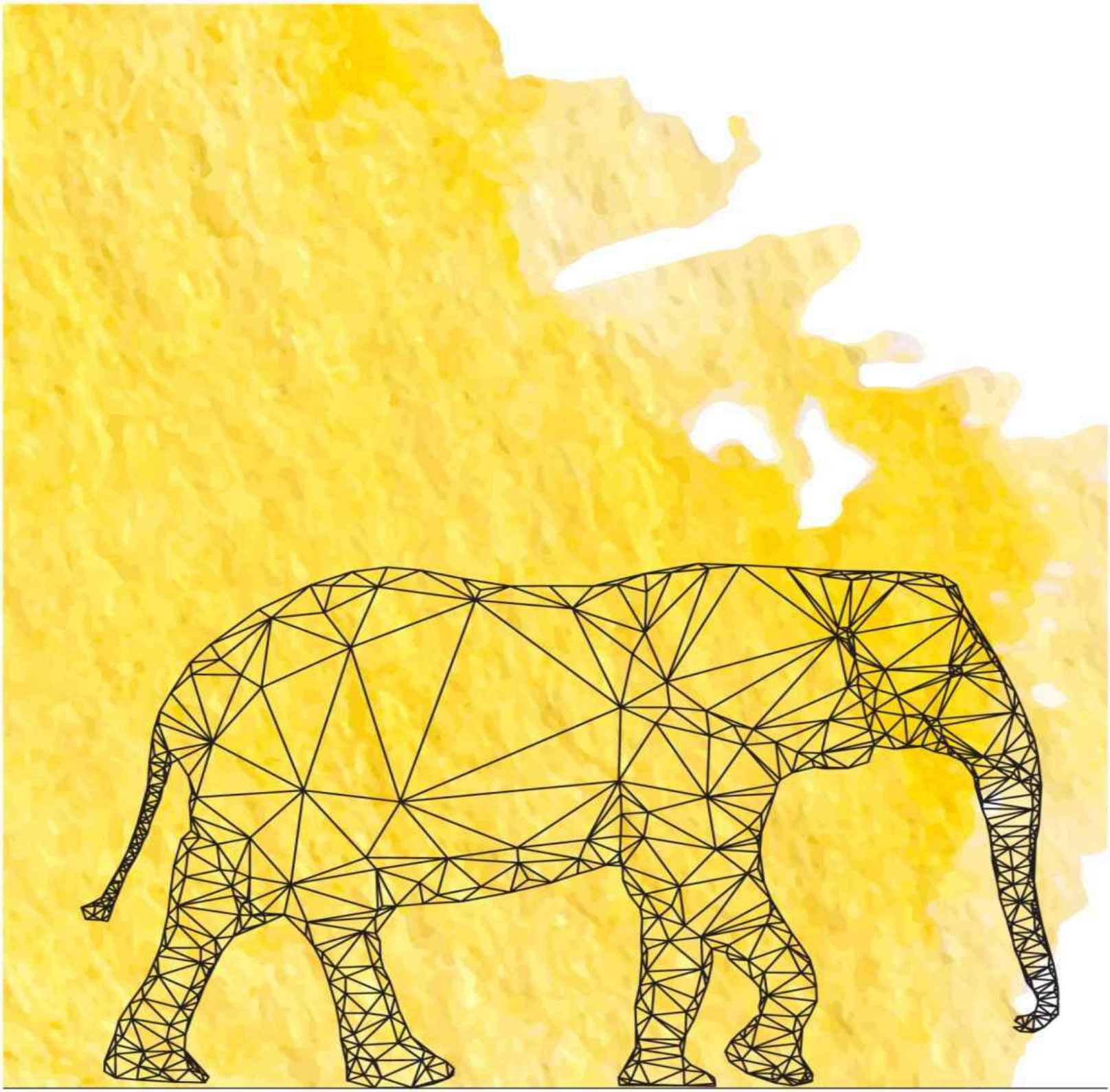

SYSTEM DESIGN INTERVIEW - AN INSIDER'S GUIDE

System Design Interviews
– An insider's guide

To my wife, parents and grandparents for their love and support.

FORWARD

Software engineering interviews are hard. Among all the interview questions, the hardest are the system design questions. These questions require the interviewees to design a high-level architecture for a software system, which could be news feed, Google search, etc. These questions are intimidating. Many people are afraid of system design interviews as there is no certain pattern to prepare. The questions are usually very big scoped and vague. The processes are open-ended and unclear without standard or correct answer.

System design interviews are widely adopted by companies because the skills tested in these interviews are similar to those required by a software engineer's daily work, namely the communication and problem solving skills. These abilities are evaluated in how an interviewee analyzes a vague problem, how he/she solves it step by step, how he/she explains the idea and discusses with others as well how to evaluate the system and optimize it.

The system design questions are open-ended, just like in the real world, there are many differences and variations in the systems. The idea of system design questions is to have a technical discussion on a problem. The desired outcome is to come up with a high level architecture solution to achieve the goals in the question. The discussions could go in many different ways depending on the goals of the interviewer. Some interviewers want to see a high level architecture covering all aspects, while some might choose one or more areas, typically algorithms, or bottlenecks of the system to have a deeper dive. Both the interviewer and interviewee shape the direction of the discussions. Constraints and tradeoffs are always discussed.

This goal of this book is to provide a reliable and easy to understand strategy

to approach system design questions. The process and justification of your ideas are the most important things in system design interviews. Thus the combination of right strategy and knowledge is vital to the success of your interview. This book provides valuable ways to fix both problems. By the time you finish the book, you are well-equipped to tackle any system design questions.

This book kicks off by providing solid knowledge of building a scalable system. The more knowledge gained through practical experiences or readings you have, the better you are at the system design interview.

This book takes a step by step approach showing how to scale a system. Scalability is the core of the knowledge base and at the same time, it is where candidates struggle the most. This step by step guide provides significant amount of information building up your knowledge base, which creates a solid foundation.

After building up the knowledge, it is time to apply the knowledge to real interview questions. This book provides three popular system design interview problems to illustrate the systematic approach with detailed steps you can follow. You will be good at the system design interviews with enough practices.

TABLE OF CONTENTS

CHAPTER ONE: SCALE FROM ZERO TO TEN MILLION USERS

Designing a system that supports millions of users is challenging. It's a journey that requires continuous refinement and endless improvement. In this chapter, we'll go through this journey together by building a system that supports a single user, and then scales up to 10 million users gradually. After reading this chapter, you'll master a handful of tools that will help you crack system design interview questions more easily.

One user

A journey of a thousand miles begins with a single step. Building a complex system is no difference. We need to start with something simple and basic. On day one, you are probably the only user and everything is running on a single server. Let's take a look of the single server setup.

Single server setup

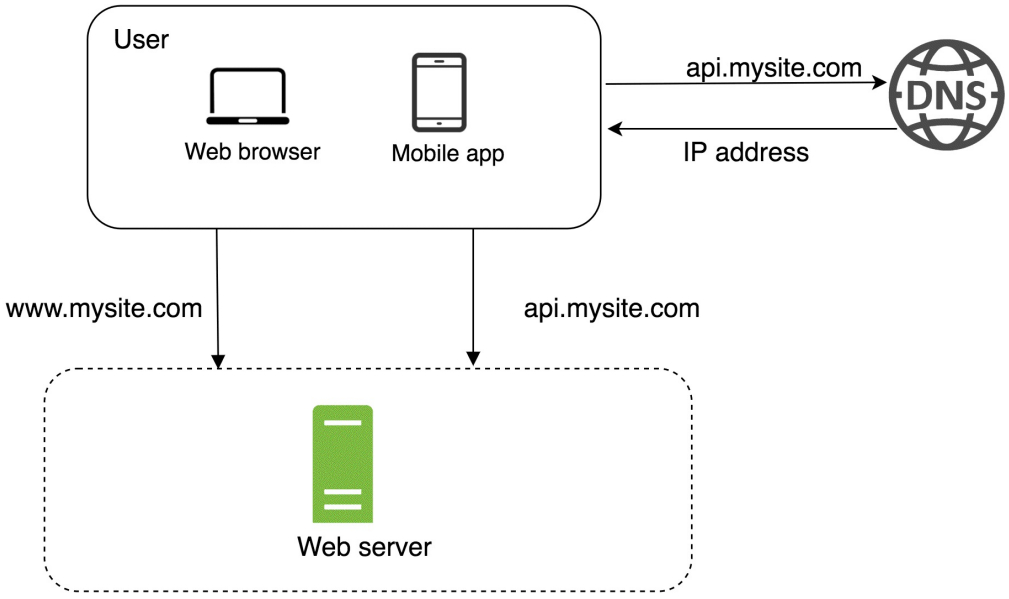

Figure 1

Figure 1 shows how a single server setup might look like. Everything is running in one web server: web app, database, search engine, etc. To better understand why it's set up this way, it is helpful to look into each component of the system in Figure 1.

• Usually, the Domain Name System (DNS) server is used as a paid service provided by the hosting company and is not running on your own server. Users connect to the DNS to obtain the Internet Protocol (IP) address of the server

where your website is hosted. Once the IP address is obtained, Hypertext Transfer Protocol (HTTP) requests are sent directly to your web server. [1]

• The traffic from your web server generally has two sources: web browser and mobile app.

1)- For web browser: the web server generates a html page and web browser renders the html page for users.

2)- For mobile app: The communication protocol used between mobile app and web server is usually HTTP protocol. JavaScript Object Notation (JSON) is a very commonly used API response format to transfer data due to its simplicity.

Figure 2 shows what an API response in JSON format might look like.

GET /users/12 – Retrieve user object for id = 12


```
{
    "id": 12,
    "firstName": "John",
    "lastName": "Smith",
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        "212 555-1234",
        "646 555-4567"
    ]
}
```

Figure 2

One hundred users

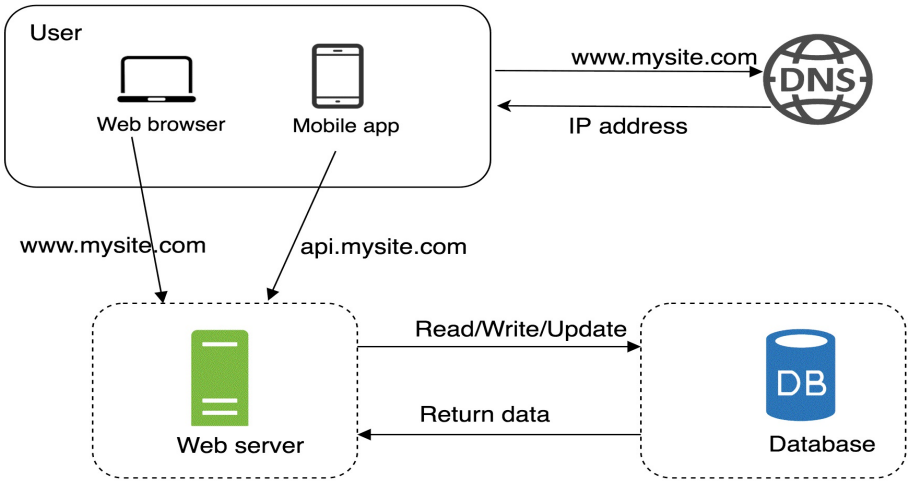

Figure 3

While the user base grows, it's time to think about separating out a single web server to multiple servers: one for web/mobile traffic, the other for database (Figure 3). Using separate servers for web tier and data tier allows them to scale independently of each other.

What databases to use?

For databases, you have to decide whether to use traditional relational databases or non-relational databases. Let's first take a look of their differences and best use cases for each.

Relational databases are also called relational database management systems (RDBMS) or SQL databases. The most popular ones are MySQL, Oracle database, PostgreSQL, etc. Relational databases represent and store data in tables and rows. You can also perform join operations using SQL across different database tables.

Non-Relational databases are also called NoSQL databases. The most popular ones are CoachDB, Neo4j, Cassandra, HBase, Amazon DynamoDB, etc. These databases are usually grouped into four categories: Key-value stores, Graph stores, Column stores, and Document stores. You cannot perform join operations in non-relational databases.

For most developers, relational databases are the go-to option because a table structure is easy to understand and can easily support over 10 million users, but there are some reasons to explore beyond relational databases. Here are a few reasons why non-relational databases might be the right choice for you:

- Your application requires super low latency.
- Your data are unstructured or you don't have any relational data.
- You need to store massive amounts of data (> 5 TB in one year).

Vertical scaling vs horizontal scaling

Vertical scaling, also referred as "scale up", means that you scale by adding more power (CPU, RAM, etc) to an existing machine. Horizontal scaling, also referred as "scale out", means that you scale by adding more machines into your pool of resources.

When the traffic is low, vertical scaling is a great option. The simplicity of vertical scaling is its main advantage. Unfortunately, it comes with serious limitations.

- Vertical scaling has its hard limit. No matter how much money you can spend, it's not possible to add unlimited CPU and memory to a single server.
- Vertical scaling doesn't have failover and redundancy. If the server goes down, the website/app goes down with it completely.

Due to those limitations of vertical scaling, horizontal scaling is almost always

more desirable.

One thousand users

In the previous design, users are connected to web servers straightly. This means, in case the web server goes offline, users will not be able to access it. In another scenario, if many users try to access the web server simultaneously and web server hits its limit to handle the load, users generally experience slower response or might not able to connect to the server at all. To address those problems, load balancer comes to rescue.

Load balancer

A load balancer distributes incoming traffic among web servers defined in a load-balanced set. Figure 4 shows how load balancer works.

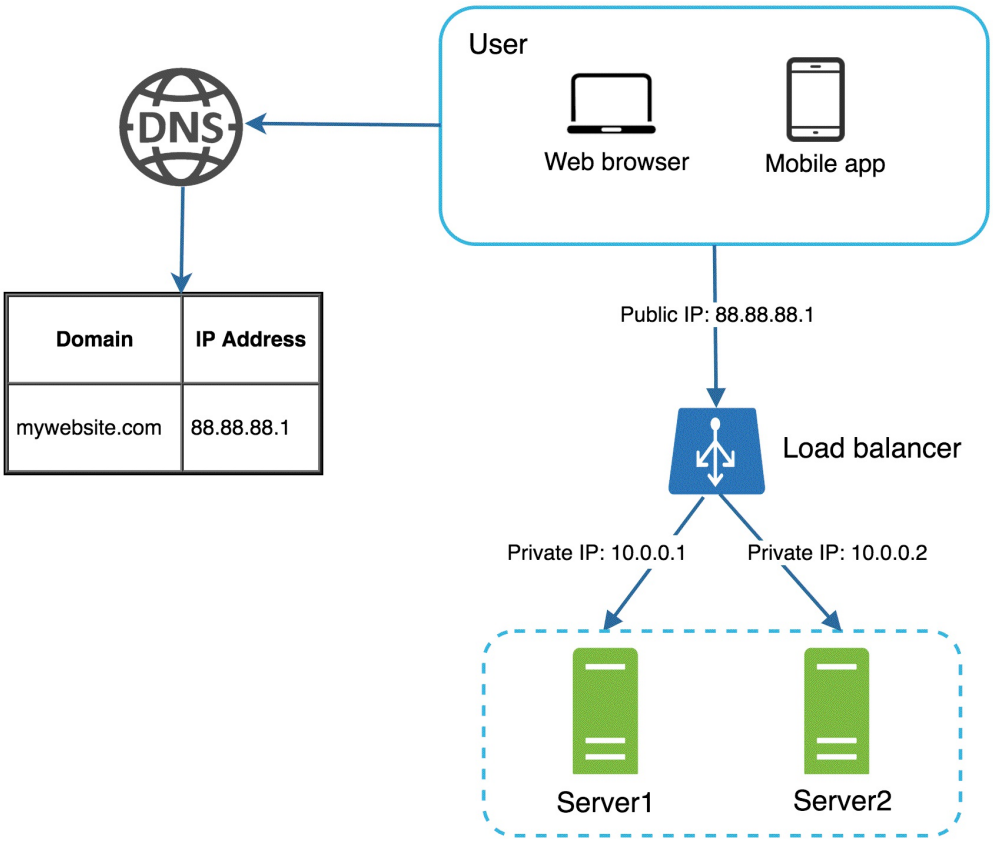

Figure 4

As shown in Figure 4, users always connect to the load balancer's public IP directly. Web servers are not reachable directly by the client anymore.

