural qualities	Conceptual integrity, correctness, completeness

Table 1 Categories and example attributes of non-functional qualities[1]

Quality attributes often conflict with each other and a system designed to satisfy some qualities may need to trade off the others. For one, almost any quality attribute has a negative impact on the performance of a system. For instance, to achieve portability, the main technique to do so is to isolate system dependencies, which introduces overhead into the system's execution, typically as process or procedure boundaries, and this in turn hurts performance (Bass, Clements and Kazman 2003).

In this book chapter, we focus on scalability as an attribute of web applications, related concepts, and means to achieve it.

Scalability as a general quality attribute refers to the ability of a system to handle growing demands. The kinds of demands are subjective and will depend on the system in question. Regardless, scalability is one of the most valuable quality attribute of a system. Not only is a scalable system assured to perform well under increasing demand, it would also reduce the need of having to redesign the system under such challenges, and this translates to business gains such as the mitigation of possible financial loss or decreased customer confidence.

The following section defines what scalability means in the context of web applications.

Quality attributes of web applications include reliability, usability, security, scalability, maintainability and time-to-market. A prioritization of these attributes would depend on the application domain. For example, in a real time web application, reliability and performance would take precedence over maintainability or security. This prioritization would be reversed in a mass market application.

As described in the previous section, scalability refers to the system's ability to handle demands. Demands on a web application may be of two forms: traffic volume and data volume. From the systems viewpoint, a scalable web application ensures that an acceptable performance, availability and other quality attributes are maintained despite the growth of traffic and dataset. In a commercial setting – and this may normally be the case for most web applications that are able to garner that much traffic and dataset increase in the first place – decisions on what, how and when to scale are dependent on many other factors beyond the technical and engineering

[1] The IEEE ISO 9126 standards specification details a classification of quality attributes. This classification will not be covered in this book chapter.

realms, including cost, return on investment, development team expertise, team size, and time-to-market.

Hence, the work needed to ensure a scalable web application may well start from the levels of organization and processes, even before focusing on a scalable design and architecture of the system(Martin Abbott 2010). The focus of this book chapter however, will be on the design, architecture and implementation of a scalable web application.

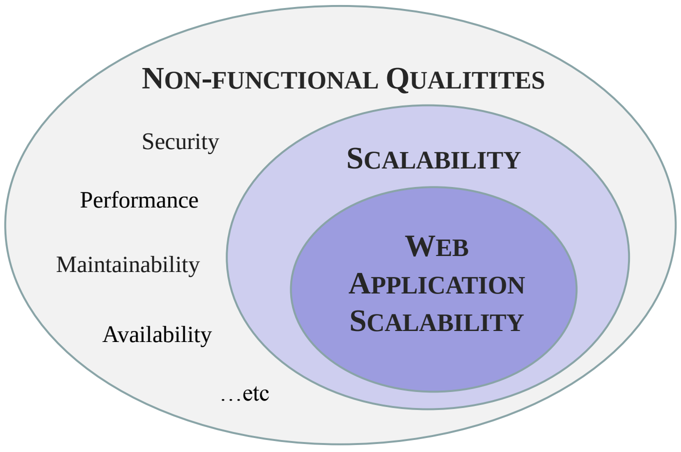

Figure 1 The focus of this book chapter

A web application, as opposed to a web site, is a network-based application that uses the web browser as a way for end users to interface with the system. Unlike web sites, users often form and manipulate the contents of web applications. Scalable data manipulation is hence important in ensuring the scalability of web applications. For the purpose of discussion, we categorize data into two forms:

- those normally stored in the database
- those stored as files such as media content and documents

It goes without saying that the web server is a central component of a web application. Understanding and selection of suitable web servers is hence vital for a scalable web application.

In this book chapter, we identify the database, file storage and web (or application) server as the three components for which scaling strategies will be discussed.

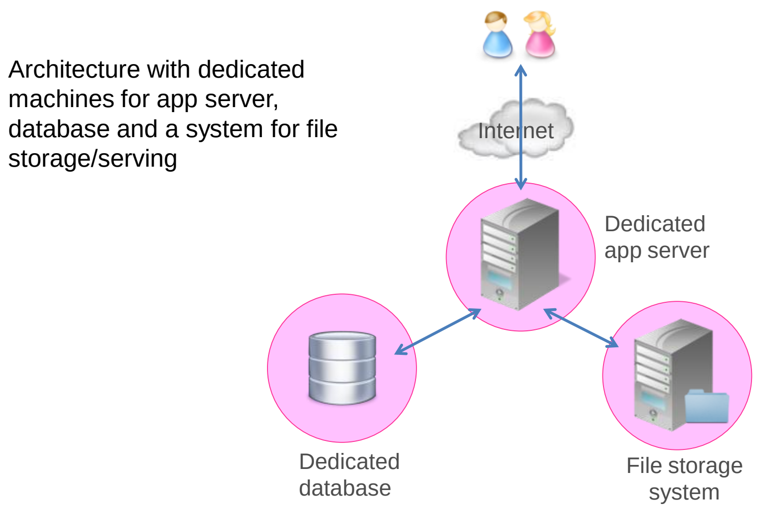
Architecture with dedicated machines for app server, database and a system for file storage/serving
Internet
Dedicated app server
Dedicated database
File storage system

Figure 2 The three scalable components discussed in this book chapter

DEFINTION A ***web server*** is a computer in a network that receives HTTP requests and serves web pages to web browsers. An ***application server*** ("app server") not only includes the functionality of the web server, but also offers support for database independence, security, etc. This book uses the two terms interchangeably.

2 Scaling Methods

There are two main methods of scaling, namely scaling up (also known as *vertical scaling*) and scaling out (*horizontal scaling*).

2.1 Scaling Up

Scaling up or vertical scaling refers to resource maximization of a single *node* to expand its ability to handle increasing load. In hardware terms, this includes adding processing power and memory. In software terms, scaling up may include optimizing algorithms and code. Optimization of hardware resources, such as parallelizing or having optimized number of running processes are also considered techniques of scaling up (Wikipedia: Scalability n.d.).

DEFINITION A ***node*** is a single computing or storage unit.

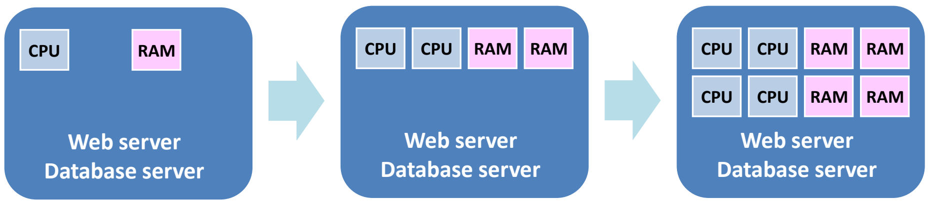

Figure 3 Example of scaling a single node containing both web and database servers

Although scaling up may be done fairly easily, this method suffers from several disadvantages. Firstly the addition of hardware resources results in diminishing returns instead of linear scale (Figure 4). The cost for expansion also increases exponentially. In addition, there is the inevitable downtime requirement for scaling up. If all of the web application services and data reside on a single node, vertical scale on this node does not guarantee the application's availability.

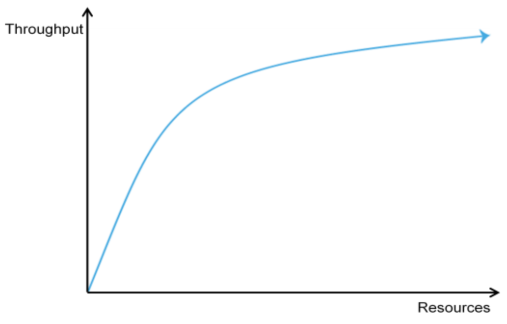

Figure 4 Diminishing returns on vertical scale

2.2 Scaling Out

Scaling out or horizontal scaling refers to resource increment by the addition of nodes to the system. Having multiple nodes allows us the possibility of keeping the system up even if some nodes go down, thus, avoiding the "single point of failure" problem and increasing the availability of the system.

3 A Deeper Look at Scaling

An *implementation* is the usage of a particular technology and is a choice made after considering system needs as specified by the *design and architecture*. That *implementation* and *design and architecture* are separate concepts should be kept in mind when designing for a scalable system. A technology agnostic design and architecture (TAD and TAA respectively) is able to increase scalability through a reduction of both the cost to scale and risk associated with scale (Martin Abbott 2010).

DEFINITION Design and architecture that are technology agnostic i.e. ***TAD*** and ***TAA*** are not based on specific vendor technologies, but are instead based on industry standard terms. This ensures protection from the effects of commoditization that may put the business and project at an undesirable cost dependency on solutions by a particular technology provider.

Here, we introduce a concept of scalable architecture using the **AKF Scale Cube** (Martin Abbott 2010). Each axis of the AKF cube describes a general scaling approach that may be applied to different areas of scale in a web application system (such as data or services)[2].

The X-axis of the AKF scale cube represents the cloning of data and services. An X-axis scale results in the original workload being *split evenly* among groups of resources, with each group doing the *same kinds of work as the original*.