The load balancer distributes the traffic among web servers. For better security, the communication between servers always use private IPs. A private IP address is an IP address that's reachable only between servers in the same network. It's not reachable over the Internet. The load balancer communicates with web servers through private IPs.

After a load balancer and a second web server are added as shown in Figure 4, we have successfully solved the no failover issue and improved the availability of the web tier.

• If server 1 goes offline, all the traffic will be routed to server 2. The website won't go offline. In this case, you need to add a new healthy web server to the server pool to balance the load.
• What if the website grows extremely rapidly that two servers aren't enough to handle the traffic? With load balancer, it becomes quite easy! You only need to add more servers to the web server pool and the load balancer will route the traffic for you.

Now the web tier looks good, what about the data tier? The current design only has one database. Obviously it doesn't support failover and redundancy. A common technique to address this is called database replication. Let's take a close look at it.

Database replication

Database replication can be used in many database management systems, usually with a master/slave relationship between the original (master) and the copies (slaves).

A master database receives data from applications. A slave database gets copies of that data from the master. Slaves are therefore read-only from the application's point of view while a master is read-write. All of the data-modifying commands like insert, delete or update must be sent to the master. The read command is sent to the slave. Most applications require a much higher ratio of reads to writes. So setting up master-slave replication could let an application distributes its queries more evenly and effectively. Figure 5 shows a master database with multiple slave databases.

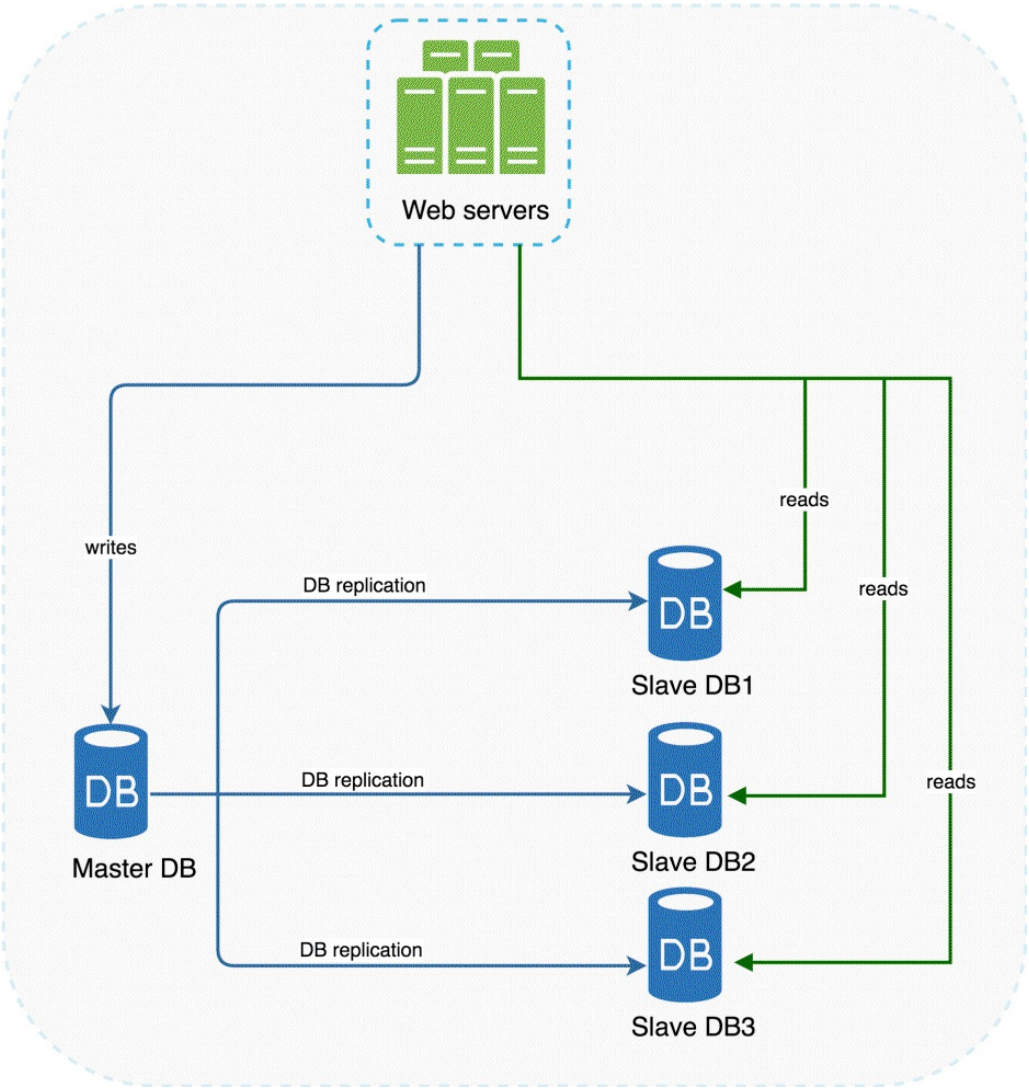

Figure 5

In the case of MySQL, the advantages of replication include:

• Scale-out solutions - spreading the load among multiple slaves to improve performance. In this environment, all writes and updates must take place on the master server. Reads, however, may take place on one or more slaves. This model can improve the performance of writes (since the master is dedicated to updates), while dramatically increasing read speed across an increasing number of slaves. [2]
• Data security - because data is replicated to the slave, and the slave can pause the replication process, it is possible to run backup services on the slave without corrupting the corresponding master data. [2]
• Analytics - live data can be created on the master, while the analysis of the information can take place on the slave without affecting the performance of the master. [2]
• Long-distance data distribution - if a branch office would like to work with a copy of your main data, you can use replication to create a local copy of the data for their use without requiring permanent access to the master. [2]

Earlier we talked about how load balancer helps to improve system availability. You might ask the same question here: what if one of the databases goes offline? The design in Figure 5 could handle this case gracefully:

1)- If there is only one slave database and it goes offline, read operations will be directed to the master database temporarily. As soon as you notice the slave database issue, a new slave database should be promoted. In case of multiple slave databases are online, we don't even need to direct read operations to the master database because the other online salves could pick up the operations. Promoting a new slave database is the only thing needs to be done.

2)- If the master database goes offline, a slave database could be promoted to

be the new master. All the database operations will be temporarily executed on the new master database. A new slave database needs to be setup for replication as soon as possible. In production systems, promoting a new master is more complicated because data in a slave database might not up to date. The missing data might be able to be added back by running some data recovery scripts. Some other replication methods like multi-masters, circular replication, etc could help, but they are more complicated, which are beyond the scope of this book. For readers who are interested, please refer to the materials listed in bibliography [3] [4].

After adding load balancer and database replication, the design for a thousand users is shown in Figure 6.

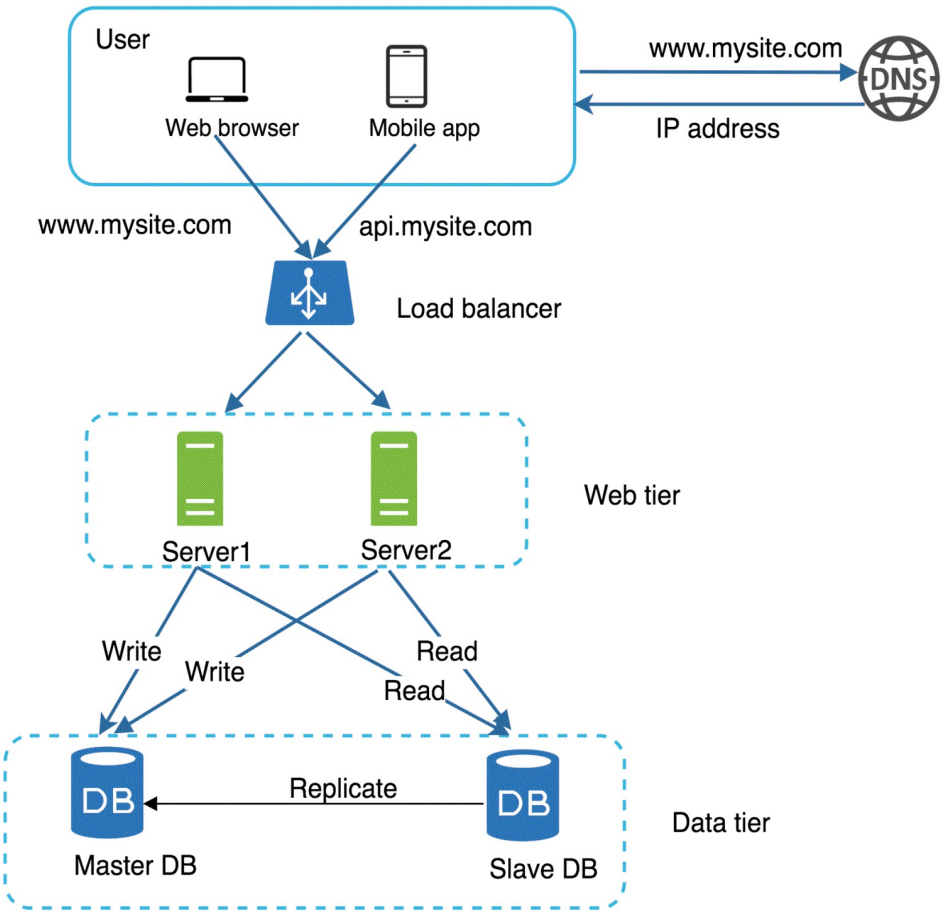

Figure 6

Let's take a close look how the flow works:

• A user gets the IP address of the load balancer from DNS.
• A user connects to the load balancer by IP address fetched from previous step.
• Traffic is routed to either Server 1 or Server 2.
• A web server reads user data from slave database.
• A web server routes any data-modifying operations, including write, update and delete schema changes to master database.

Ten thousand users

Now that you have pretty solid design for web and data tiers, it's time to get started on improving load/response time of the website/application. This can be done by adding a cache layer and shifting static content (javascript/css/image/video files) to content delivery network (CDN).

Cache

A cache is a temporary storage area that stores the result of expensive responses or frequently accessed data, usually in memory, so that subsequent requests can be served much more quickly. In our previous design illustrated in Figure 6, every time a new web page loads, usually one or more database calls are made to fetch data. Performance of the application is greatly affected by the unnecessary database calls. Cache can be effectively used to mitigate this problem.

Enter the cache tier

The cache tier is a temporary data store layer, which is much faster than the database. The benefits of having a separate cache tier are better utilization of memory and CPU resources and having the ability to scale the cache tier independently of other tiers. Figure 7 shows a possible setup for a cache tier: A web server, after receiving a request, first checks if the cache has the response available. If so, it sends the data to the client. If not, it queries the database, and stores the response in itself and sends it back to the client. This way, the load to database is reduced.

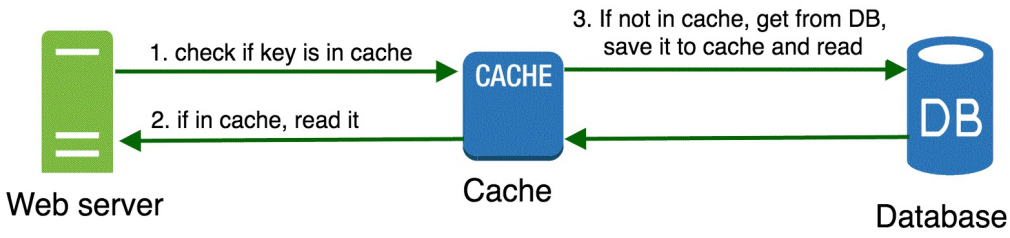

Figure 7

Interacting with cache servers is simple because most cache servers provide APIs for common programming languages. Figure 8 shows how a typical memcache API looks like. All you need to specify is the key of the value you want to store and time to live (TTL) after which the object would be removed from the cache.

```
SECONDS = 1
cache.set('myKey', 'hi there', 3600 * SECONDS)
cache.get('myKey')
```

Figure 8

Considerations for using cache

Lifetime of Cached Data. It's a good idea to implement an expiration policy that causes data to be removed from the cache if it is not accessed for a specified period of time. Do not make the expiration period too short because this can cause applications to continually fetch data from the database. Similarly, do not make the expiration period so long that the cached data are likely to become stale. [3]

Consistency. An item in the database may be changed at any time by an external service, and this change might not be reflected in the cache until the next time the item is loaded into the cache. When scaling across multiple

regions, maintaining consistency between data in cache and the persistent storage becomes a big technical challenge. For further details, the paper "Scaling Memcache at Facebook" published by Facebook explains in detail how Facebook solves the consistency issue. [4]

Mitigating Failures. A single cache server represents a potential single point of failure. A single point of failure is a part of a system that, if it fails, will stop the entire system from working. As such, multiple cache servers spreading across different datacenters are recommended to implement the cache tier. Another recommended approach is to overprovision the required memory by some percentages. This provides a buffer of extra caching capacity as the memory usage increases. If you are interested in learning more about advanced mitigating failure techniques, I recommend reading this article "Best Practices for Implementing Amazon ElastiCache" [5].

Evicting Data. It is possible that cached data fills up all the allowed memory in a server. In this case, any requests to add new items to the cache might cause some items to be removed forcibly. This is called cache eviction. Least-recently-used (LRU) is the most popular cache eviction policy. When performing LRU caching policy, you always throw out the data that was least recently used. However, other eviction policies like Least Frequently Used (LFU) or First In First Out (FIFO) might be also used to satisfy your use case.

Content Delivery Network (CDN)

A CDN is a network of geographically dispersed servers. Each CDN server caches static content of a site like images, videos, css, javascript files, etc and dynamic content. Dynamic content caching enables the caching of HTML pages based on request path, query strings, cookies, and request headers. Dynamic content caching is a relatively new concept and beyond the scope of this book.

If you are interested in knowing more, please read the two articles mentioned in bibliography [5] [6]. This book will focus on using CDN for caching static content.

In high level, here is how CDN works: when a user visits a website, a server closest to the user will deliver the static content, therefore ensuring a faster download time of assets. Intuitively, you know that the further away users are from a website's datacenter, the slower the website loads. For example, if a website's servers are located in San Francisco, people in Los Angeles will get the content faster than people in Europe.

A global CDN would help to solve this problem by allowing users from Europe to download static content from a closer source, say in London. This reduces latency and provides a faster load time. Figure 9 is a great example that shows how CDN improves the load time [5].

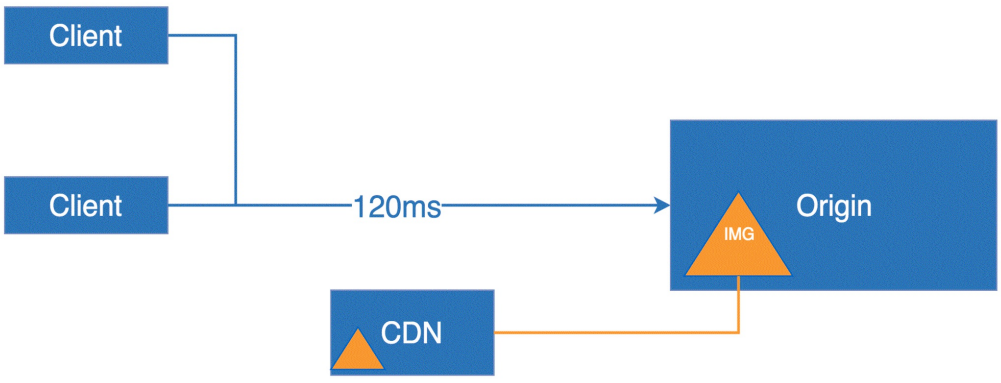

Figure 9

Now that we know how CDN works in high level, let's use the example illustrated in Figure 10 to examine more closely.

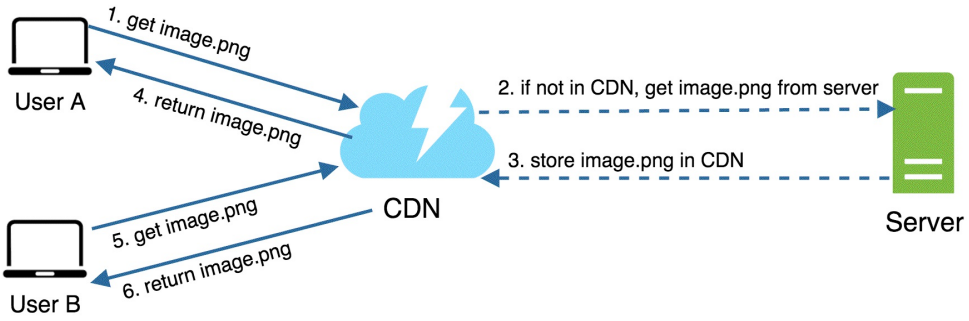

Figure 10

1)- User A requests image.png by using a URL with specified domain name, such as www.mycdn.com/assets/image.png. DNS routes the request to the best performing CDN server that is geographically closest to the user.

2)- If the CDN server does not have image.png in the cache, CDN server requests the file from the origin. The origin can be a web server or online storage like Amazon Simple Storage Service (S3).

3)- The origin returns image.png to the CDN server, including optional HTTP headers describing the file's Time-to-Live (TTL).

4)- The CDN caches the file and returns the file to User A. The file remains cached in the CDN until the TTL expires.

5)- The same user (User A) and additional users (User B in Figure 10) may then request the same file (image.png) using that same URL.

6)- If the TTL for the file hasn't expired, CDN returns the file from the cache. This results in a faster, more responsive user experience.

Considerations of using CDN

Cost. CDNs are paid, third-party services. You are charged for data transfers

from the CDN. Setting a realistic cache expiry period for content helps to ensure freshness, but not so short as to cause repeated reloading of content from the web server or online storage to the CDN. Assets that are rarely downloaded will cause the two transaction charges without providing any significant reduction in server load.

Invalidating files. If you need to remove a file from CDN before it expires, you can do one of the following:

- Invalidate the object from the CDN using its provided API.
- Use object versioning to serve a different version of the object.

CDN fallback. You should consider how your website copes with a failure of the CDN. You can setup your website to detect failure of CDN and request resources from the origin if CDN is unavailable.

Design for ten thousand users

Figure 11 shows the design after we add CDN and cache tier.

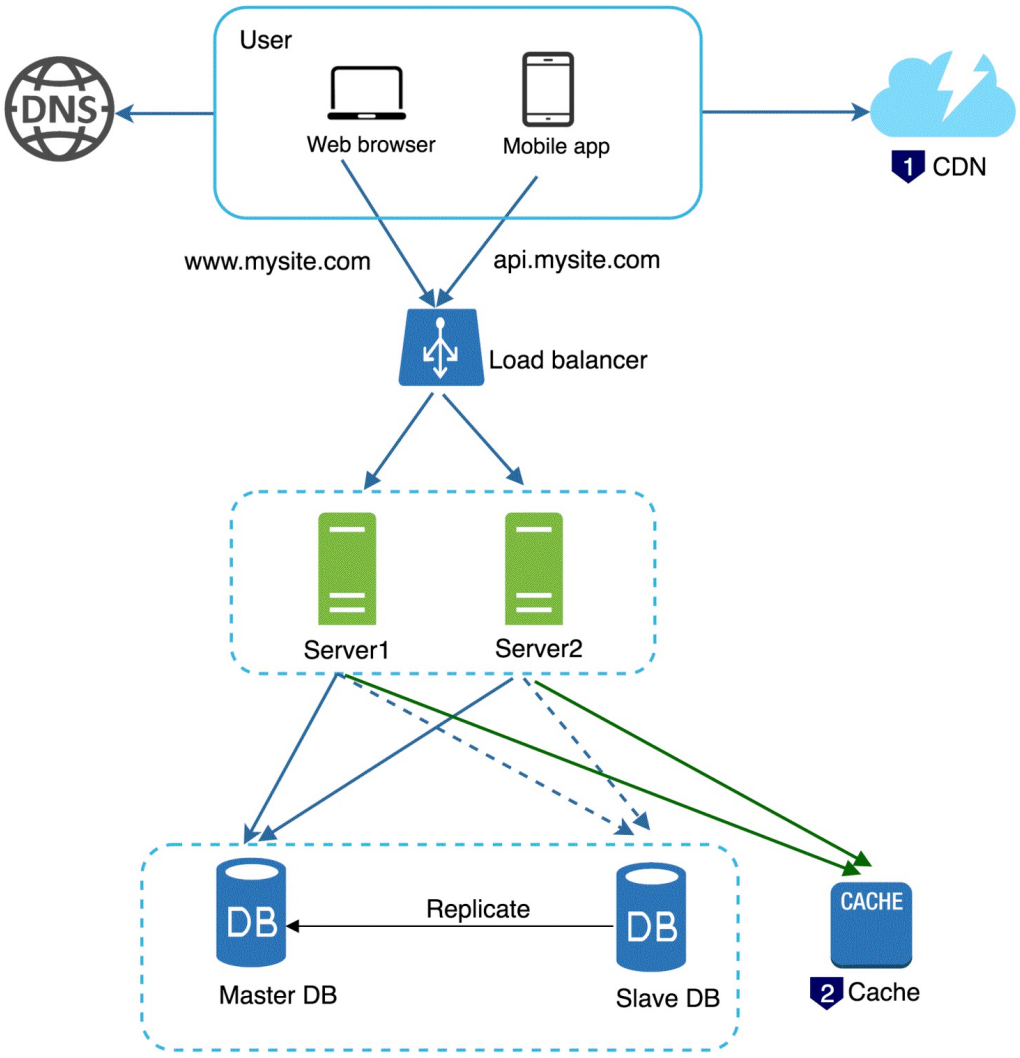

Figure 11

• Static assets (JS, CSS, images, etc) are no longer served from the web servers. They are fetched from the CDN for better performance. The load on web servers is heavily reduced.

• Database load is lightened by caching data in the cache tier.

One hundred thousand users

Now it's time to consider scaling the web tier horizontally. To make this happen, we need to move state (for instance user session data) out of the web tier. In web application design, the golden rule to achieve scalability is storing state not in the web tier but in the relational database or NoSQL. Each web server in the cluster can then access state data from databases. It's called stateless web tier/architecture.

Stateless web tier

Let's first inspect the difference between stateful and stateless servers. A stateful server remembers client data (state) from one request to the next. A stateless server keeps no state information. Because stateless servers don't hold any state, all the web servers are interchangeable, allowing better scalability.

Stateful Architecture

Figure 12 shows an example of a stateful architecture.

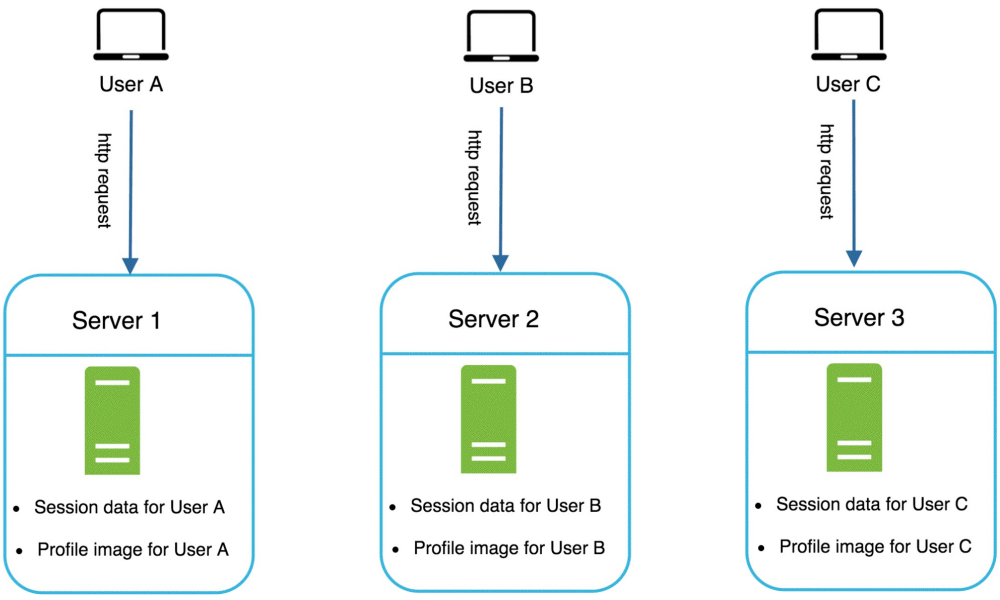

Figure 12 Stateful architecture

For the stateful architecture in Figure 12, user A's session data and profile image are stored in Server 1. Therefore, to authenticate User A, http requests have to be routed to Server 1. If a request is sent to other servers like Server 2, authentication would fail because Server 2 doesn't contain User A's session data. Similarly, all http requests from User B have to be routed to Server 2; all requests from User C have to be sent to Server 3.

The issue here is that you now have to make sure every request from the same client is routed to the same server. This can be done with sticky sessions (bind a user's session to a specific server) [7] in most load balancers, but it adds overhead. Additionally, it makes adding or removing servers much more difficult as you have to be very careful when you route users. In this design, it's challenging to handle server failures and scale server pools dynamically.

Stateless Architecture

Figure 13 shows the stateless architecture.

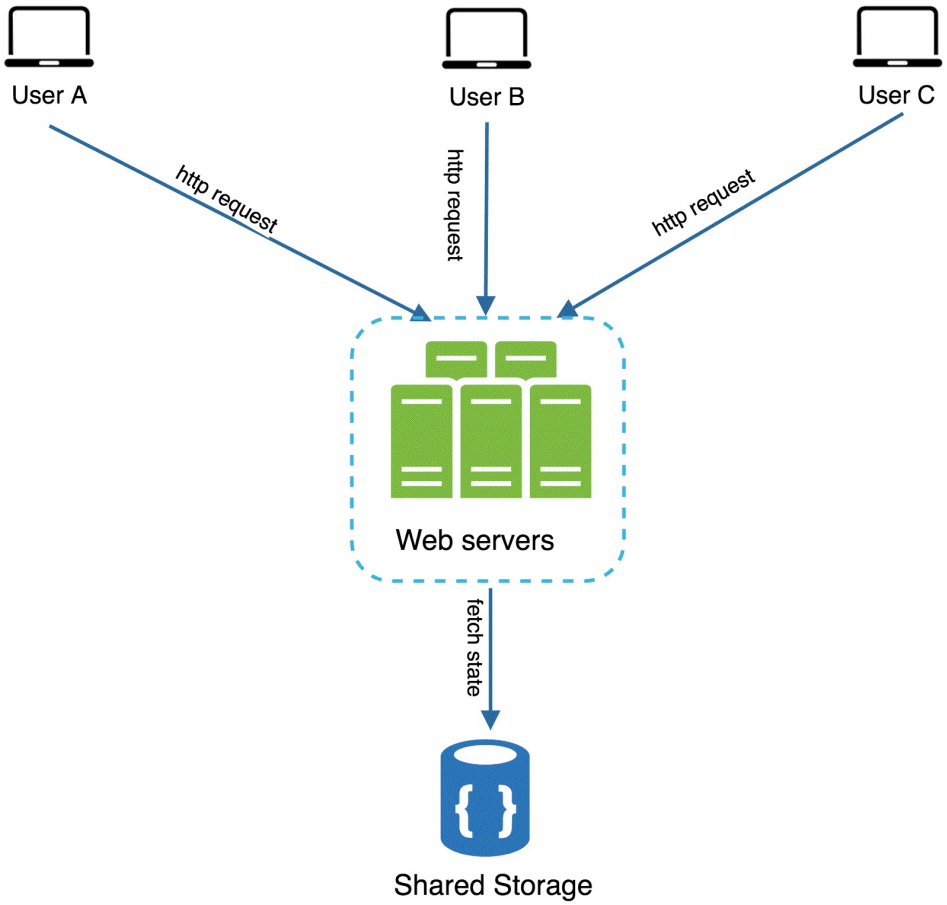

Figure 13 Stateless architecture

In this stateless architecture, http requests from users can be sent to any web server. A web server then fetch state from a shared data store. All of the state is stored in a shared data store and kept outside of web servers. Scaling the web tier is simple because you can simply add or remove web servers based on traffic load.

Design for one hundred thousand users

Figure 14 shows the new design.

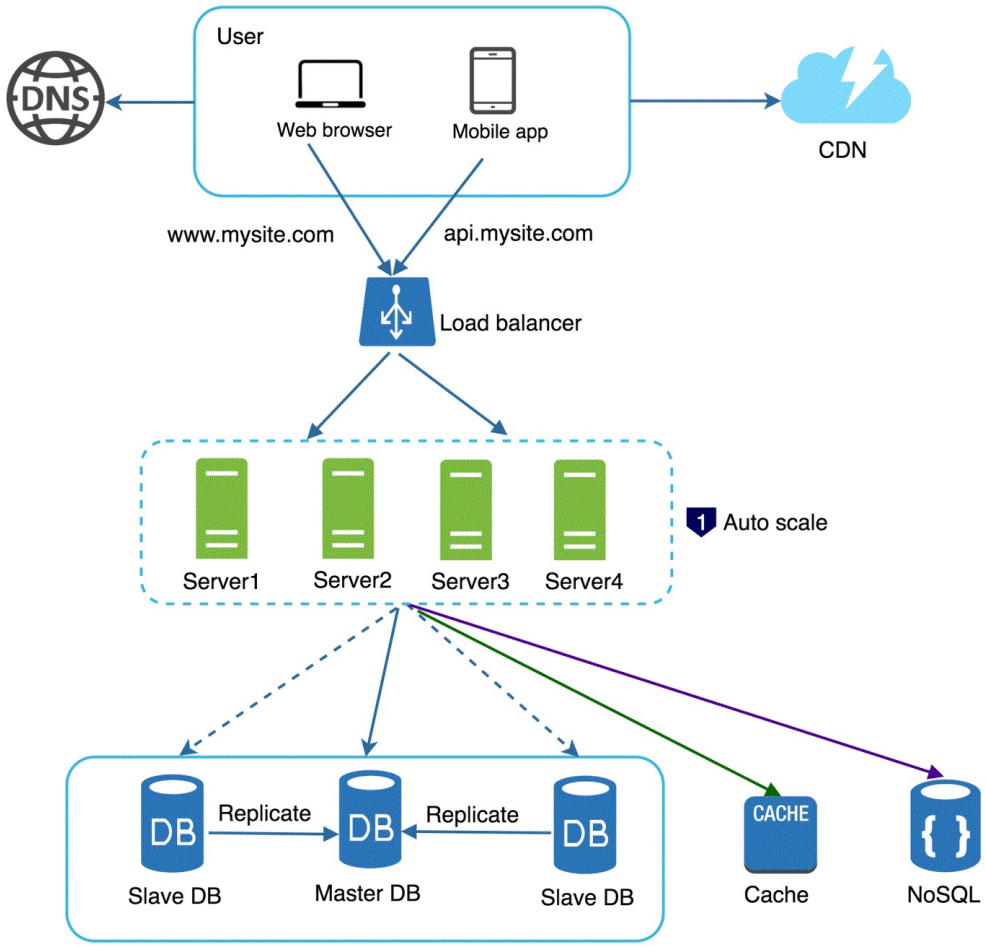

Figure 14

In this design, we move session state out of the web tier and store them in shared data store. The shared data store could be database, Memcache/Redis, NoSQL, etc. NoSQL data store is chosen here because your session data are distributed and replicated. Auto scaling means adding/removing web servers automatically based on the traffic load. After state is removed from web servers, auto scaling of web tier could be easily achieved.

Five hundred thousand users

Your website grows rapidly and attracts significant amount of users internationally. To improve availability and provide a better user experience across wider geographical areas, deploying your site to more than one datacenter is crucial.

Multiple datacenters

Figure 15 shows an example with two datacenter setup. In a normal state of operation, users would be geoDNS-routed to the closest datacenter, with a split of x% and (100 – x)% (in this case US-East and US-West). geoDNS is a DNS service that allows domain names to be resolved to IP addresses based on the location of a user.