For example, when applying an X-axis scale to a student team, all team members will work on all deliverables, with each team member shouldering even amounts of divided workload in each deliverable. This appears a good strategy. Whenever the lecturer increases the workload, the team would hire a new member and redistribute his demands. However, the lecturer's demands may grow over time and the deliverable requirements may come in various sizes. Some deliverables take a long time to complete and slow down delivery of even the lighter deliverables. Since each member is working on the same kinds of deliverable, this bottle neck affects all, and hence the entire team.

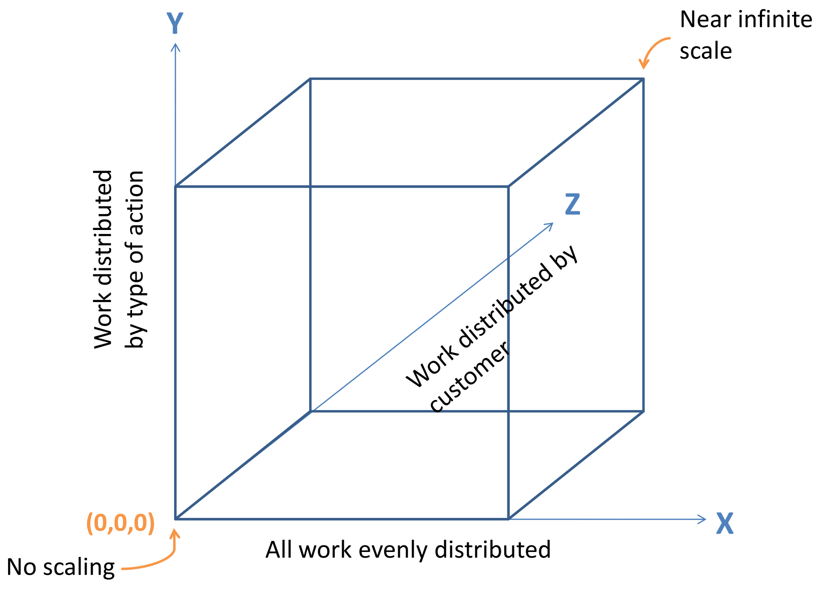
Y
Near infinite scale
Work distributed by type of action
Z
Work distributed by customer
(0,0,0)
No scaling
All work evenly distributed
X

Figure 5 AKF scale cube

[2] Note that these scaling approaches may also be applied to scaling the organization and processes that support the web application.

The Y-axis scale avoids this bottleneck by separating work according to the type of data or services. It may be viewed as splitting work by responsibility for an action, and is often referred to as service or resource oriented splits.

Y-axis scale splits the work for any specific action (or set of actions) and its associated data from other types of actions. Unlike the X-axis scale which is merely a distribution of work among clones, the Y-axis scale distributes work to specialized groups. In a Y-axis scale, it is possible to pipeline or parallelize processing flows. In the case of the student team, each member is now tasked with a specific deliverable, according to one's expertise. This also means that a member who works on a lighter deliverable is able to work on many instances of this deliverable and not be slowed by a different, more demanding deliverable. Work throughput suffering of the demanding deliverable is also isolated to only the member who "specializes" on it.

The Z-axis represents distribution and segmentation of work by customer e.g. customer need, location or value. Unlike Y-axis splitting whose work distribution is based on the type of work or service internal to the web application, Z-axis scale splits work or data based on customer (or user, or requestor) considerations.

Suppose that Facepage, a hypothetical social networking application, services users from around the world, and earns its revenue from advertisers. Additionally, its advertisers are given accounts to which they may log in to Facepage and adjust their advertising preferences. The log in page and services for advertisers, although done through the same Facepage system, are different from the ones for normal users. In this case, Facepage may group their customers into two: normal users and advertising users, and perform a Z-axis scale based on categorization. This would entail separation of data and code related to advertising users from those related to normal users.

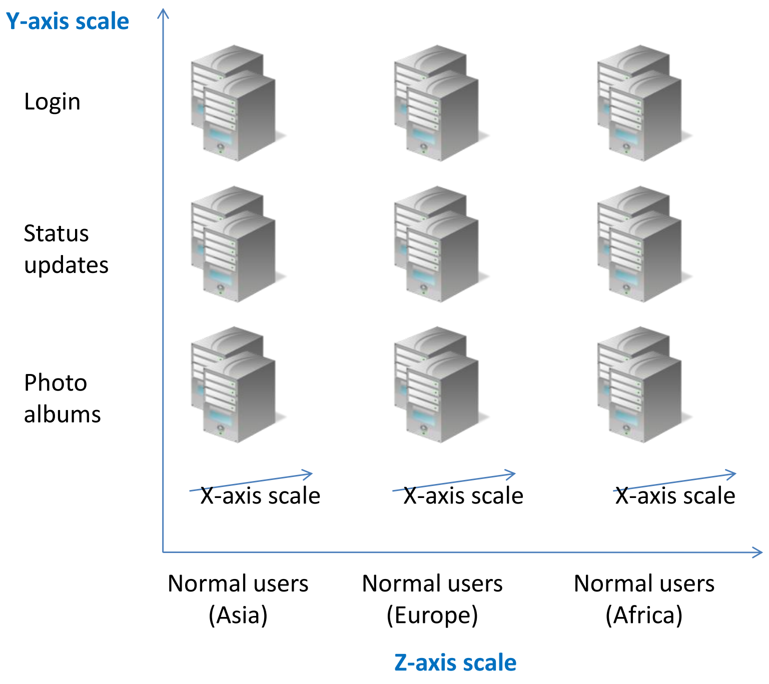

Figure 6 Facepage's scale (advertising customers not shown)

Z-axis scaling is as easy to conceptualize as the previous scaling axes, but is more challenging and costly to implement. However, when performed, the benefits may outweigh the cost over time.

Every web application's scaling needs vary from the next, as it is subject to the services it offers, its users, and even to cost and projected growth. If an application is expected to stay small and grow slowly, it may never need more than one axis of scale. If growth is quick, and foreseen to be violent and unexpected, then it would be better to plan for two or all three axes of scale.

Even when a web application starts within a monolithic system, it is still useful to apply the AKF cube concepts in its early design. This may be in the form of implementing logical groupings of data and code in preparation for future Y- or Z-axis scale, or could be separation of web servers to handle different types of data within a single node.

It is possible – as it is often done in practice – to perform an axis scale within another axis. For example, Facepage may further do Z-axis splitting on its normal users and group them by geographical location. In practice, this split will translate to more data separation, and may also allow Facepage to place service resources in locations physically nearer to different user groups to speed up data transactions. Furthermore, Facepage may perform X-axis scale on each of these groups to spread out workload to more resources, at the same time ensuring availability with the X-axis clones.

The table below summarizes the AKF scale cube axes.

	X	**Y**	**Z**
Concept	Even distribution of the same work across multiple entities.	Distribution and separation of work responsibilities or data meaning among multiple entities.	Distribution and segmentation of work by customer e.g. customer need, location or value.
Advantages	Easy to implement Least costly	Increase of throughput based on reduction of data or instruction set (in each entity). Some fault isolation.	Increase of throughput based on reduction of data (in each entity). Instruction set may be reduced but dependent on design. Fault isolation.

Table 2 Short summary of the AKF scale cube axes

4 Scaling the Database

The database is a system for organizing, storing and retrieving large amounts of data. In the context of web applications, the database write to read ratio depends on the functions of the application. Our hypothetical Facepage is likely to incur a higher write to read ratio – as users post updates – compared to, say, a personal resume hosting site where user updates are not as frequent. This difference in write to read ratio is one of the factors that determine the strategy when designing a scalable database.

A scalable database means that the database should still be able to perform well under increasing traffic and dataset. Additionally, it should also be maintainable (Henderson 2006). In this book chapter, we look at the basic concepts of scaling up and out a relational database. In particular we discuss indexing as a form of scaling up and replication, and clustering and federation as forms of scaling out.

4.1 Scaling Up with Indexing

The database index is a data structure that speeds up the retrieval of data. Typically the index comprises an index key, and pointers to data. In a relational database, the index key is constructed with one or a combination of keys selected from the columns of a table. The index pointer points to the location of a row in the original database table, as shown in the figure below.

ID	FirstName	LastName	Email
1	John	Carter	1@2.3
2	Allen	Tan	1@2.4
3	Bob	Markzinski	1@2.5
4	Susan	Roberts	1@2.6
5	Julia	Lyline	1@2.7
6	Mark	Yankze	1@2.8
7	Twain	Shernie	1@2.9
8	Shania	Klow	1@2.10
9	Susan	Boyle	1@2.11
10	Bob	Goranski	1@2.12
11	Thomas	Jackson	1@2.13
12	Heinz	Wing	1@2.14

Table

FirstName	Pointer (ID)
Allen	2
Bob	3
Bob	10
Heinz	12
John	1
Julia	5
Mark	6
Shania	8
Susan	4
Susan	9
Thomas	11
Twain	7

Index

Figure 7 An index created with FirstName as the index key

The rule of thumb is to pick as the index key, a field that limits the search results. The examples below, taken from (Introduction to Database Indexes n.d.), demonstrate this on MySQL.