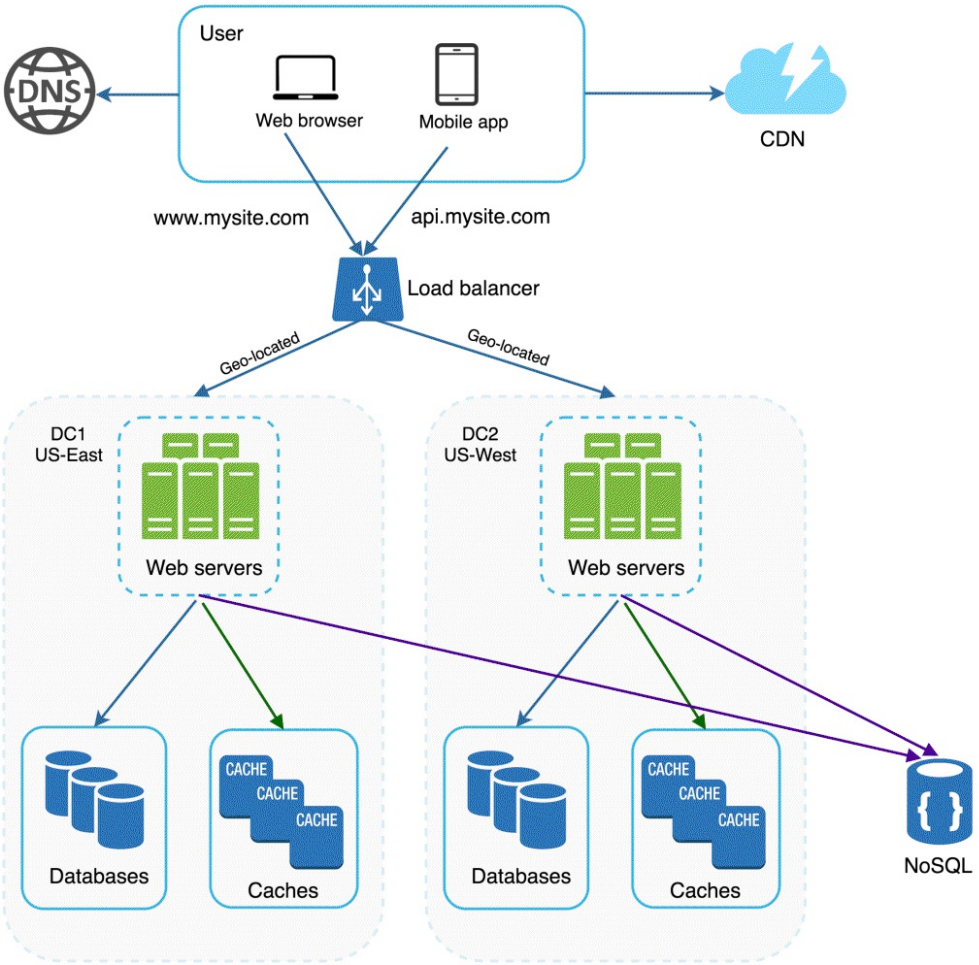

Figure 15

In the event of any significant datacenter outage, we will direct all of traffic to a healthy datacenter. In Figure 16, datacenter 2 (US-West) is offline and 100% of the traffic is going to datacenter 1 (US-East).

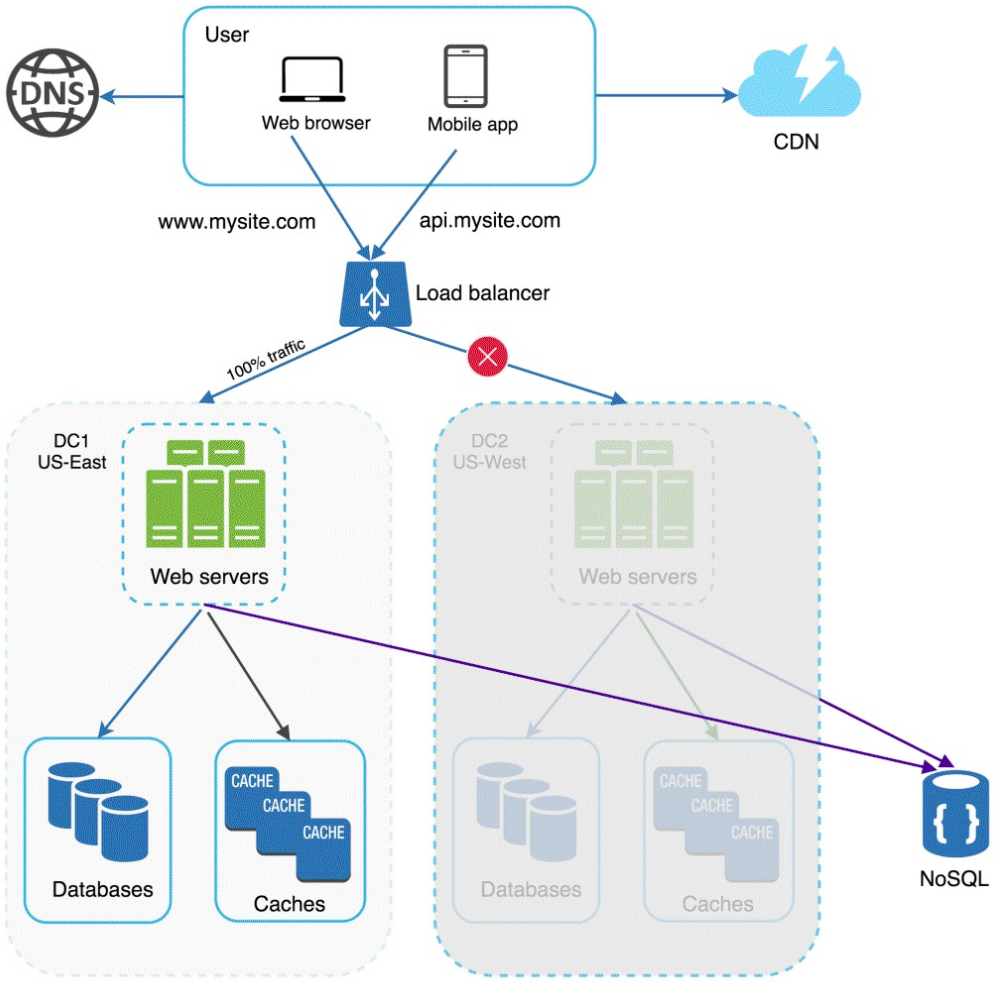

Figure 16

There are several technical challenges to solve in order to achieve multiple datacenter setup:

Traffic redirection. Effective tools are needed to direct traffic to the correct datacenter. GeoDNS can be used to direct traffic to the nearest datacenter based on geographical detection while looking up a domain name.

Data synchronization. Users from different regions are likely to use different local databases or caches, and they might be routed to a datacenter where data is not available in failover case. A common strategy is to replicate the data

across multiple datacenters. To learn more, read how Netflix implements multi-datacenter asynchronous replication. [8]

One million users

We've come a long way from where we begin. Now we want to further decouple our infrastructure by breaking our web tier into smaller services. The key is to build components that do not have tight dependencies on each other. If one component fails, the other components in the system can continue to work as if no failure happened. Messaging queue is a key strategy employed in many distributed environments to achieve loose coupling. Let's take a close look.

Message queue

A message queue is a durable component, usually stored in memory and supports high availability. It buffers and distributes asynchronous requests. The basic architecture of a message queue is simple. Input services called producers/publishers create messages and deliver them to the message queue. Other services or servers, called consumers/subscribers, connect to the queue and subscribe to the messages to be processed (showed in Figure 17).

Figure 17

Message queues allow web servers to respond to requests quickly instead of being forced to perform resource-heavy procedures on the spot. Message queues are also good when you want to distribute a message to multiple recipients for consumption or for balancing loads between workers (consumers).

Consider the following use case. Assume your application allows users to do

two tasks: 1) customize photos (cropping, re-coloring, etc) 2) generate pdf report. Different operations take different processing time, ranging from a few seconds to several minutes.

The figure below (Figure 18) provides an overview of the architecture that meets requirements listed above.

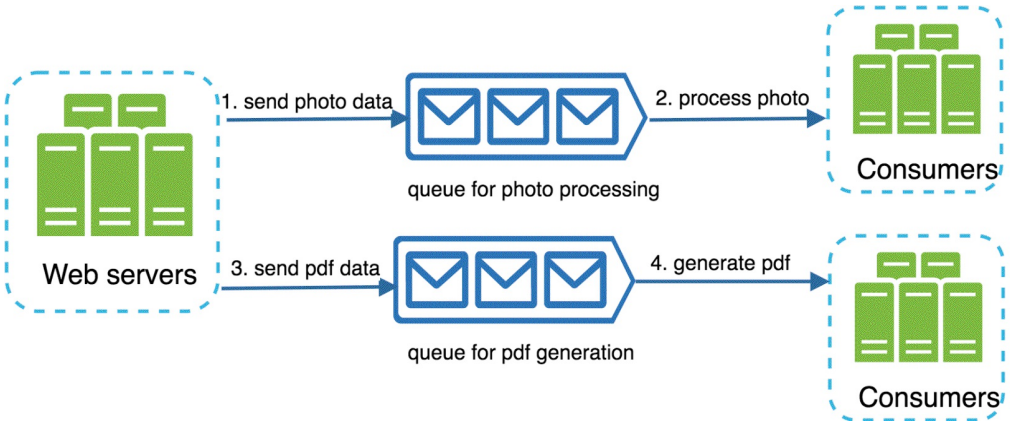

Figure 18

• Web servers send photo processing requests to the message queue which stores data necessary to process photos.
• Photo processing workers (consumers) read messages from the queue and process requests.
• Similar process is applied for pdf generation (step 3 and 4 in Figure 18). PDF generation task has separate queue and consumers.
• Those two tasks (photo processing and pdf generation) are processed in parallel.

What makes this a preferred architecture for building a scalable and reliable application? The main reason is decoupling. Now the producer can post a message to the queue when the consumer is not available to process it. The consumer can read messages from the queue even when the producer is not available.

Logging, metrics, automation

When working with a small site, which runs on just a few servers, monitoring logs, metrics and supporting automation are good practice but not a necessity. However, now your site grows to a serious business, monitoring logs, metrics and investing in automation are essential.

Logging. Monitoring error logs is very important because it helps to discover errors and problems within the system fast. You can monitor error logs per server level and use tools to aggregate them to a centralized service for easy search and viewing.

Metrics. Collecting different types of metrics can help you to gain business insights about your site, and watch health status of each infrastructure component. For example, some of the following metrics could be useful: 1) host level metrics - CPU, Memory, disk I/O, etc. 2) Aggregated level metrics – eg, performance of entire database tier. 3) Key business metrics – daily active users, revenue, etc.

Automation. The infrastructure is getting big. We need to build or leverage existing automation tools to improve efficiency. Continuous integration is a popular practice. It means each code check-in is verified by an automated build, allowing teams to detect problems early. Meanwhile, automating your build, test, deploy process and operation work could improve efficiency significantly.

Design for one million users

Figure 19 shows the updated design with what we've learnt so far. Please note in this figure, it only contains one datacenter because the other datacenter couldn't fit in a single diagram.

• Message queue is added in this design. It helps to make the system more loosely coupled and failure resilient.

• Logging, monitoring, metrics and automation tools are added.

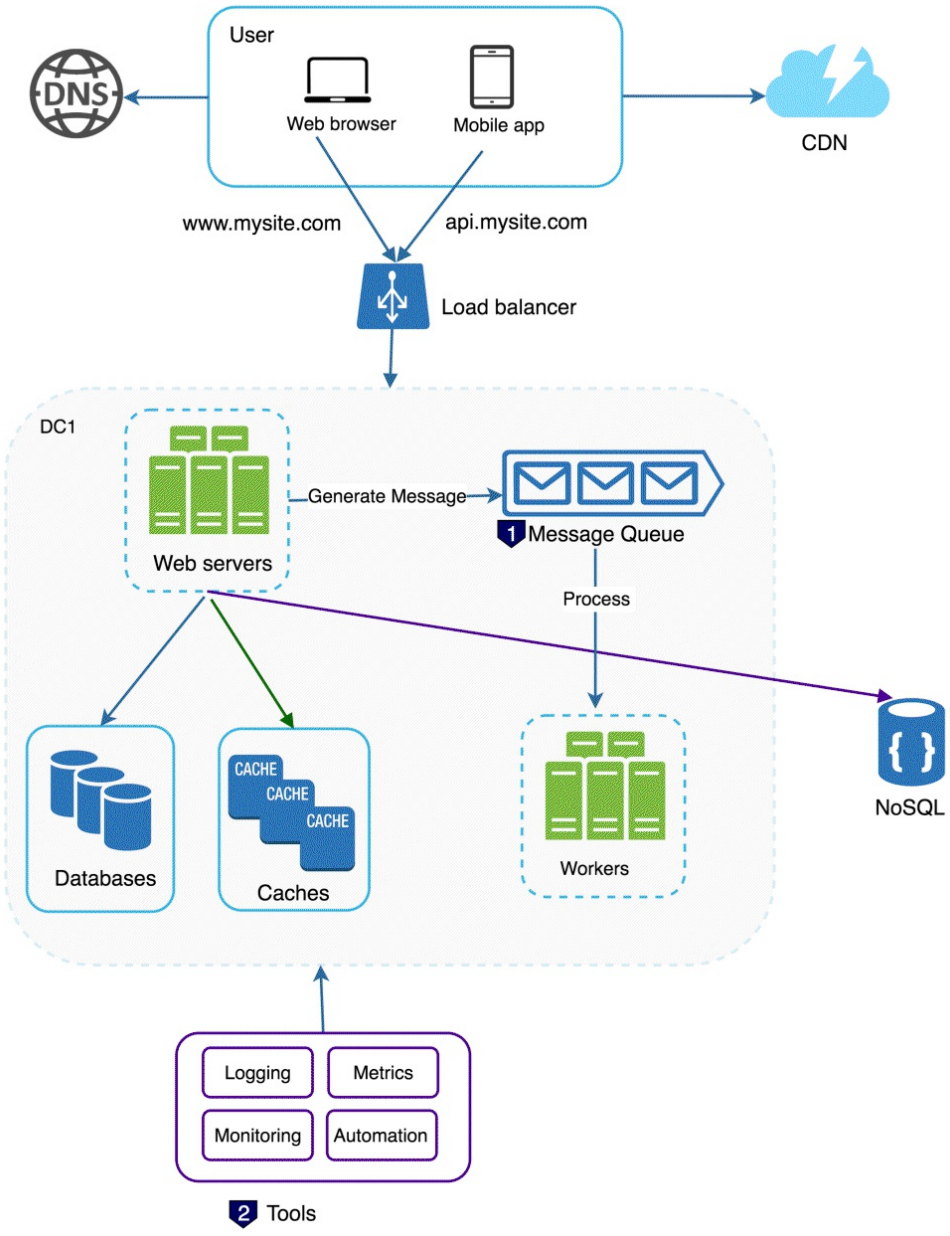

Figure 19

Five million users

Your application suddenly becomes popular. Traffic and data are started to grow everyday, and your database gets more and more overloaded. You need to scale your data tier.

Database scaling

While automated horizontal scaling (sharding) of the database tier would be an ideal solution, the implementation is complicated. The general recommendation is to start with everything you could optimize first. If the performance is still not good, it's time to tackle the bitter medicine - horizontal scaling [9] [10]. Let's first inspect easier solutions.

Get more powerful database servers.

There are some really powerful database servers. According to Amazon Relational Database Service (RDS) [11], you can get a database server with 244GB of RAM and 32 cores. This kind of powerful database servers could take away your worry about database scaling for a while. For example, stackoverflow.com in 2013 had over 10 million monthly unique visitors and it only had 1 master database! [12]

Lighten your database load

If your application is bound by read performance, you can add caches or database replicas (read from slaves). They provide additional read capacity without heavily modifying your application.

Vertical partition by functionality

It means splitting a single database into multiple databases based on

functionality. For example, an e-commerce website that implements separate business functions for billing and displaying products could store billing data in billing database and product data in product database. Different databases can be scaled independently of each other.

While partitioning by functionality, an important factor to consider is which table needs to be partitioned. Not everything needs to be partitioned. In most cases, only a few tables occupy the majority of the disk space so we should focus on large tables. Very little is gained by partitioning small tables.

The downside of vertically partition by functionality is that it is very difficult to join across databases.

Horizontal partition (sharding)

Along the path to high scalability, you will eventually end up needing to partition the database horizontally. Sharding separates very large databases into smaller, more easily managed parts called data shards. Each shard typically shares the same schema, though the actual data on each shard is unique to that shard. Sharding allows a database tier to scale along with its data and traffic growth. Many sharding strategies allow additional database servers to be added.

Figure 20 shows an example of what a sharded database looks like. Each user data is allocated to a database server based on the user id. Anytime you want to access a user's data, you use a hash function to find the corresponding shard. In the following case, user_id % 4 is used as the hash function. If the result is equal to 0, shard 0 is used to fetch data. If the result is equal to 1, shard 1 is used. The same logic applies to other shards. Figure 21 shows what users table looks like in sharded databases.

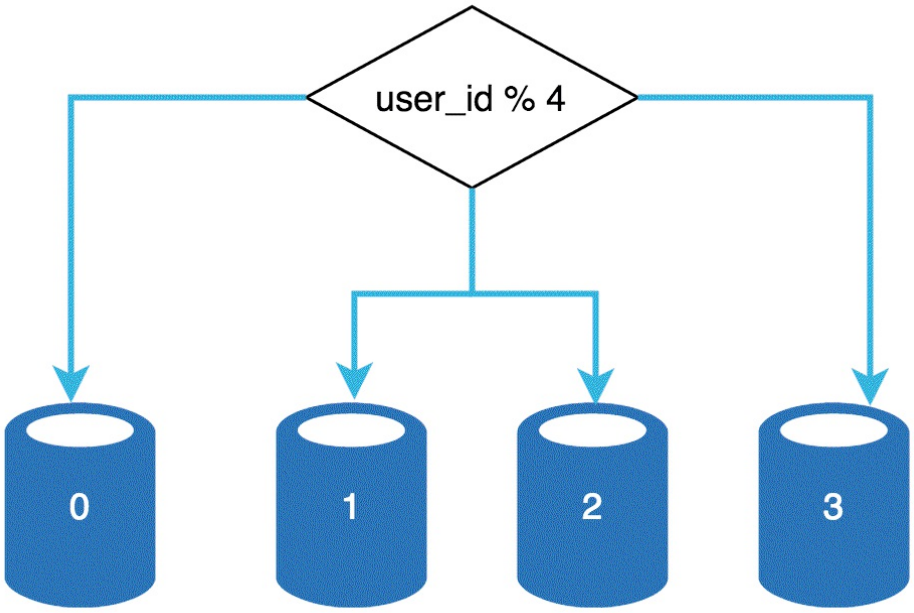

Figure 20 User database with sharding

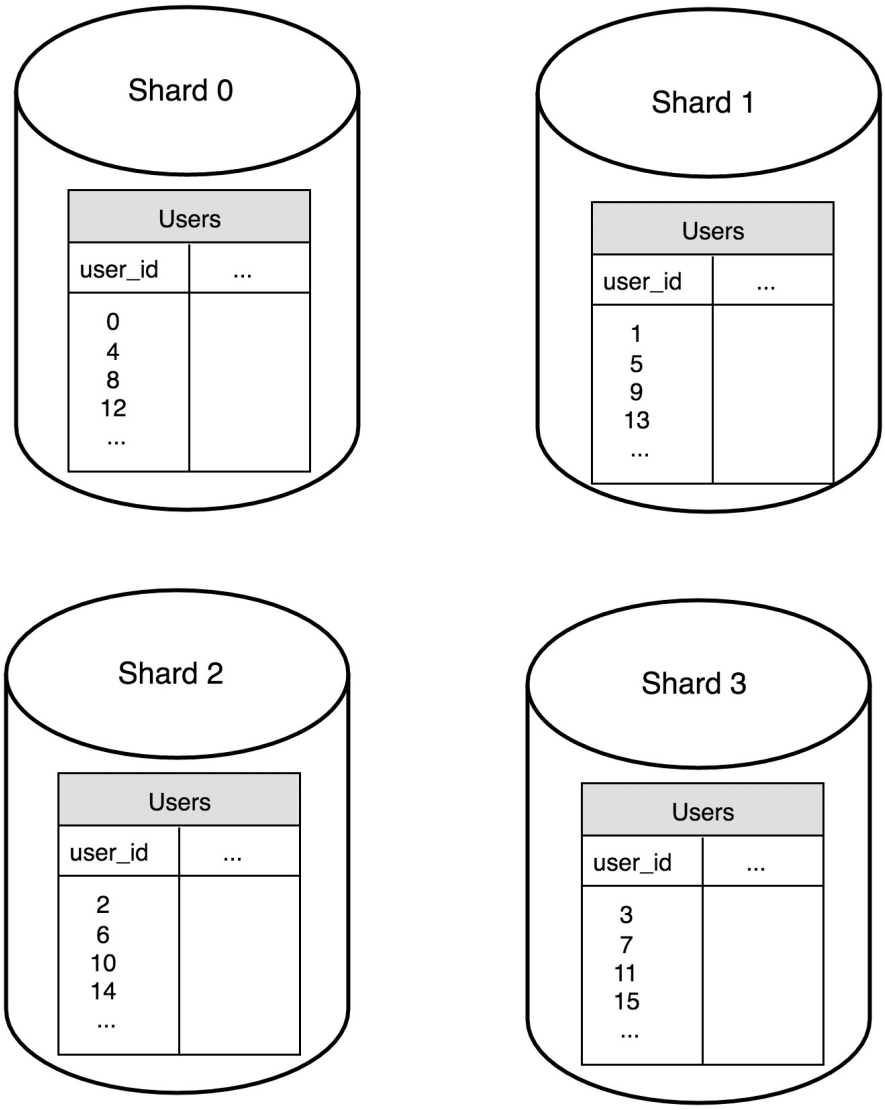

Figure 21 Users table in sharded databases

The most important factor when implementing this partitioning strategy is the choice of sharding key. Sharding key (also referred as partition key) is comprised of one or more columns which determines how data should be distributed. For instance, “user_id” is the sharding key in previous example (Figure 21). A sharding key allows you to retrieve and modify data efficiently by routing database queries to the correct database. Entries with the same sharding key are stored in the same database server. When choosing a sharding

key, it's important to ensure that data is as evenly distributed as possible.

Once a database has been sharded, new challenges are introduced to perform queries on the database. Below are some of the constraints and additional complexities introduced by sharding:

Resharding data. Resharding data is needed when 1) A single shard could no longer hold all the data in it due to rapid data growth. 2) Certain shards might experience shard exhaustion faster than others due to uneven distribution of data. When shard exhaustion happens, it requires updating the sharding function and moving data around the cluster. Doing both at the same time while maintaining consistency and availability is hard. Clever choice of sharding function can reduce the amount of transferred data. Consistent Hashing [13] is such an algorithm.

Celebrity problem. It's also called hotkey problem. Excessive access to a specific shard could cause server overload. Imagine data for Katy Perry, Justin Bieber and Lady Gaga all end up on the same shard. For social applications, that shard would become overwhelmed with read operations. In this case, the data of a celebrity might need to be placed on a separate shard by itself. It could even need further partition.

Join and denormalization. Once a database has been sharded across multiple servers, it is hard to perform joins across database shards due to performance and complexity constraints. A common workaround is to de-normalize the database so that queries that previously require joins can be performed on a single table.

Moving some functionality to NoSQL

Depending on your business use cases, you might be able to move some of the

functionalities to NoSQL. For example, you can move non-relational data to NoSQL. Here is an article that covers a lot of use cases that NoSQL is good for [14].

Design for five million users

In Figure 22, we shard databases to support rapidly increasing data traffic. At the same time, some of the non-relational functionalities are moved to NoSQL data store to lighten database load.

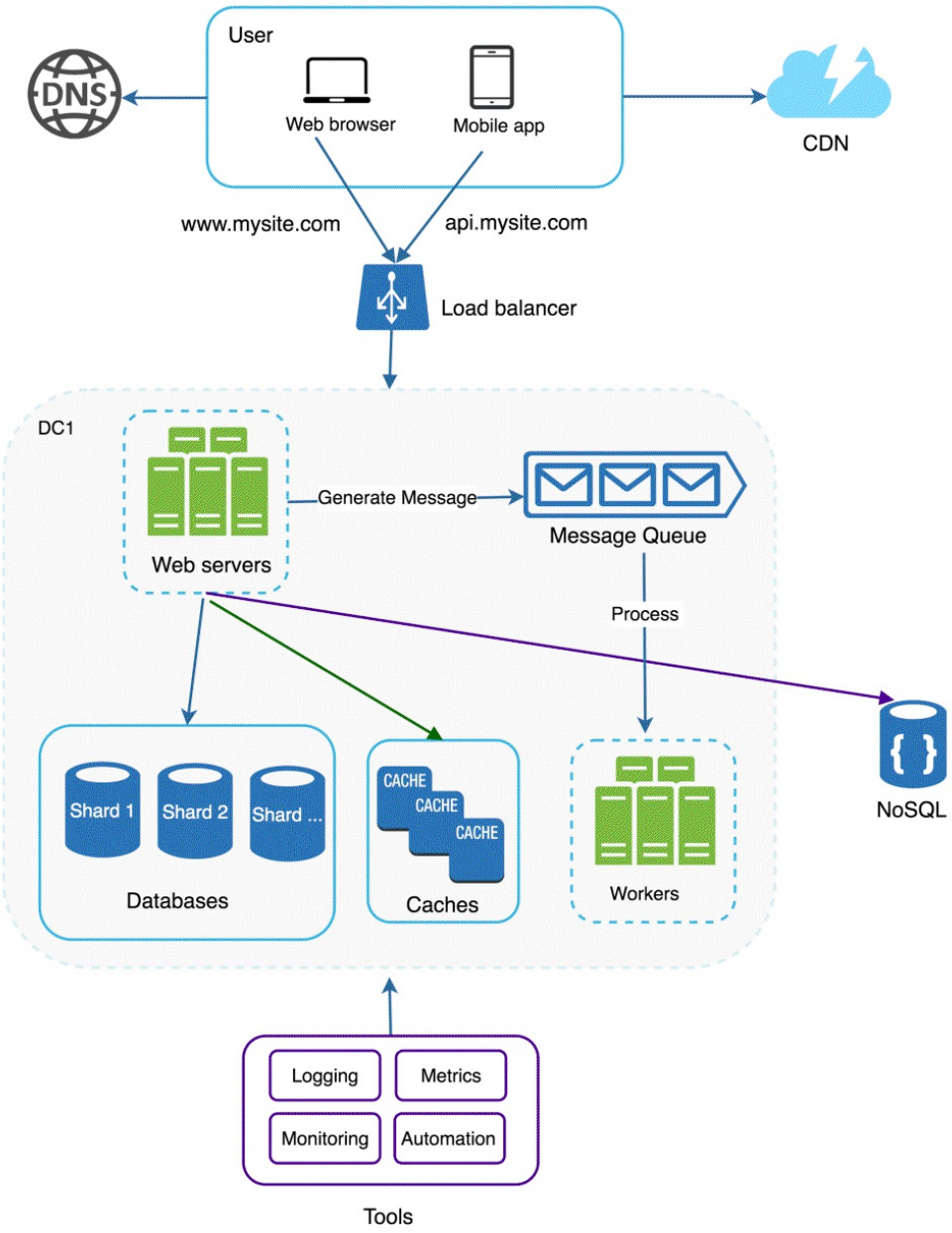
User
DNS
Web browser
Mobile app
CDN
www.mysite.com
api.mysite.com
Load balancer
DC1
Web servers
Generate Message
Message Queue
Process
Shard 1
Shard 2
Shard ...
Databases
CACHE
CACHE
CACHE
Caches
Workers
NoSQL
Logging
Metrics
Monitoring
Automation
Tools

Figure 22

Ten million users

Scaling is an iterative process. Iterating on top of what we've learnt in this chapter could get us pretty far. To scale beyond 10 million users, more fine tuning or new strategies are needed. For example, you might need to optimize your system further or decouple the system to even smaller services. All of the techniques learnt here should provide a good foundation to tackle new challenges. To conclude this chapter, let's take a quick summary of how we scale to 10 million users and beyond:

- Keep web tier stateless
- Build redundancy at every tier
- Cache data as much as you can
- Support multiple data centers
- Host static assets on CDN
- Scale your data tier by sharding
- Split tiers into individual services, like using messaging queue
- Monitor your system and use automation tools for your system

In the next few chapters, we will apply the techniques learnt here to solve some of the most commonly asked system design interview questions.

CHAPTER TWO: DESIGN CONSISTENT HASHING

When we talk about scaling horizontally in distributed systems, one very important topic to think about is distributing data efficiently and evenly across servers. To achieve this, consistent hashing is a very commonly used technique. Let's take an in-depth look of the problem we're trying to solve.

Context and problem

If you have a collection of *n* cache servers, a common way of load balancing across them is using the following process:

```
hash = hashFunc(key)
serverIndex = hash % serverSize
```

Figure 23

When the serverSize changes, all keys need to be re-mapped because the index is calculated by a modular operation.

Let's take a look of the following example in Figure 24. Keys are incrementing integers from 0 to 7. To fetch the server index where the key is hashed to, you can perform modular operation f(key) % 4. This works well when the system is stable, aka no servers are added or removed. In Figure 24, key0 and key4 are mapped to server 0, key1 and key5 are mapped to server1, so on and so forth.

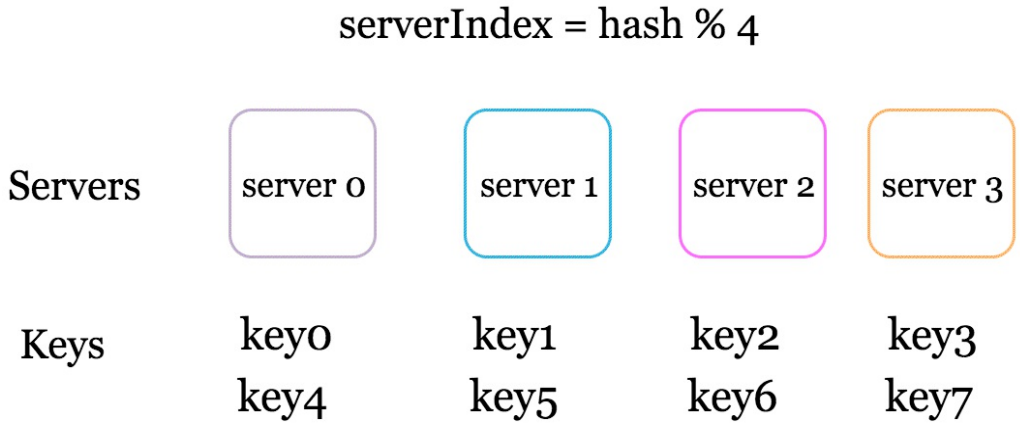

Figure 24

But what if a server, i.e. server 1, goes down? Using the same hash function, we get the same hash value for a key, but applying modular operation we get different server indexes than before. It's because the number of servers is reduced by one. Figure 25 explains it in more detail.

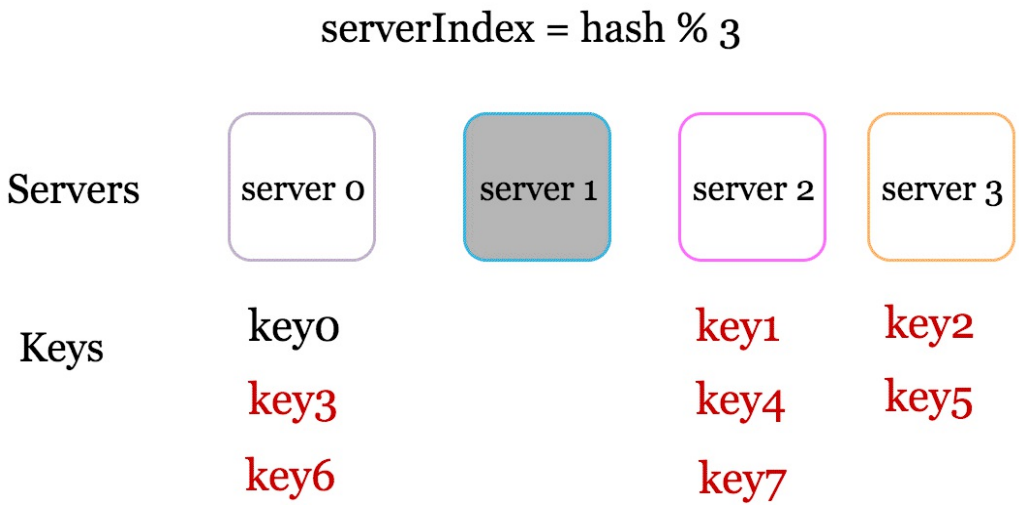

Figure 25

In Figure 25, almost all the keys need to be re-mapped (7 out of 8. Re-mapped keys are highlighted in red). This means that a cache client goes to the "wrong" server, the lookup operation misses. You essentially get a storm of recalculation as your cache contents shift from their old server to a new server. How can we do better? Consistent hashing comes to the rescue.