EXAMPLE

Suppose we have a table created with the query below:

```
CREATE TABLE subscribers (
   subscriberid INT PRIMARY KEY,
   emailaddress VARCHAR(255),
   firstname VARCHAR(255),
   lastname VARCHAR(255)
);
```

In our web application, we expect frequent searches to the table based on the *emailaddress* field e.g. with the query

```
SELECT firstname, lastname FROM subscribers WHERE
emailaddress='email@domain.com';
```

We should then use *emailaddress* as the index key with the SQL command:

```
CREATE INDEX my_idx ON subscribers(emailaddress);
```

EXAMPLE

Suppose there are two tables:

```
CREATE TABLE newsitem (
  newsid INT PRIMARY KEY,
  newstitle VARCHAR(255),
  newscontent TEXT,
  authorid INT,
  newsdate TIMESTAMP
);
```

```
CREATE TABLE authors (
  authorid INT PRIMARY KEY,
  username VARCHAR(255),
  firstname VARCHAR(255),
  lastname VARCHAR(255)
);
```

Our frequent query below would benefit from an index on the *newsitem.authorid* field as records are quickly retrieved from the index before it is matched to the authors table. There is no need to consider optimizing lookup for *authors.authorid* as it is a primary key and is indexed by default.

```
SELECT newstitle, firstname, lastname FROM newsitem n, authors a WHERE
n.authorid=a.authorid;
```

This example particularly shows the benefits of indexing field involved in a table join.

Internally, the index may be implemented as trees or hashes and so is able to enhance the computational complexity of retrieval, but on the downside, will take up additional storage space. While the index enables faster reads, writes become slower as references to the index has to be added on update, addition or removal of data.

4.2 Scaling Out the Database

The AKF scale cube concepts may be applied to scale out a database. The figure below narrows the scope and definition of each cube axis to the context of database scaling. Table 3 summarizes the general scale concepts with database-specific scale concepts.

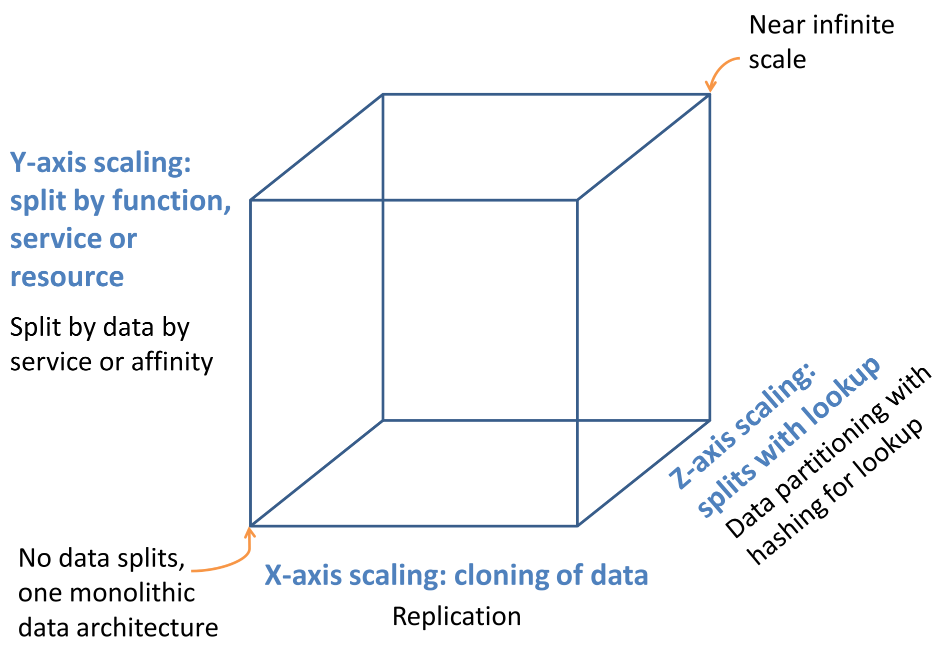
Near infinite scale
Y-axis scaling: split by function, service or resource
Split by data by service or affinity
Z-axis scaling: splits with lookup
Data partitioning with hashing for lookup
No data splits, one monolithic data architecture
X-axis scaling: cloning of data
Replication

Figure 8 AKF database scale cube

	X	Y	Z
AKF concept	Even distribution of the same work across multiple databases i.e. cloning of data.	Distribution and separation of data meaning i.e. data splitting by function, service or resource.	Distribution and segmentation data by customer e.g. customer need, location or value
Concept discussed in book	Data replication	Data partitioning	Data partitioning

Table 3 Concepts to scaling out a relational database, with reference to the AKF scale axes

4.3 Scaling out using replication

In database replication, copies (replicas) of data are stored on multiple machines. Clients may read from any of these replicas. Whenever a write occurs, the updated data is replicated to all machines to ensure consistency. The database server that does this replication is called the master, while those at the receiving end of the replication are called slaves. The following sub-sections are summarized from *Building Scalable Web Sites* (Henderson 2006).

4.3.1 Database replication using the master-slave model

From a single server, the simplest replication model to scale out to is the single master and single slave setup (Figure 9). Client web server(s) may now read from any of these two machines. However, all writes must only be directed to the master, who will subsequently replicate data updates to the slave.

This simple two-machine configuration may be extended to more slaves (Figure 10).

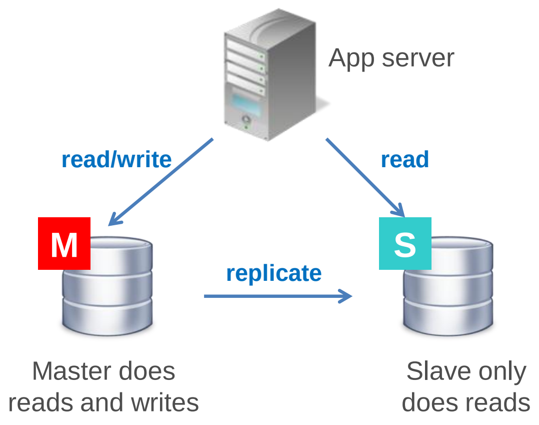

Figure 9 Master-slave replication model

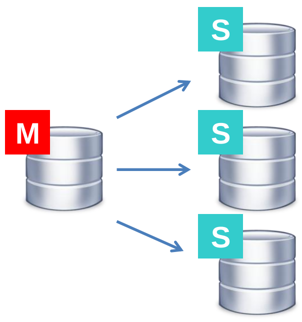

Figure 10 Master-slave replication model with multiple slaves

Database replication enables read demands to be distributed among multiple machines (Figure 11). This is precisely the concept of X-axis scale. However, all machines still need to perform every write operation (Figure 12).

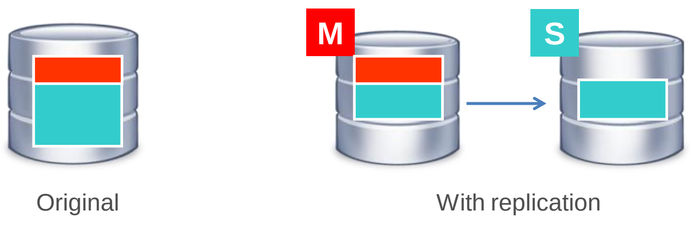

Figure 11 Distributed reads

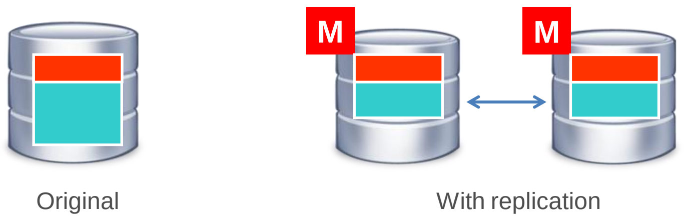

Figure 12 Replication unable to scale writes

Beneath the hood, data replication between a master and a slave is done via threads on both the master and slave. In very large web applications, it is possible to see configurations with hundreds of slaves to a single master. In such cases hundreds of replication threads on the master need to be context-switched to do ensure replications on all salves. This may lead to significant load on the master machine. This can be eased by converting one of the slaves to a master who will be responsible in updating a number of other slaves. In Figure 13, instead of having one master replicating to six slaves, there are now two masters replicating to three slaves each (in a real set up the slaves may number to say, 100 to one master).

There are two main disadvantages in a tree configuration: since replication has to be relayed from master to slave, a tree configuration will lead to longer replication delays in the lower tier slaves. Also, if the master in a middle tier fails, then replication fails for its slaves, leading to stale databases. In general, a master-slave replication model introduces a single-point-of-failure at the master machine (Figure 14).

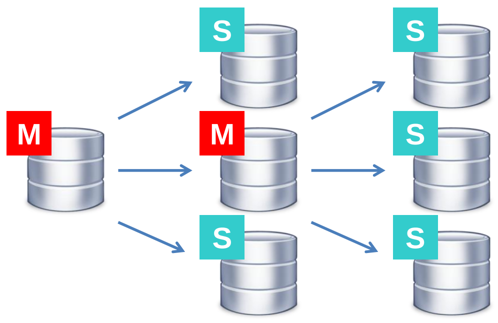

Figure 13 Master-slave tree configuration

However, in a tree configuration, this “subtree of slaves” dependency on a middle-tier master may be utilized to our advantage. Since replication is only relayed down from master to the slaves beneath it, and never up the tree, we may create a branch of slaves with specialized function.