Consistent hashing

"Consistent hashing is a special kind of hashing such that when a hash table is re-sized and consistent hashing is used, only k/n keys need to be re-mapped on average, where k is the number of keys, and n is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped." [15]

Let's discuss this in more detail. Consider the output range of the hash function *f*, i.e. *x0, x1, x2, x3, ..., xn*. In the following example, SHA-1 is used as hash function *f*. The range for SHA-1 goes from 0 to 2^160.

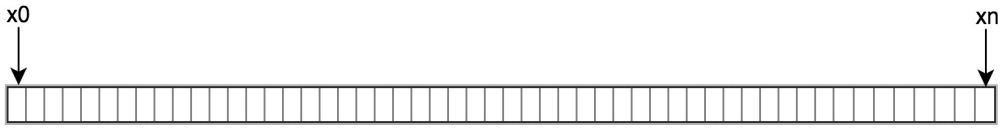

Figure 26

If we connect both ends, we end up with a ring as shown in Figure 27:

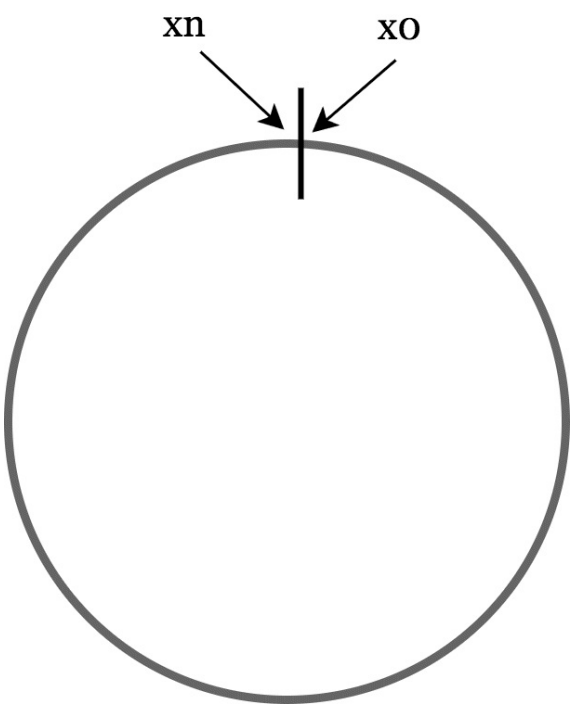

Figure 27

Using the same hash function *f*, we can map servers to corresponding positions in the ring. Please note that the same server is always mapped to the same position.

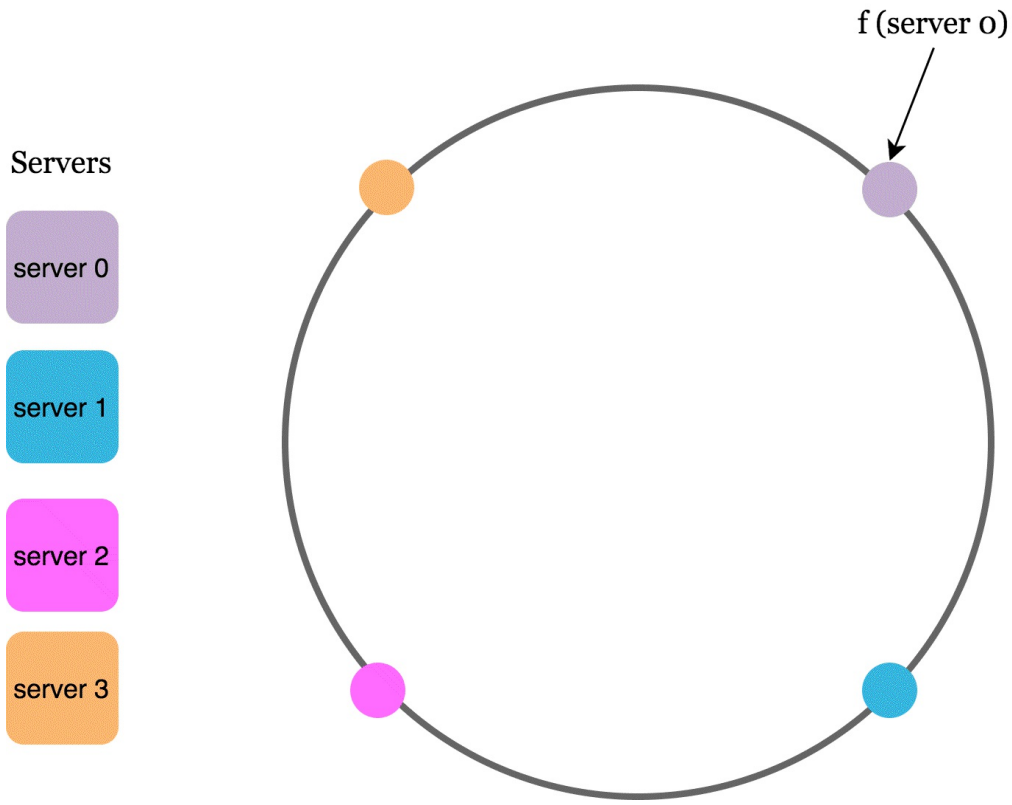

Figure 28

To determine which server a key lives in, we go clockwise from the key position in the ring till we find a server. Figure 29 is used to explain this process. Assume keys (*key0, key1, key2, key3*) are mapped to the black points. Going clockwise, *key0* is stored in *server 0*, *key1* is stored in *server 1*, *key2* is stored in *server 2* and *key3* is stored in *server 3*.

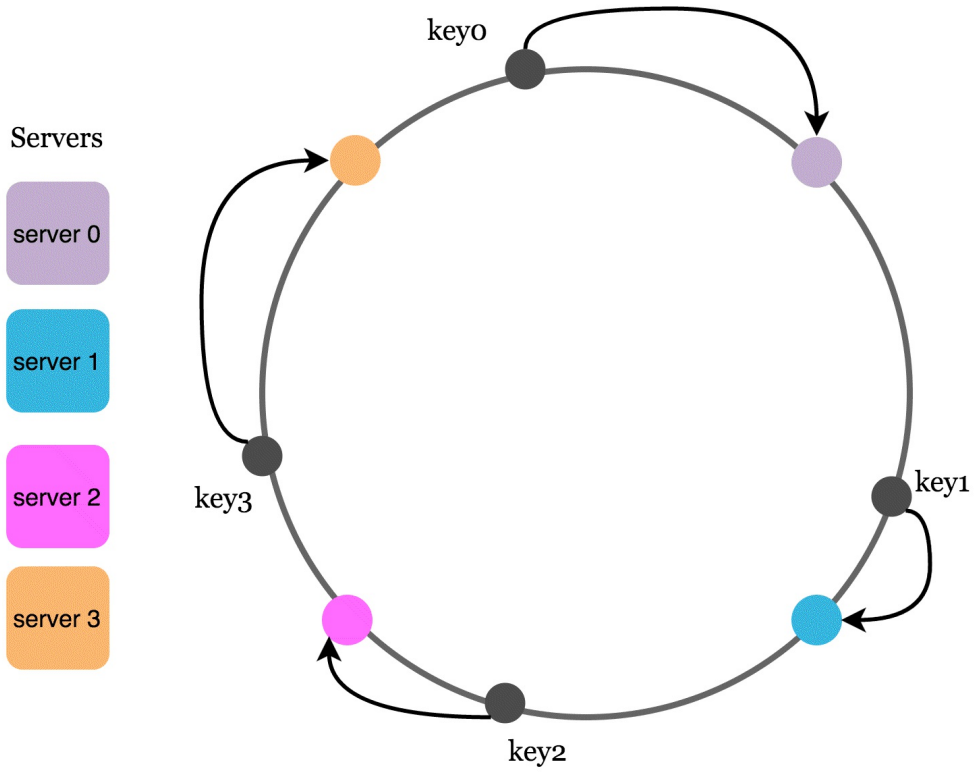

Figure 29

Using the logic described above, adding a new server to the ring does not mean that all keys need to be re-mapped. Only a fraction of keys need to be moved to a different server. In Figure 30, a new server *server 4* is added to the system. Only *key0* needs to be re-mapped. *key0* is mapped to *server 4* because *server 4* is the first server going clockwise from *key0*'s position in the ring. The other keys are mapped to the same positions.

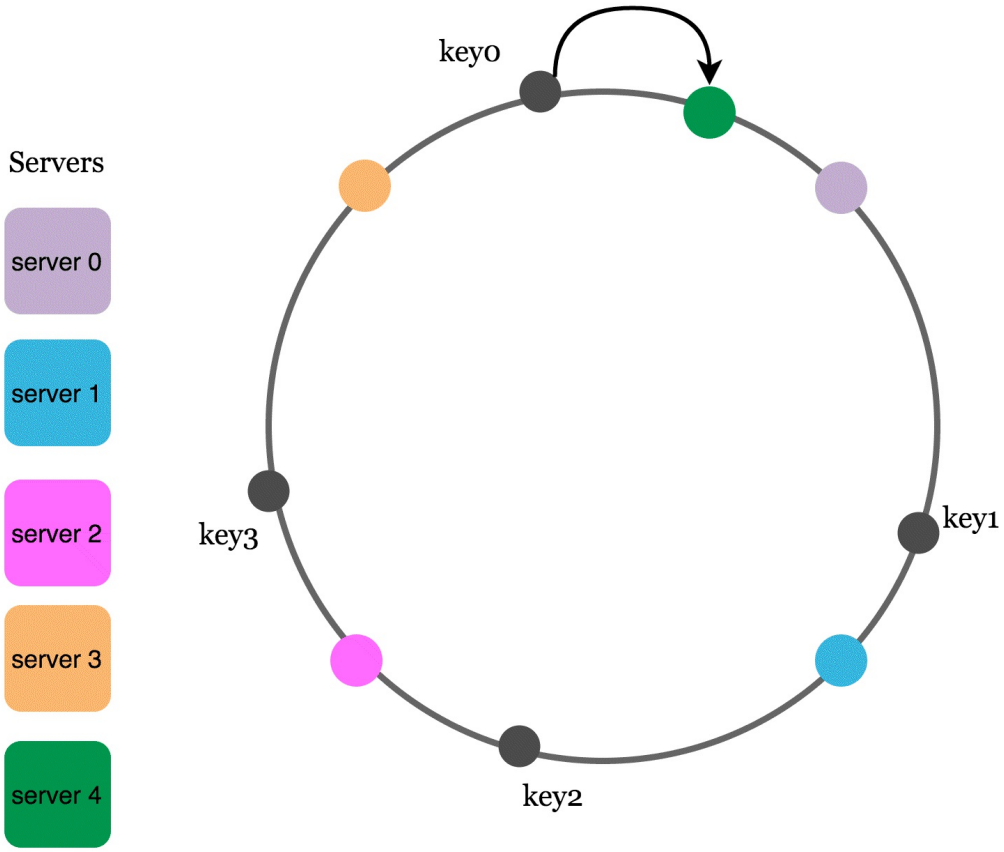

Figure 30

Similarly, only a fraction of the keys need to be re-mapped when a server leaves the ring. For example, after *server 1* goes offline, *key1* is re-mapped to *server 2* in Figure 31.

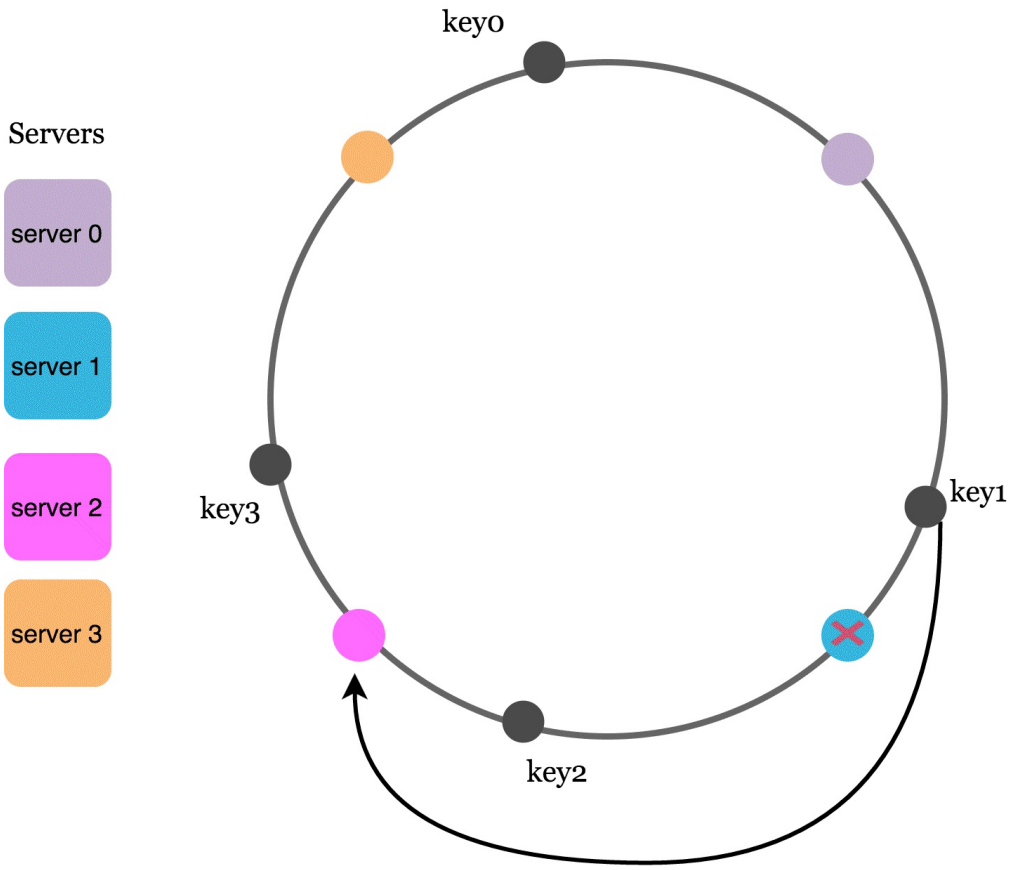

Figure 31

What we've talked is the essence of Consistent Hashing. The idea was presented in a paper by Karger et al. in 1997 [16]. The basic steps are:

1)- Map servers to the ring using a well distributed hash function.

2)- To find out which server a key lives in, go clockwise from the key position until the first server encountered in the ring is found.

This works well, but there is a problem. It’s impossible to keep the same size of partitions in the ring for all servers considering a server could be dynamically added or removed. The size of the partitions in the ring assigned to each server could be very small or fairly large. It is also possible to have a very non-uniform distribution of keys in the ring. For instance, if servers are

hashed to positions as listed in Figure 32, most of the keys would be stored in server 2. The solution to this problem is to introduce virtual nodes.

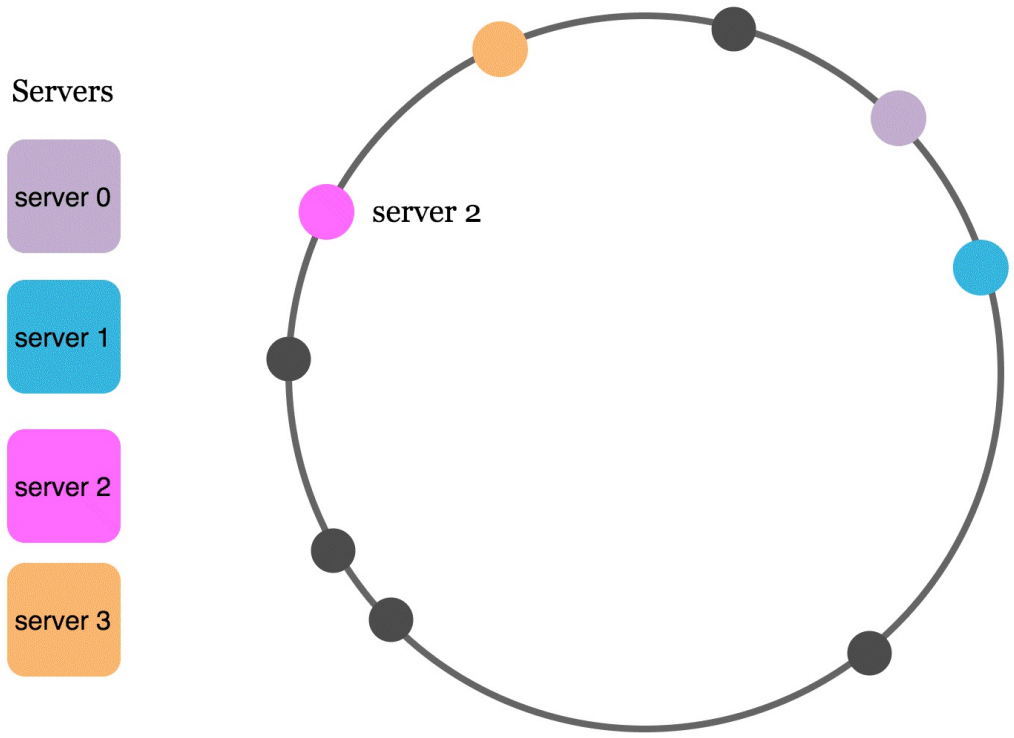

Figure 32

Virtual nodes

To distribute keys more evenly among servers, multiple virtual nodes on the ring for each server are created. With virtual nodes, each server is responsible for multiple partitions in the ring. For example mentioned in Figure 33, *server 0* and *server 1* both have 3 virtual nodes in the ring. 3 is arbitrary chosen here. In real applications, the number of virtual nodes is usually larger than 100.

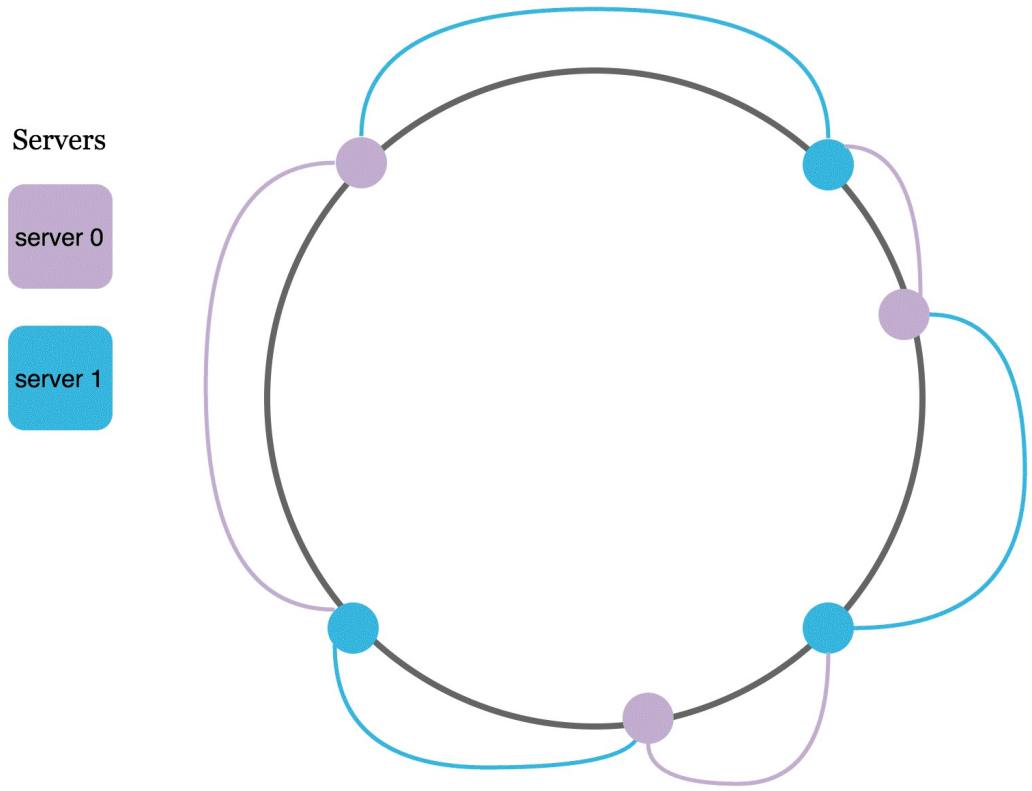

Figure 33

As the number of virtual nodes increase, the distribution of keys become more balanced. Examples mentioned in Figure 34 and Figure 35 are used to prove this point. In Figure 34, all four keys are stored in server 0. None of the keys are stored in *server 1*. In Figure 35, each server has 3 virtual nodes in the ring. In this case, keys are more evenly distributed among servers with each server stores 2 keys.

With enough virtual nodes (usually > 100), if a server is removed, the load handled by this server is evenly distributed across the remaining nodes in the ring. Similarly, when a physical server is added, it receives a roughly equivalent amount of data from the other nodes in the ring.

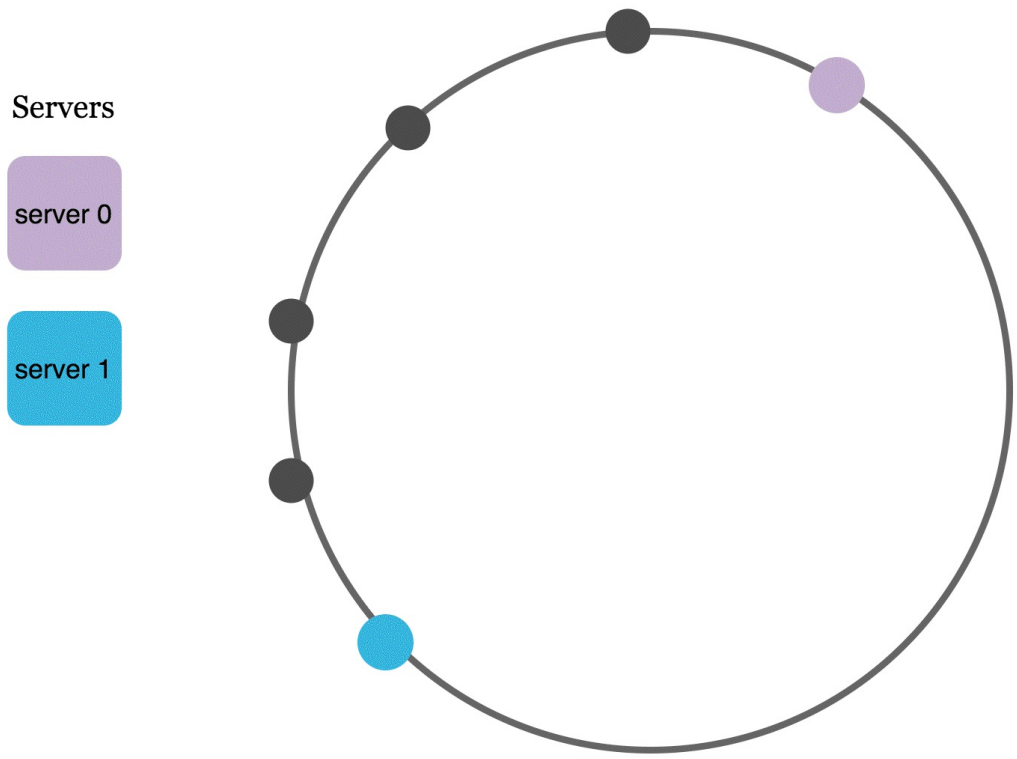

Figure 34

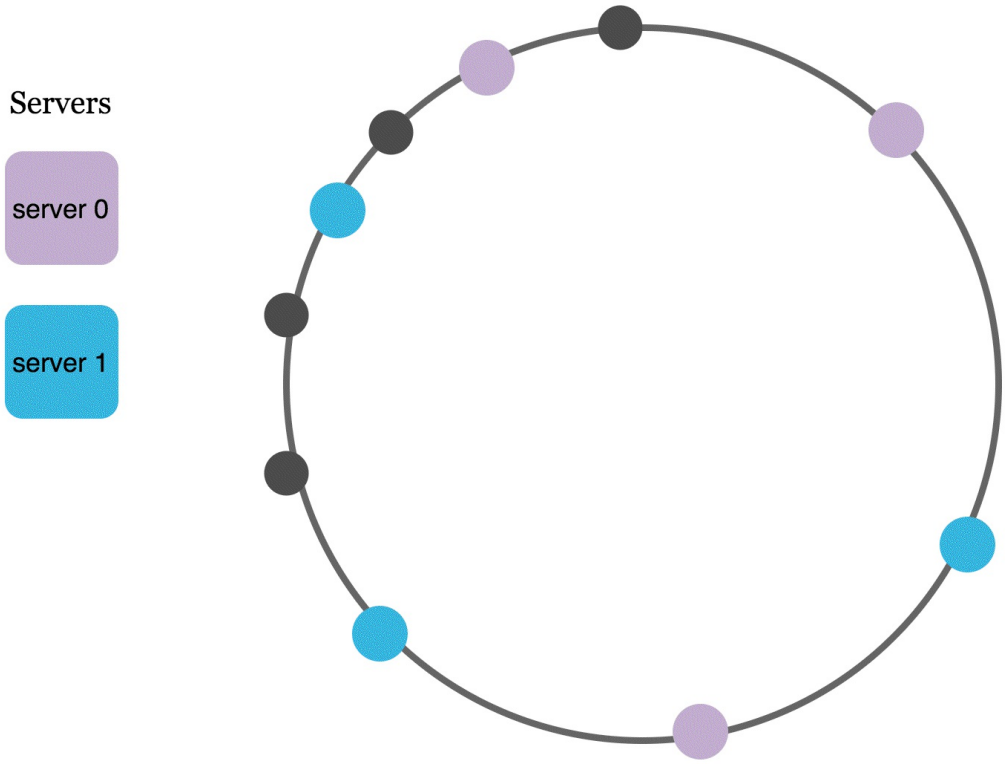

Figure 35

Implementation

In order for consistent hashing to be effective, it's important to have a hash function that uniformly distributes values. Cyclic Redundancy Code (CRC32) is used as the hash function in our implementation. Using CRC32 is completely optional, and you can use any hash algorithms that have good distribution (For example, SHA-1 or MD5). Given an input, the hash function returns a value between 0 to 2^32 – 1. The following code snippet shows how the hash function might be implemented.

```
private Long hash(String key) {
   CRC32 crc = new CRC32();
   crc.update(key.getBytes());
   return crc.getValue();
}
```

Physical servers are represented by "Server" class as follows.

```
class Server
{
   public String ipAddress;
   Server(String ipAddress) {
      this.ipAddress = ipAddress;
   }

   public String toString() {
      return ipAddress;
   }
}
```

The hash ring is represented as a sorted map of integers (SortedMap in java). The range of possible values are 0 to 2^32 - 1 (range of CRC32). When a server is added, multiple virtual nodes are added to the ring. The number of virtual nodes is configurable through *numberOfVirtualNodes*. The location of

each virtual node in the ring is calculated by hashing the server's name along with a numerical suffix. This implementation can avoid collision (two virtual nodes hashed to the same position) because the hash keys are always different.

```
private final SortedMap<Long, Server> hashRing;
private final int numberOfVirtualNodes;

public void add(Server server){
   for(int i = 0; i < numberOfVirtualNodes; i++){
      hashRing.put(hash(server.toString() + i), server);
   }
}
```

Similarly, to remove a server, we just need to remove all the virtual nodes from the sorted map.

```
public void remove(Server server){
   for(int i = 0; i < numberOfVirtualNodes; i++){
      hashRing.remove(hash(server.toString() + i));
   }
}
```

To find the server where a key is stored, aka our *get* method, go clockwise from the hash position, i.e. value of *hash(key)*, until we meet the first virtual node in the ring. This clockwise lookup process is simulated by using a tail map. Assume tail map is represented by *SortedMap<K,V> tailMap(K key)*. In this representation, tail map returns a view of the portion of this map whose keys are greater than or equal to the *key*. The following code snippet explains how the *get* method works.

```
public Server get(String key){
   if (hashRing.isEmpty()) {
      return null;
   }
   Long hashVal = hash(key);
   if (!hashRing.containsKey(hashVal)) {
```

```
        SortedMap<Long, Server> tailMap = hashRing.tailMap(hashVal);
        hashVal = tailMap.isEmpty() ? hashRing.firstKey() : tailMap.firstKey();
    }
    return hashRing.get(hashVal);
}
```

Below is the complete implementation in Java.

```
import java.util.*;
import java.util.zip.CRC32;

public class ConsistentHash
{
    private final SortedMap<Long, Server> hashRing;
    private final int numberOfVirtualNodes;

    /**
     * Constructor
     * @param numberOfVirtualNodes number of virtual nodes
     * @param servers  physical servers
     */
    public ConsistentHash(int numberOfVirtualNodes, Collection<Server> servers)
        {
        this.numberOfVirtualNodes = numberOfVirtualNodes;
        hashRing = new TreeMap<>();
        if(servers != null){
            for(Server n : servers){
                this.add(n);
            }
        }
    }

    /**
     * When a physical server is added, add numberOfVirtualNodes virtual nodes to the hashRing.
     * @param server
     */
    public void add(Server server)
        {
        for(int i = 0; i < numberOfVirtualNodes; i++){
            hashRing.put(hash(server.toString() + i), server);
        }
    }
    /**
     * When a physical server is removed, remove all of the virtual nodes
     * @param server server to remove
     */
```

```
public void remove(Server server)
    {
    for(int i = 0; i < numberOfVirtualNodes; i++){
        hashRing.remove(hash(server.toString() + i));
    }
}
/**
 * Get the physical server a key is mapped to.
 * @param key
 * @return the server a key is mapped to
 */
public Server get(String key)
    {
    if (hashRing.isEmpty()) {
        return null;
    }
    Long hashVal = hash(key);
    if (!hashRing.containsKey(hashVal)) {
        SortedMap<Long, Server> tailMap = hashRing.tailMap(hashVal);
        hashVal = tailMap.isEmpty() ? hashRing.firstKey() : tailMap.firstKey();
    }
    return hashRing.get(hashVal);
}

/**
 *
 * @param key key to hash
 * @return hash value. range 0 ~ 2^32 - 1
 */
private Long hash(String key)
    {
    CRC32 crc = new CRC32();
    crc.update(key.getBytes());
    return crc.getValue();
}

// Usage examples: add server, remove server, get the server based on a key
public static void main(String[] args)
    {
    // add two physical servers. Each server has 200 virtual nodes
    List<Server> servers = new LinkedList<>();
    servers.add(new Server("10.0.0.1"));
    servers.add(new Server("10.0.0.2"));
    int numberOfVirtualNodes = 200;
    ConsistentHash consistentHashObj = new ConsistentHash(numberOfVirtualNodes, servers);
```

```
        // add a new server
        Server newServer = new Server("10.0.0.3");
        consistentHashObj.add(newServer);

        consistentHashObj.hash("key0");
        // find out where server "key0" is mapped to.
        System.out.println(consistentHashObj.get("key0")); //return 10.0.0.3
        // remove a server
        consistentHashObj.remove(newServer);
        System.out.println(consistentHashObj.get("key0")); //return 10.0.0.2
    }
}

class Server
{
    public String ipAddress;
    Server(String ipAddress) {
        this.ipAddress = ipAddress;
    }

    public String toString() {
        return ipAddress;
    }
}
```

Usage

You might be curious about how consistent hashing is used in production systems. Here are some of the examples where it is used [15]:

- Couchbase automated data partitioning
- Partitioning component of Amazon's DynamoDB [19]
- Data partitioning in Apache Cassandra
- Akamai content delivery network

CHAPTER THREE: DESIGN A KEY-VALUE STORE

You can think of a key-value store as a very simple form of a database which supports the following two operations:

- **put**(key, value) // insert “value” associated with “key”

- **get**(key) // get “value” associated with “key”

Single server key-value store

Building a key-value store in a single server is easy. A common approach is to store key-value pairs in a hash table. Using a hash table means everything is stored in memory. A potential problem is that fitting everything in memory may not be possible, so the following optimizations are needed.

• Compress your data.
• Store your data in disk and only put frequently used data in hash table. In this case, the hash table acts like a cache system.

Even with the above optimizations, a single server can reach its capacity very quickly due to its memory limit. Implementing a distributed key-value store is required to support big data.

Distributed key-value store

Distributed key-value store is also called distributed hash table. The main idea is to distribute key-value pairs across many servers. When designing a distributed system, it's important to understand CAP (**C**onsistency, **A**vailability, **P**artition Tolerance) theorem. The CAP theorem should be taken into consideration when we make high level architectural decisions.