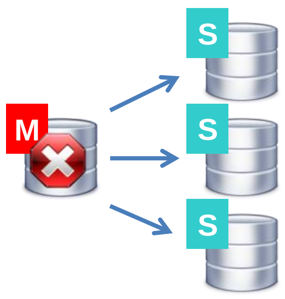

Figure 14 Single point of failure in master-slave model

<u>EXAMPLE</u> TAKING ADVANTAGE OF A TREE CONFIGURATION
Suppose we run web application where users may post current events. We have a master-slave tree configuration, such as in Figure 13, set up with MySQL databases using the InnoDB storage engines. We would now like to add a search feature to our application that will enable searches based on keywords (which may be tags by users, or event topic, or even from words in event description). MySQL allows a FULLTEXT type indexing that is able to support such a search feature. However, the FULLTEXT indexing is only available with MySQL's MyISAM storage engine.

We could issue commands to change all the storage engines in our tree from InnoDB to MyISAM and then create FULLTEXT indexes on our tables. These commands will be replicated from the root master to all the machines in the tree and the changes will be made to all. There is a major disadvantage to this setup: the FULLTEXT index takes a lot of additional space, and as discussed previously, slows down database writes. Since events in our application may be updated or added frequently, this is undesirable, as index updating will need to be replicated to the entire tree. Yet another major downside is that the InnoDB storage engine is *transactional* (refer section 8 for definition), while MyISAM is not. It is hence not a good decision to convert the storage engines of all data to MyISAM.

Instead of doing this, we may choose instead to specialize the three bottom tier machines in Figure 13 to support this search feature, meaning that we will only change those three slaves into MyISAM storage engines with the FULLTEXT indexing that we need. Our application will direct search queries to these three slaves. When our application does a write, the replication is still done down the entire tree, but only the three specialized slaves would do the additional FULLTEXT index updates on top of table updates.

4.3.2 Multi-master Replication Model

Instead of having a one-way replication between a master and slave, the multi-master model enables each master to replicate data to the other. Figure 15 and Figure 16 are possible example configurations with the multi-master model.

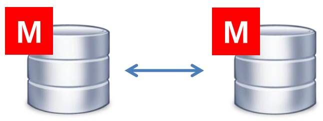

Figure 15 Multi-master replication model

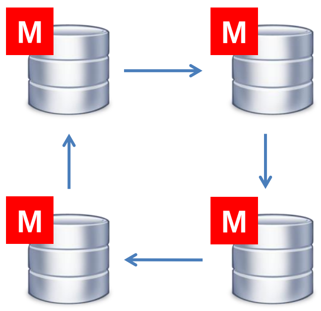

Figure 16 Multi-master replication model in a ring configuration

Since there are now more than one machine responsible for updates, the multi-master model eliminates the single-point-of-failure that the master-slave model has. However, this model introduces the danger of data collision. This may happen if we use the primary ID auto-incrementing feature, and were to insert two different records into each of the masters simultaneously.

There are several ways to handle data collision. We may author the application code such that writes to certain tables are only issued to one master and writes to other tables are issued to the other master. We may also avoid the auto-incrementing feature and use alternatives such as other unique keys or other services to generate unique IDs. These are not necessarily convenient workarounds, depending on our needs.

Replication configurations can be designed to suit our web application. We may for example, combine both multi-master and master-slave models as shown in Figure 17.

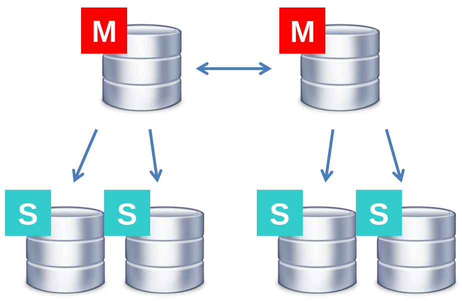

Figure 17 Dual-tree replication model

4.3.3 Replication Delay and Consistency

Replication delay is the time it takes for data to be replicated to a machine. Replication delays across all machines will vary. A faster slave machine, or one serving less threads for example, would be able to update its data faster than a slower machine. Hence, there will be a time during replication when data is not the same (in other words, not consistent) across all the database replicas. Data inconsistency may also be attributed to a replication model: in the master-slave model, the master machine always has the most updated copy of the data at any time. This is however untrue for any machine in the multi-master model (assuming there is traffic), since different writes would happen at the same time to multiple masters. If we need consistent reading, then we could enforce a synchronous replication (where writes are replicated to all machines) before performing a read.

4.4 Scaling out using partitioning

With a relational database, there are two main ways of partitioning data, namely vertical and horizontal partitioning, and both may be used to achieve either Y- or Z-axes scaling. The difference between the axes scaling is that Z-axis data splitting requires a lookup on request, while the Y-axis does not. Unlike replication where each database server stores exact data replicas, data partitioning involves multiple database servers (or instances) carrying different datasets. Thus, data partitioning is able to scale out writes.

4.4.1 Vertical Partitioning

When performed on a database, vertical partitioning results in clusters of databases with each cluster containing a subset of tables from the original database. These clusters may reside in physically separated servers, or could simply be different database instances within the same machine.

Tables within a cluster are not allowed to join to with tables in a different cluster. Thus, joined tables are normally placed in the same cluster. Otherwise, we may also change our application logic to avoid making the join. With this join restriction, tables involved in the same application function are usually placed in the same cluster. The converse may not be true.

If we start with large tables, it may be necessary to normalize them prior to horizontal partitioning. Data normalization is a specific way to achieve the more general concept of horizontal partitioning.

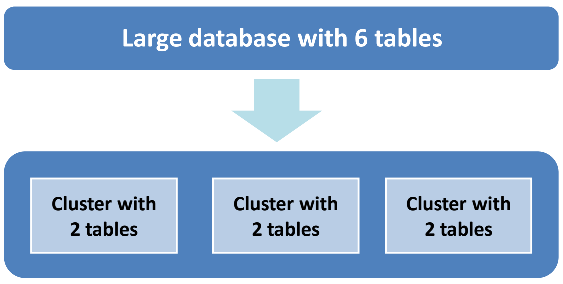

Figure 18 Partitioning data into clusters

4.4.2 Horizontal Partitioning

Horizontal partitioning splits a database table by rows. These subsets of data are then distributed to different database servers, which we refer to as shards[3]. The attributes on which we choose to perform sharding depends on the application. For example, data may be sharded according to customer locations; the resulting shards may now be physically placed to decrease network delays.

When a request reaches the application, a form of look up mechanism is needed to determine the shard from which data should be retrieved.

Horizontal partitioning should be achieved such that there is no need to fetch, merge or sort data across shards. Data splitting thus has to be performed carefully on related tables so that all records that need to be fetched together are stored in the same shard.

[3] Other literature or sources may have different definitions for the term “shard”.

ID	FirstName	LastName	Faculty
1	John	Carter	FASS
2	Allen	Tan	SoC
3	Bob	Markzinski	SoC
4	Susan	Roberts	Biz
5	Julia	Lyline	FASS
6	Mark	Yankze	SoC
7	Twain	Shernie	Biz
8	Shania	Klow	Biz
9	Susan	Boyle	SoC
10	Bob	Goranski	FASS

ID	FirstName	LastName	Faculty
1	John	Carter	FASS
5	Julia	Lyline	FASS
10	Bob	Goranski	FASS

ID	FirstName	LastName	Faculty
2	Allen	Tan	SoC
3	Bob	Markzinski	SoC
6	Mark	Yankze	SoC
9	Susan	Boyle	SoC

ID	FirstName	LastName	Faculty
4	Susan	Roberts	Biz
7	Twain	Shernie	Biz
8	Shania	Klow	Biz

Figure 19 Original table (left) and shards partitioned by faculty (right)

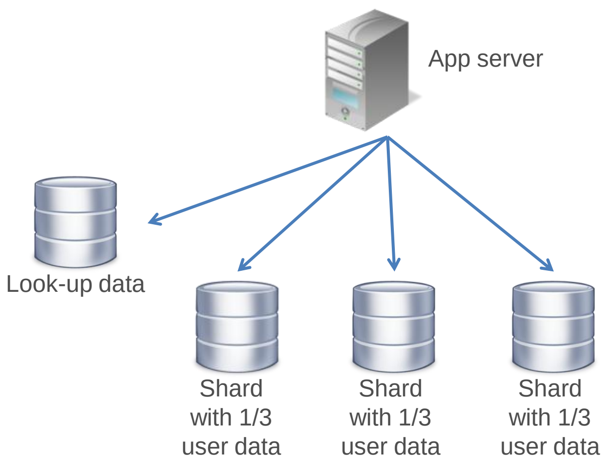

Figure 20 Sharded data with lookup

5 SCALING WEBSERVER AND FILE STORAGE

The term “web server” (or “webserver”) refers to a computer put in the network to handle web requests (HTTP requests) made by other computers. Any computer that connects to the network can be set up as a web server. A web server usually runs special software called the “web server software” that handles HTTP requests.

Nowadays, web servers are often required to have very high processing power (multiple CPU chips and high amount of RAM) to cope with high amount of user requests. Also, web servers often have their input/output devices (such as keyboards, monitors) “trimmed off” so that they can be fit nicely as boxes into racks / clusters in order to save space should the total amount of web servers is high and space is precious.

Figure 21 A server rack (Server Rack n.d.) and a server cluster(Server Cluster n.d.)

A file storage server is very similar to a web server. However, their main purpose is in data storing and not request handling. Similar to web servers, file storage servers often have unnecessary hardware trimmed off to ramp up other critical parts such as CPUs and RAM in the case of web server.

Figure 22 A typical web server(Google Web Server n.d.)

INFO The most common web server software is Apache, currently being used by more than 60% of the Internet web servers (March 2011 Web Server Survey n.d.).

5.1 Load Balancing

Load Balancing is a scaling out technique that can be applied to a cluster of computers. A load balancer will accept requests from users and then direct them to the right web server within a cluster. The "right" server here is decided by certain criteria, mainly regarded as the load balancing strategy. There are many strategies, common ones are:

- *Round Robin*: each server takes turn to receive requests. This is the simplest strategy, similar in spirit to First In First Out applied in caching.
- *Least number of connections*: the server with the lowest number of connections will be directed the request. This is an attempt to prevent high load.
- *Fastest response time*: the server that has the fastest response time (either recently or frequently) will be directed the request. This is an attempt to handle requests as fast as possible.
- *Weighted*: this strategy can be highly customized since the weightage can be configured. This strategy takes into account the scenario where servers in the cluster may not have the same capacity (processing power, storage, etc.). Thus, the more powerful servers will receive more requests than the weaker ones under weighted strategy.

A common misconception is that load balancing can only be applied on a cluster of web servers since only web servers are going to accept requests and process them. In actual fact, dedicated web servers usually do not have the required data with them to process requests and often send out these requests to dedicated database/file storage servers. This layer of requests can also be load balanced.

There are both hardware and software solutions for implementing load balancing. Hardware load balancers are usually the dominant choice. They can be very fast and extremely powerful, and capable of processing more than 85000 requests per second with a throughput of approximately 10Gbps. These hardware load balancers are however, also very costly. There are cheaper load balancers, alternatively, we may also use software load balancers (such as nginx, Linux Virtual Server, etc.). In the case of software load balancers, there are usually code configurations that we have to specify in order for the software to behave like a load balancer and how they should redirect requests.

There is a limit to what vertical scale can achieve for popular services such as Google, Facebook, Amazon, Flickr and Youtube. These systems need to scale out and be load balanced. Thus, load balancing has now become a must in almost any web service architecture, playing an important part in ensuring availability and scalability of a system.

Besides balancing load, a load balancer also tries to achieve other important qualities, such as:

- *Server cluster awareness*
    - A load balancer should be well aware of its server cluster.
    - Should a server breaks down in the cluster, or a new server joins the cluster, the load balancer should have appropriate actions (stop sending requests to the failed server and/or continue distributing requests to the new server) to keep request routing as undisrupted as possible.
- *Load balancing performance*
    - A load balancer should not just try to even the load, it should also try to do so well. This means that it should be fast in deciding the target server to direct requests to.
    - This is important, for service response time is very significant to users. Any extra overhead added to response time will render the service much less attractive to others with better response time.
- *Service resilience*
    - The awareness and performance qualities above subsequently ensure resilience and reliability of the entire system.

5.2 Webserver Software Optimization

As mentioned, a web server is just a computer running a combination of software to support handling of web requests. Web server optimization simply means optimizing the software running on the server.

Apache is popular in the industry, mainly because it is stable and supports many features, but that does not mean that these features operate optimally. When it comes to serving static content (images, videos, etc.), Apache might not be the best choice. nginx (pronounced: “engine x”) is a light-weight web server software that is better at handling static content than Apache.

INFO APACHE & NGINX REQUEST HANDLING

To understand why nginx would be better at handling static content than Apache, it is important to know how Apache and nginx handle requests.

Apache runs a multi-process (prework) / multi-threaded (worker) model. It has a master process running in the background listening to requests. When a new request arrives, the master process spawns (or forks) a new a sub-process or assigns a new worker thread to handle the new request. When the number of requests increases, the cost for context-switching between threads/processes becomes a burden to the server. Furthermore, Apache pre-loads mod_php (or mod_python etc, depending on the dynamic language in use) in memory to boost performance when serving dynamic content. If the request is for static content, then this memory usage is rather redundant.

nginx, on the other hand, uses a single-thread event-driven model. It has one master process and a few worker threads (two or three). The master process actively listens to requests. An incoming request (assuming for static content) is passed to the worker thread. The worker thread parses the request and then asks the input/output (I/O) socket to fill in data. When I/O is working the worker thread continues to handle other requests. When I/O finishes filling in data, it notifies the process/worker to come and fetch the data and return it to the client. This way, the memory overhead is minimal and it can serve substantially more concurrent requests compared to Apache.

5.3 Caching

Caching is a data processing and data access optimization technique. It increases these performances by storing results of computations or copies of original data in a data storage location, usually the memory or CPU caches, as these locations usually have faster access speed than the storage in which the data originally resides, such as the disk. Thus, future requests that ask for the same data can then be served faster by just accessing the cache (in case of a cache hit). However, since cache spaces are usually much smaller than the original data storage, cache misses are unavoidable. , there exist *cache strategies* to help increase cache hit rate. There are also *cache policies* to decide what the system should do in case the data in the cache is modified.

DEFINITION ***Cache strategies*** are algorithms that increase cache hit rates, while ***cache policies*** are rules used to decide a system's actions when data in the cache has to be modified.

There are many cache strategies. They can perform very differently but share a general feature: they estimate the time it takes for data in the cache to be accessed again in the future. There are several common cache strategies, such as:

Belady/The clairvoyant algorithm: In the case of a cache miss or when the cache is full, this strategy tells the system to discard information that will not be needed in the furthest future. This algorithm is practically impossible to implement and is entirely theoretical. Information on how far in the future will a data piece be accessed is impossible to accurately and precisely calculate. Other cache strategies try to come close to this, by giving estimations of that information in different ways.

Least Recently Used: LRU has been a standard for caching for quite a long time. LRU simply –discards information that is least recently accessed in the cache. It does this by maintaining an age bit. Data with the biggest (oldest) age bit will be discarded should a cache miss occurs.

Least Frequently Used: LFU is an improved version of LRU. Instead of maintain an age bit, LFU maintains an access count instead. Data with the lowest access count will be discarded.

Adaptive Replacement Cache: ARC is a combination and improved version of both LFU and LRU. One might not realize that LFU and LRU methods do not overlap. A piece of information that is recently accessed is not necessarily frequently accessed. Thus, they can actually be combined to further increase cache hit rate. ARC does the job by combining these two. However, ARC takes one step further by maintaining this information not just for data that currently in the cache but also for data that used to be in the cache.

Cache policies are usually referred to as write policies, since read policies rarely differentiate cache systems. It is the write policies that impact cache systems differences:

Write-through: When data in the cache is changed, the change is synchronously reflected on the original storage, usually the disk.

Write-back (lazy write): A write is only done to the disk when "dirty" information is discarded from the cache. Dirty information is information that has been written over by some operation, or in simple words, has already been changed. This is maintained by a dirty bit attached to cached data.

No-write-allocation: Cache systems under this policy only cache read operations results. Writes are written directly to the disk.

The main reason that makes caching works is *locality of references*. One would think that keeping old calculated results will never do any good. In the case of computer operations, it is the exact opposite. Computer operations usually exhibit locality of references, meaning that they frequently access data that was accessed in the past. Locality of references comprises the following cases, which happen frequently in computer programs:

Temporal locality: Data that was accessed is likely to be accessed again the future. This is easily exhibited by multiple threads trying to access/modify a shared data.

Spatial locality: If data was accessed, nearby data is likely to be accessed in the future. This is easily exhibited by a loop modifying values in an array.

Equidistant locality: If data was accessed, data that are away from it in an equidistant manner is likely be accessed in the future. This is easily exhibited by a loop modifying an array in an equidistant manner (say, jumping two units each time instead of one).

Branch locality: If data was accessed, its "branch" data locations are likely to be accessed in the future. This is easily exhibited by if-else, switch, and finally statements within a loop.

Caching is said to *exploit* locality of references. In contrast, other optimizing techniques may try to increase locality of references. These techniques are usually done in software code, where data is deliberately arranged so that they are close together.

5.4 Distributed caching

This section is summarized from (Distributed Caching on the Path to Scalability n.d.).

Caching is a very familiar concept, implemented in both hardware and software. Traditionally, caching has been a stand-alone mechanism, but that is no longer viable for applications that now run on multiple servers and in multiple processes within each server.

In-memory distributed caching is a form of caching. It allows the cache to span multiple servers so that it can logically grow in size and in transactional capacity. Distributed caching has become feasible now for a number of reasons:

Cheap memory enables computers to be equipped with gigabytes of storage.

Proliferation of fast network cards.

Distributed caching works well on lower cost machines, which we can easily add.

Distributed caching is a scalable structure. It distributes work across multiple servers but maintains a logical view of a single cache.

Memcached

Memcached is a distributed *memory caching* system. It caches data, objects and especially database query results in RAM (hence the name "memory caching"). Memcached was born with the sole purpose of alleviating database load.

Like any other distributed caching system, Memcached is a scalable solution for the more servers you add to the cluster, the larger the logically combined cache space will be. This further increases cache performance.

Before Memcached, each individual server node has its own cache memory and is unaware of cache memories on other servers. When request for data with key X comes in to server A, server A calculates and caches data in memory. But if the same request is directed (through the load-balancer) to server B, B has to calculate the data again and cache it. This results in both a "cache miss" and wasted memory space (since both A and B need to store the data in their own memory spaces).

Memcached combines the servers' memory spaces together, resulting in a single huge cache memory space. When a request for data with key X comes to any server (say A), the request is transferred to right server to handle this request, determined by the global hash function hash_choose_server(key) returning the id of the server. When it comes to the destined server, normal caching mechanism takes place and transfers the response back to A.

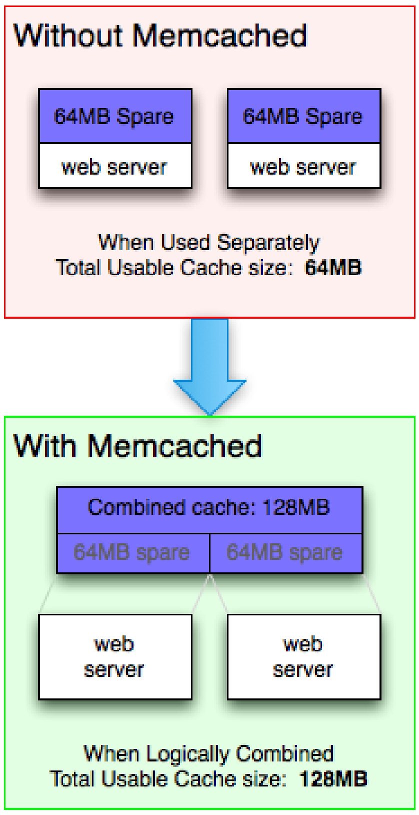

Figure 23 How Memcached works(Memcached Diagram n.d.)

This logically combined cache eliminates a lot of problems such as:

Limited cache space on each server. Without distributed caching, all the memory on a system will act as separate cache spaces to each server they belong to. They cannot share anything between each other nor can they access data in the other. This is limits cache space. The typical size of the memory is 4GB for 32-bit systems. However, in large systems data can be up to 100GB or even more. Thus, the cache hit rate is extremely low.

Memory space wastage. With a non-distributed caching system, when the same request is handled by different servers, the system would have cached the same data in different cache storages on those servers. This leads to duplicates of data, which is a wasteful way of using memory.

5.5 Content Distribution Network (CDN)

Content Distribution Network (or Content Delivery Network, CDN) is an optimization technique which helps to:

- increase data access performance for users that are far away from the central server(s)
- avoid bottleneck at the central server(s)

CDN is a system of computers that possess copies of static data from the central server(s). These computers are spread geographically everywhere on the network so as to cover as many regions as possible, thereby reaching out to users that were originally very far away from the service.

Traditionally, the central server handles both static and dynamic data. With the CDN, it will now only respond to clients with a very basic web page with content served from itself. The rest of the web page components, usually images and static data will be served from servers within the CDN instead. This greatly offloads the web servers with processing and storage of static content, leaving them to serve only dynamic content.

In practice, CDN is not a stand-alone technique. It often, if not always, integrates a lot of other scaling techniques to realize the concept and further improve its performance. These techniques are:

- Caching, usually at the web server layer i.e. web caching
- Load balancing, usually hardware load balancer for superior performance
- Request routing, a technique is actually used by load balancing

INFO CLOUDFLARE (www.cloudflare.com)
CDN services like Amazon S3 or Akamai may require changes in the application code e.g. changing the URL of the image in responding HTML, or writing code to transfer uploaded file to Akamai services. Cloudflare offers an alternative CDN service that is suitable for small scale web applications. To use it, the user need only point his domain to Cloudflare's nameserver.

HOW CLOUDFLARE WORKS
Suppose our web site, mysite.com, is hosted on a webserver at the IP address 1.1.1.1.

Originally, when someone types in "mysite.com" in the web browser, it will make a DNS lookup (to our ISP provider) to discover the IP address of "mysite.com". It then directs subsequent requests to that IP address (in this case, 1.1.1.1).

When we point our domain to Cloudflare's nameserver, DNS lookup is now handled by Cloudflare's DNS server. Cloudflare returns the IP address of a data center that is geographically nearest to the visitor (say 9.9.9.9). From now on, subsequent requests from visitors will go to 9.9.9.9.

When a request arrives at 9.9.9.9, the Cloudflare's frontline servers check if the resource is in the local cache. Cloudflare caches parts of websites that are static e.g. images, CSS, and Javascript. Utilizing CDN technology, these static resources are delivered from the data center nearer to the visitor.

If the request is for a type of resource that Cloudflare does not cache, or if a current copy is not available, a request is made from the data center (9.9.9.9) back to the original server (1.1.1.1). Because it uses dedicated premium routes, data passage via Cloudflare may have less hops than the route it would normally go travel.

6 Load Testing & System Monitoring

6.1 Load testing

Load testing is the process of putting a lot of demand on a system and measuring its response. This is to observe and study the system's behavior under peak conditions. When the demand is known to exceed the system's capacity, the term *stress testing* is used instead. (Wikipedia: Scalability n.d.)

Specifically, in the domain of web scalability, the concept of load testing is simple: simulate large amounts of concurrent HTTP requests to the destined system (Figure 24).

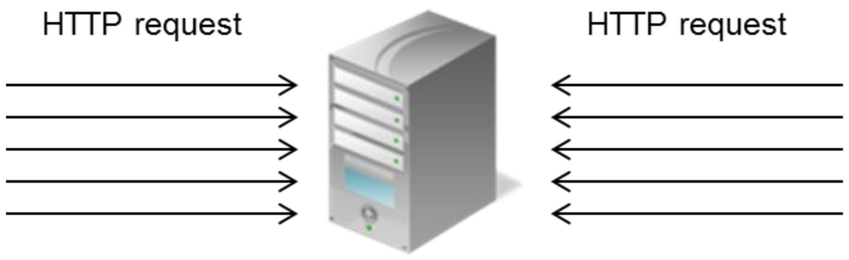

Figure 24 Load testing mechanism

There are a lot of load testing tools available, ranging from very simple (load test one URL) to advanced (allow programming for virtual users). The table below for a lists several existng tools:

Tools	Note
Pylot	Easy to use. Simple. Generate response graphs.
Apache JMeter	Allows remote-control of many different nodes. Steep learning curve. Has GUI, simplifying configurations.
PHPTestSuite	PHP-based MySQL load generation. Write tables and queries in PHP.
Multi-mechanize	Advanced version of Pylot. Allows developers to use Python to write automated interaction with websites (virtual web users).

Table 4 Load testing tools

6.2 System monitoring

System monitoring refers to the process of observing the statuses of different components of the system. Table 5 lists common monitoring tools.

Tools / Services	Note
JetProfiler	MySQL database monitoring tool. Easy to use. Shows real-time graphs. Free and cross-platform. Paid version has more features.
Nagios	Free tool. Popular enterprise tool for large system. Supports many features, graphs supported.
Munin	Monitors and draws graphs depicting system performances on daily/weekly/yearly basis.
ServerDensity	Paid service, monitors server through web interface. Installs extra daemon on server. Support many common software. Has plugins.
BrowserMob	Similar to ServerDensity. Also supports load testing (using real browser requests instead of simulated request)

Table 5 System monitoring tools

7 Advanced Topic: Logical Volume Management

Logical Volume Management (LVM) is a method of logically arranging and allocating storage space on a group of mass-storage devices (usually group of disks with large capacity) so that the *logical disk space* appears as different volumes. More advanced compared to the conventional disk partitioning technique, where one can only split/“concatenate” partitions from within a physical disk, LVM allows partitions from any disks be grouped together to form logical/virtual disks that can later be managed normally as if they are physical disks. Figure 25 illustrates this concept.

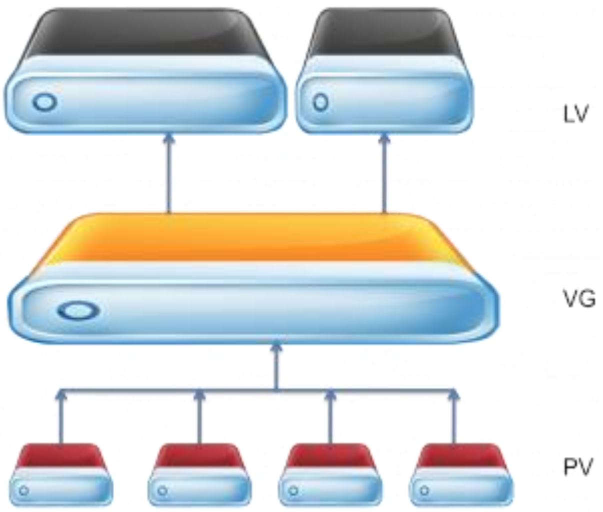

Figure 25 Logical volume management (LVM Diagram n.d.)
LV = Logical Volume, VG = Volume Group, PV = Physical Volume

As we can see, there are actually four physical volumes in this system. However, with LVM, we can group them into just two virtual volumes. In detail, the first two physical volumes and some parts of the third form the first logical volume. The rest of the third physical volume and the fourth form the second logical volume.

Visually and conceptually, LVM can "concatenate" all the physical storage into one big chunk. Then, it can split this chunk in whatever manner we need to. This brings great flexibility to the size of the virtual disks. This, however, might not be the exact way that certain systems perform LVM. For example, Linux does it by combining disk partitions.

In Linux, partitioned physical disks result in a pool of disk partitions. To be more flexible, these partition pools may not have the same sizes. In order to create a logical volume, the Linux LVM will then "pick" certain partitions and logically combine them. To further improve flexibility, Linux LVM will not require these partitions to be contiguous.

In Linux LVM, it is very easy to "shrink" a volume by releasing one of its partitions back to the pool (see Figure 26)

Normally, LVMs are not features implemented by the web developer. They are often pre-incorporated inside the file or operating systems either as a hardware or software component.

LVM, or in more general view, storage virtualization, helps manage large and complex storages such as that found in web applications. Without LVM, the combination of small physical disks into large-sized volumes would not be possible, limiting the users to a low storage capability. In the scalability context, LVM helps tremendously in combining newly added disk storages as well as managing them. It eliminates the need to move data around in order for them to be contiguous. It also allows large data to span multiple disk spaces – this would not be possible with un-combined small disks.

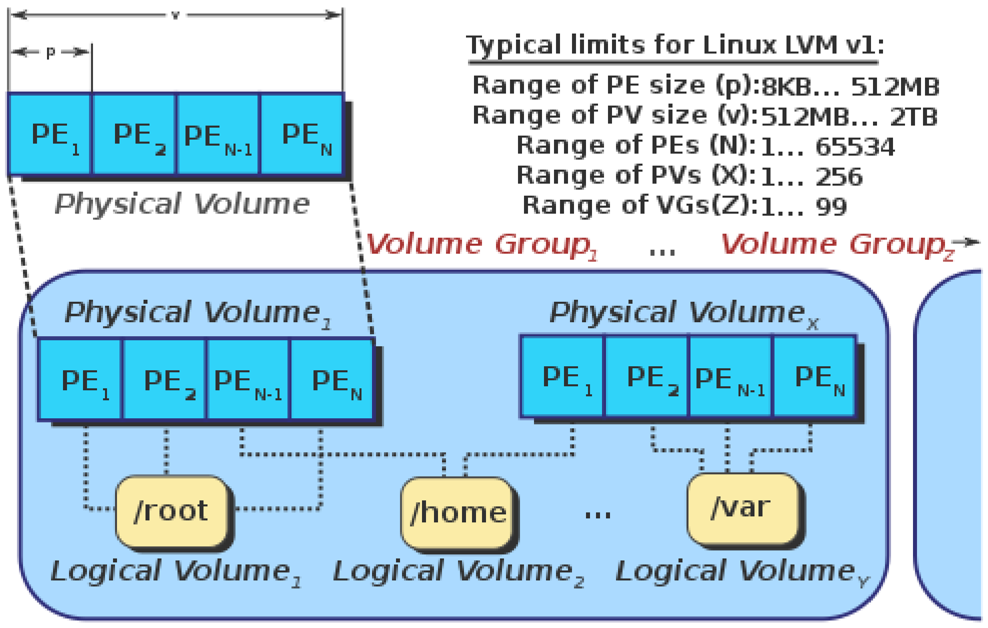
v
p
PE1 PE2 PEN-1 PEN
Physical Volume
Typical limits for Linux LVM v1:
Range of PE size (p):8KB... 512MB
Range of PV size (v):512MB... 2TB
Range of PEs (N):1... 65534
Range of PVs (X):1... 256
Range of VGs(Z):1... 99
Volume Group1 ... Volume GroupZ
Physical Volume1
Physical VolumeX
/root
/home
...
/var
Logical Volume1 Logical Volume2 Logical VolumeY

Figure 26 LVM in Linux (Linux LVM n.d.)
PE = Physical Extents i.e. equivalent to physical disk partitions

While one might not be able to fully control LVM since it is incorporated inside the filesystem and/or operating systems, understanding how it organizes disk spaces would still help greatly towards writing scalable programs and designing scalable web architecture.

8 Advanced Topic: Database Choices for Scalable Web Applications

The relational database model has been widely used in systems since its introduction in 1970 (Codd 1970). Lately, new storage systems, termed *NoSQL*, have gained popularity especially in web applications. NoSQL or Not Only SQL, is a concept of distributed data storage that differ from the relational data model[4]. These databases are known for their ease in horizontal scaling, fast indexes on data, high availability, and in some cases, Javascript-like APIs that is more favourable to web programmers.

The choice of database depends on both functional and non-functional requirements of a web application. An application that prioritizes *ACID transactions* over availability would want a database that supports those needs than an application with contrary requirements.

This sub-section introduces the general features of relational and NoSQL databases and ends with simple use cases that demonstrate the selection of a fitting database.

DEFINITION A database ***transaction*** is a unit of work that the database performs. Normally, a transaction is required to be performed in its entirety or not at all. For example, the transfer of funds from one account to another is a transaction comprising a debit of one account and a credit of the receiving account. The ACID properties guarantee that a database transaction is:

Atomic: Either the entire transaction succeeds or should be rolled back if not.
Consistent: A transaction will never leave the database in an inconsistent state.
Isolated: A transaction is independent of, and does not interfere with another.
Durable: A successful transaction persists against any system failure i.e. the database should

[4] Despite its suggestive name, "NoSQL" is not anti-relational, but rather refers to non-relational databases.

be able to recover it.

In this book chapter, a transaction will infer an ACID transaction unless otherwise stated.

8.1 Relational Database

In a relational database each data record is stored as a row in a table. A record has attributes (corresponding to the table columns) which are pre-specified in a schema. Hence, all data is structured in a relational database. Relational databases use the standard Structured Query Language (SQL) as its "application programming interface (API)". The web developer interfaces with the database via SQL bindings in languages like Java and PHP.

When we query the database for information, it retrieves data by returning a subset of a table or joins of tables. This process can be expensive with large tables and complex join queries, but can be mitigated with sharding. However, in a large-scale application, even the sharding process is complex (as mentioned in 4.4.2 Horizontal Partitioning, related tables need to split as well) and has to be carefully designed. Another way of lessening joins is by denormalizing data, which goes against the advocated principle of normalization. While normalization enforces data integrity by preventing duplicate data, denormalization prevents expensive joins at the price of having duplicate data and rests the responsibility of ensuring integrity on the developer.

Relational databases are well known for enforcing ACID transactions. It is due to such data integrity and reliability – and the fact of being tried and tested for over 30 years – that it still remains a credible choice of database.

8.2 NoSQL Databases

Unlike relational databases, there is no standard query language for NoSQL databases. Instead, the interface to these databases varies by providers.

There seems to be many categories of NoSQL databases. This book chapter discusses three of them namely key-value, document and column databases. Discussions in the following sub-sections are summarized from *High Performance Scalable Data Stores* (Cattell 2010).

8.2.1 Key-Value Database

Unlike the concept of a row in a table as with relational database, a record in the key-value database is a pair of key and value. The key is programmer-specified and the value may be scalar, binary large objects (BLOBs), or types such as lists and sets (depending on the database provider).

In a scaled out system, records will be distributed over nodes by hashing the key. A query for a range of values will thus need to go to every node. Some providers like Scalaris allow nodes to hold ranges of keys, avoiding visits to every node and providing for better load balancing.

Key-value databases, similar to the memcached distributed in-memory cache, may store data in memory, but also provides persistence mechanisms. As expected of most distributed databases, other functionality includes replication, versioning, concurrency control, transactions and sorting. The table below lists key-value databases and their functionalities.

Databases	Concurrency	Data storage	Replication[5]	Transactions[6]

[5] Replication may be either synchronous, in which all replica are always in sync, or are updated asynchronously.
[6] N = not supported. L = transactions limited to a single node or record.

	control			
Redis	Locks	RAM	Async	N
Scalaris	Locks	RAM	Sync	L
Tokyo Tyrant	*MVCC*	RAM / disk	Async	L
Voldemort	Locks	RAM / BDB	Async	N

Table 6 Key-value databases

DEFINITION Multi-version concurrency control, or ***MVCC***, is a method of concurrency control in which versions of data are created, as needed, for transactions accessing the same data.

Currently, key-value databases are not as popular as its document or column counterparts.

8.2.2 Document Database

A record in a document database is stored as a document[7], which is a set of attribute-value pairs. Unlike key-value databases, it supports multiple indexes and varying types of documents per database. The values may be scalar and may also be nested documents.

The document database is schema-less i.e. the structure of records need not be pre-defined as with relational database. Attributes are created when specified at runtime. Some databases may use the term "collection" to refer to a group of documents. It may be necessary to define a schema on collection indexes.

EXAMPLE
In databases like MongoDB, the documents can be specified with JSON e.g.

```
{ "name": "John Doe", "age": "23" }
{ "name": "Jane Doe", "age": "24", "city": "Dublin" }
{
   "name": "Mary Jane",
   "occupation": "stage actor",
   "spouse": { "name": "Peter Parker", "occupation": "photographer" }
}
```

The records above show that data can be semi-structured in a document database. Although both records refer to a similar type of data e.g. personal details, the *city* attribute is not enforced upon a record that does not have any value for it.

Since web applications constantly need data in the form of a document e.g. XML or JSON, the interface of document databases are well suited to web application development. Its schema-less feature is also a draw for web applications with semi-structured data.

Databases	Concurrency control	Data storage	Replication	Transactions

[7] Although commonly described as "storing documents", this description is not accurate. The system really stores objects that are described in a document notation such as JSON.

SimpleDB (Amazon)	None	Amazon's S3	Async	N
CouchDB	MVCC	Disk	Async	N
MongoDB	Field-level	Disks	Async	N

Table 7 Document databases

The example document databases offer neither ACID transactions nor synchronous replication. Instead of the consistency definition in ACID, they offer *eventually consistent* data and sacrifice transactions for high availability, performance, and scalability. Concurrency and atomicity properties are also weaker than ACID compliant databases.

To support scalability, databases like MongoDB provide automatic sharding to scale writes.

DEFINITION Unlike the ACID guarantee, ***eventual consistency*** promises data consistency at some point eventually, but until then, it is possible that a user accesses stale and inconsistent data due to yet uncommitted data changes within a transaction. There is no guarantee for data recovery if a failure were to occur during a transaction.

8.2.3 Column database

In a relational database, each record is stored contiguously on memory/disk by row. Data retrieval is fast when the database knows exactly which row to retrieve. However, in applications that require many field (column) comparisons, the overhead in disk seeks can slow down reads.

Column databases differ from relational databases in that the data storage is organized by column as opposed to row. This reduces amount of read data, especially in large data warehouses such as those at Google.

id	tweet	user	created
1	It's 4am now	20	2011-03-23
2	Ate a burrito	6	2011-02-05
3	Beer on sale	93	2011-03-29

In a row-based storage

```
1, It's 4am now, 20, 2011-03-23,
2, Ate a burrito, 6, 2011-02-05,
3, Beer on sale, 93, 2011-03-29
```

In a column-based storage

```
1, 2, 3,
It's 4am now, Ate a burrito, Beer on sale,
20, 6, 93,
2011-03-23, 2011-02-05, 2011-03-29
```

Figure 27 Data table and corresponding storage in row- and column-based databases

EXAMPLE

The query below requires a relational database to compare the *user_id* field value in each *tweet_table* row and retrieve all matching rows. It then returns only the values in the *tweet* column of the row subset.

```
SELECT tweet FROM tweet_table WHERE user_id = 93
```

In a table with many columns, the total amount of data read (e.g. row subsets in *tweet_table*) can amount to much more than the size of data retrieved (e.g. *tweet* values).

This is not so in a column-oriented database. Once comparisons are done in the *user_id* column, the database will now have row ID/number corresponding to values in the *tweet* column. It may then do direct reads from there.

Column databases can be scaled out by distributing columns over multiple nodes. When sharding is applied, the rows are normally split by range rather than hashing, a strategy common to optimize read efficiency in a distributed system. For example, when Google uses their Bigtable column-based database to store web pages, they index each row by reversed domain-names, so that pages in the same domain are stored near each other, which in turn makes host and domain analyses more efficient (Fay Chang 2006). Bigtable also introduces the concept of column families, which basically group several related columns. These column families may also be distributed over nodes. Bigtable controls concurrency by storing time-stamped copies of data in each table cell; old copies will be garbage-collected. The table below compares more similar databases.

Databases	Concurrency control	Data storage	Replication	Transactions
Bigtable (Google)	Locks, timestamps	Google's GFS	Sync (locally), Async (remotely)	L
HBase	Locks	Hadoop	Async	L
Cassandra	MVCC	Disk	Async	L

Table 8 Column-oriented databases

8.3 Use Cases

Factors to consider when selecting a database include support for ACID transactions, level of concurrency control, ease of scaling e.g. data partitioning and availability, among others. According to *Brewster's CAP theorem*, it is not possible to cater to all factors. The following are simple examples showing selection of each database for specific reasons.

DEFINITION ***Brewster's CAP theorem*** states that at most two of the three CAP properties can be met in a distributed computer system. The CAP properties are:

Consistency: The system is always consistent (it operates fully or not at all)
Availability: The service is always available (it operates fully or not at all)
Partition-tolerance: The system still operates despite message loss i.e. it responds correctly even when there are failures in the network, unless the failure is a total network failure)

8.3.1 Key-Value Database Use Case

Key-value database can be utilized to store data that needs to be quickly accessible by unique identification. For example, personalized homepages can be stored as values accessible by user IDs. When the user logs in, the homepage is read directly from the key-value database rather than being dynamically generated with select and join relational database queries. Each (ID, homepage) record is updated only when changes are made to user data.

8.3.2 Document Database Use Case

Most document databases use eventual consistency, but offers higher performance and availability. In an electronic store with large customer base and frequent user visits, it is not feasible to slow down the system with frequent usage of concurrency locks. Hence, the web application may prioritize customer accessibility and experience over concurrency handling. It thus chooses to risk cases where two customers make the same purchase for the last item in the store, and may need to personally contact the customers when it happens.

Due to eventual consistency, document database is not suitable for data that need to be accurate at all times e.g. it may not be suitable for storing a customer's electronic wallet data.

Document databases are also useful for web applications that use many semi-structured objects, or need to create and store objects dynamically.

8.3.3 Column Database Use Case

Mostly, column databases are used in cases when higher throughput is necessary and when horizontal and vertical data partitioning is needed. Existing databases such as HBase and HyperTable support these well. Column databases also provide better transaction guarantees than document databases.

9 Advanced Topic: State Management

The protocol used by web browsers, HTTP, is a stateless protocol. The web server, by default, has no knowledge of whether current incoming requests are sent by the same client who sent the previous ones. Most real-life web applications however, need the web server to "remember" this information. For example, a user visiting a shopping site, would click on a link to a checkout page to pay for the items she chose at the previous page. This requires the server to "remember" who has logged in and associate the user with the items ordered, in other words, it requires a stateful web server. To support such stateful requests, the concept of *session* is introduced to allow a web server to have semi-permanent information about individual visitors.

In the popular web server software Apache, when a client C first sends a request to web server A, A stores the session on the server, to be accessed later through a session ID (also generated by Apache). The client is then informed of the session ID. The next request made by the same client will automatically carry that session ID with it. This way, the server will know a lot more information about a client if it has visited the site before.

Usually the web server stores session information in a directory on server. However, in a scaled out system, it is usually the case that the second request coming from the same client may land in a different web server, say B, instead of A (this is due to load balancing). Since this is the first time C makes a request to B, C is unable to retrieve the session information associated to that user.

Clearly we see where the problem comes from. A and B have their own individual session storage and do not know about prior client requests to the other server.

The solution is to have both of them share the same session storage. Session data in its nature is short-lived and not permanent. Some of the storage choices we could pick are:

- Use Memcached
- Use database table
- Use NFS (network file system)

Memcached is usually used since it is the fastest method among the three. The second method entails extra load on the database server and may not be desirable. NFS is rarely used because it is slow.

10 References

Bass, Lens, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Pearson, 2003.

Browne, Julian. *Brewster's CAP Theorem.* 11 January, 2009. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem (accessed 17 April, 2011).

Cattell, Rick. "High Performance Scalable Data stores." 22 feb, 2010.

Codd, E. F. "A relational model of data for large shared data banks." *Communications of the ACM*, June 1970: 377-387.

Distributed Caching on the Path to Scalability. http://msdn.microsoft.com/en-us/magazine/dd942840.aspx.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data." *Google Research Publications.* 2006. http://labs.google.com/papers/bigtable-osdi06.pdf (accessed 16 April, 2011).

"Google Web Server." *http://www.seoconsult.com/seoblog/google-and-search-engine-optimisation/.*

Henderson, Cal. *Building Scalable Web Sites.* Sebastopol, CA: O'Reilly, 2006.

Introduction to Database Indexes. http://www.interspire.com/content/2006/02/15/introduction-to-database-indexes/ (accessed 20 March, 2011).

"Linux LVM." *http://en.wikipedia.org/wiki/Logical_volume_management.*

"LVM Diagram." *http://highscalability.com/blog/2010/8/16/scaling-an-aws-infrastructure-tools-and-patterns.html.*

March 2011 Web Server Survey. http://news.netcraft.com/archives/2011/03/09/march-2011-web-server-survey.html.

Martin Abbott, Michael Fisher. *The Art of Scalability: Scalable Web Architecture, Processes and Organizations for the Modern Enterprise.* Addison-Wesley, 2010.

"Memcached Diagram." *memcached.org/about.*

Offutt, Jeff. "Quality Attributes of Web Software Applications." *IEEE Software* 19, no. 22 (Mar/Apr 2002): 25-32.

"Server Cluster." *http://www.skullbox.net/clusterpart2.php.*

"Server Rack." *http://en.wikipedia.org/wiki/Server_(computing).*

Tian, Jeff. *Software Quality Engineering.* John Wiley and Sons, 2005.

Wikipedia: Scalability. http://en.wikipedia.org/wiki/Scalability (accessed 15 March, 2011).