CAP theorem

CAP theorem states that it is impossible for a distributed system to simultaneously provide more than two out of the three following guarantees: consistency, availability and partition tolerance. First, let's establish a few definitions.

Consistency: All the replicas are in sync and maintain the same state of any given object at any given point of time.

Availability: Every request receives a response. There is no guarantee that the response contains the most recent data.

Partition Tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between servers. [17]

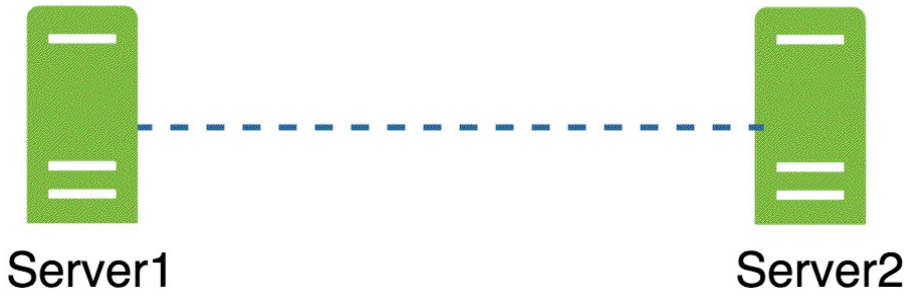

Figure 36

Figure 36 illustrates the ideal case that network partition never occurs. Data written to *server1* would be automatically replicated to *server2* and vice versa. Both consistency and availability are achieved.

However, given networks aren't reliable in the real world, the distributed system must tolerant partitions. When partitions occur, you have to choose between consistency and availability.

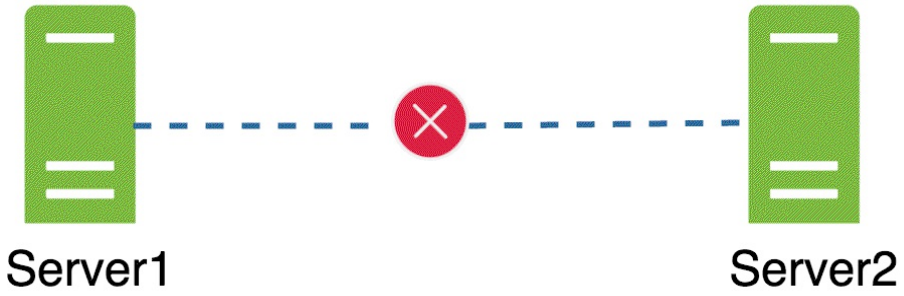

Figure 37

In Figure 37, due to network partition, *server1* cannot communicate with *server2*, aka data replication is broken. If clients write data to *server1*, *server1* cannot propagate this change to *server2*.

If we choose consistency over availability, we'll have to block all write operations to *server1* and *server2* to avoid data inconsistency between those two servers, which by definition makes the system unavailable. Bank systems usually have extremely high requirement for consistency. It's crucial for a banking system to always show the most up-to-date balance info. If inconsistency occurred due to network partition (some replicas might have stale data), the bank system might return an error before the inconsistency issue is resolved.

On the other hand, if availability is what we desire, return the most recent version of data on reads, which could be stale if *server1* and *server2* are out of sync. For writes, both servers keep accepting writes, and data will be processed/synced later when the partition is resolved.

Choose availability over consistency when your system allows for data synchronization at a later time. Availability is also a compelling option when the system needs to be functional in spite of error or inconsistency. Let's take the Amazon's shopping cart for example. If the most recent state of the cart is unavailable and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. [18]

CAP theorem is frequently misunderstood as if one has to choose to abandon one of the three rules at all times. In fact, the choice is really between consistency and availability when partition happens. If no partition occurs, no trade-off has to be made.

Recognizing which CAP guarantees your business really needs is the first step of building any distributed system.

Design Goals

Our design aims to achieve the following:

• Ability to store “big data” (>1 terabytes)
• High availability: always respond quickly, even during failures.
• Scalability. 1) The system can be scaled to support thousands of servers easily. 2) Addition/deletion of servers should be easy.
• Heterogeneity. The work distribution must be proportional to the capabilities of individual servers. For example, more work should be distributed to more powerful servers. Less powerful servers handle less work load. This is

essential because not all servers have the same power and capacity.
• Tunable tradeoffs between consistency and latency.
• Low latency read and write operations. (On average < 10ms for reads, < 20ms for writes).
• Comprehensible conflict resolution.
• Robust failure detection and resolution techniques.

System Architecture

To achieve the design goals mentioned above, the system will be complex. We won't be able to cover every single detail of the system, but we'll discuss the core components/techniques. Please note that the design mentioned below is largely based on two popular key-value store systems: Dynamo [18] and Cassandra [19].

Data Partition

For "big data", it's not feasible to fit the complete data set in a single server. One design goal is to distribute data across multiple servers evenly. The other is to minimize data movement when nodes are added or removed. Consistent hashing mentioned in Chapter 2 can help to achieve those two goals.

Let's revisit how consistent hashing works in high level. First, servers are placed on a hash ring. Next, a key is hashed into the same ring, and it is stored in the first server that it encounters while traveling in clockwise direction. For example, in Figure 38, 8 servers, represented by *s0, s1, ..., s7*, are placed on the hash ring. *key0* is stored in *s1* using consistent hashing. Virtual nodes are not placed on the figure due to space constraint.

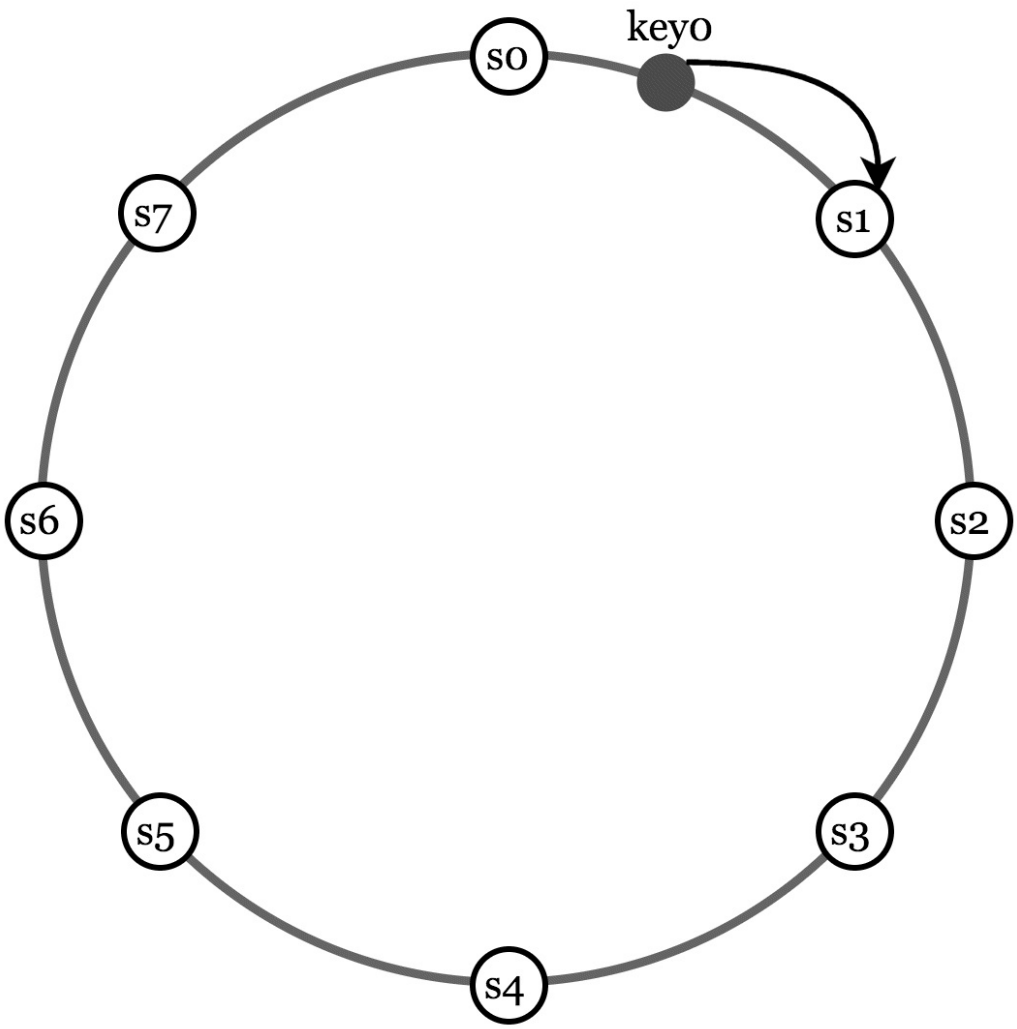

Figure 38

Using consistent hashing to partition data has the following two advantages:

High scalability. Servers could be added and removed seamlessly.

Heterogeneity. We can make the number of virtual nodes proportional to the server capacity.

Data Replication

To achieve high availability and reliability, data need to be replicated

asynchronously over *N* servers, where *N* is a configurable parameter. Usually, the total number of servers in the system is bigger than *N*. If one server fails, there's still *(N - 1)* working copies of the data in the system. Those *N* servers are chosen by a method very similar to consistent hashing. It works as follows: after a key is mapped to a position on the hash ring, walk clockwise from that position and choose the first *N* servers on the ring to store data copies. In Figure 39 (*N = 3*), using the replication method mentioned above, keys fall in the range of *(s0, s1)* are replicated at three nodes: *s1*, *s2* and *s3*. Here is a concrete example: *key0* is replicated at *s1, s2* and *s3*.

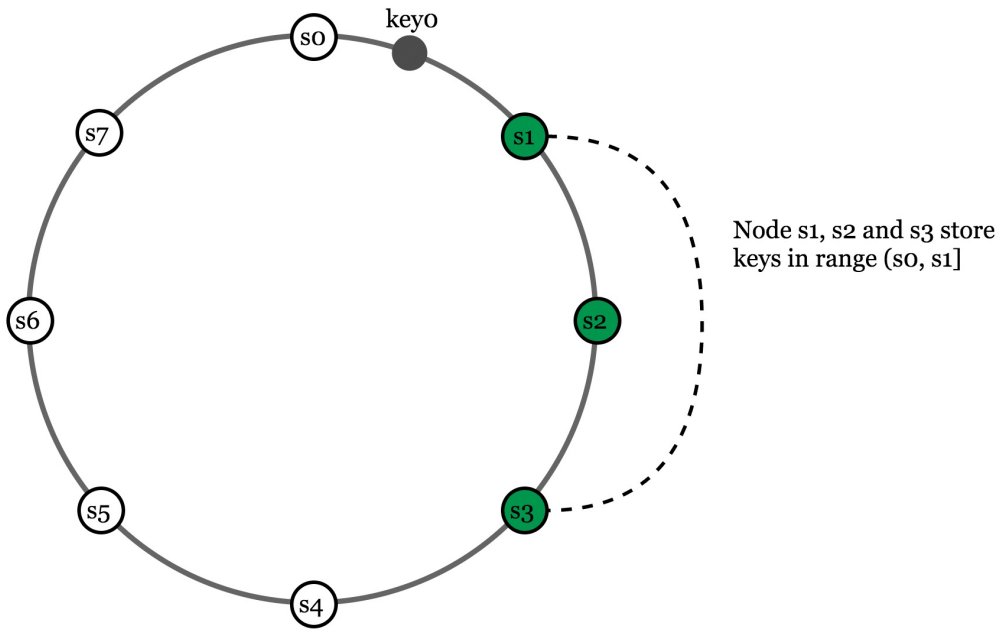

Figure 39

Because of the use of virtual nodes, it's possible that the first *N* nodes on the ring are owned by few than *N* physical servers. To avoid this, only unique servers are chosen while performing the clockwise walk algorithm. Meanwhile, taking node or data center failures into account, replicas are placed on distinct data centers. This is because nodes in the same data center often fail at the same time due to power outage, network issues, etc. In

production systems, two or three replicas per data center is a common setup [20].

Consistency

Since data are stored on multiple replicas, we need to ensure they are synchronized across replicas. Quorum consensus is used to guarantee consistency for both read and write operations. Let's first establish a few definitions.

N = The number of replicas
W = A write quorum of size W. In order for a write operation to be considered as successful, write operation needs to wait for acknowledgements from at least *W* replicas.
R = A read quorum of size *R*. In order for a read operation to be considered as successful, read operation needs to wait for responses from at least *R* replicas.

Consider the following example shown in Figure 40 with *N = 3*.

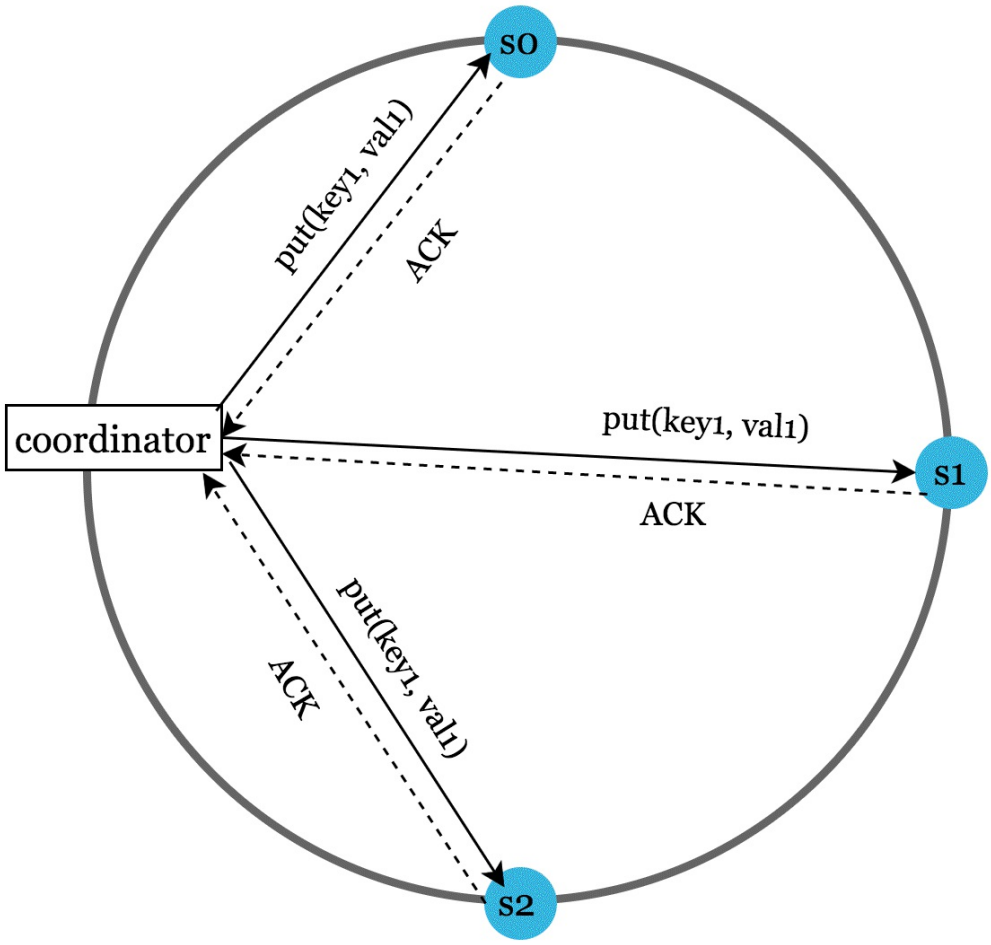

Figure 40 (ACK = acknowledgement)

W = 1 doesn't mean data are written to only one server. For instance, with the configuration in Figure 40, data are replicated at *s0*, *s1* and *s2*. What *W = 1* means is that the coordinator (a server that routes requests) needs to receive at least one acknowledge before the operation is considered to be successful. For instance, if we already get an acknowledge from *s1*, we no longer need to wait for *s0* and *s2*.

The configuration of *W, R* and *N* is a typical tradeoff between latency and consistency. If *W = 1* or *R = 1*, an operation will be returned pretty quickly because a coordinator only needs to wait for a response from any of the

replicas. If *W* or *R* > *1*, the system has better consistency but the query will be slower because the coordinator has to wait for the slowest replica.

If *W*+*R* > *N*, strong consistency can be guaranteed. This is because there are always overlaps between write nodes and read nodes. We could use those overlapped nodes to check consistency. To illustrate this, let's review the following example (Figure 41).

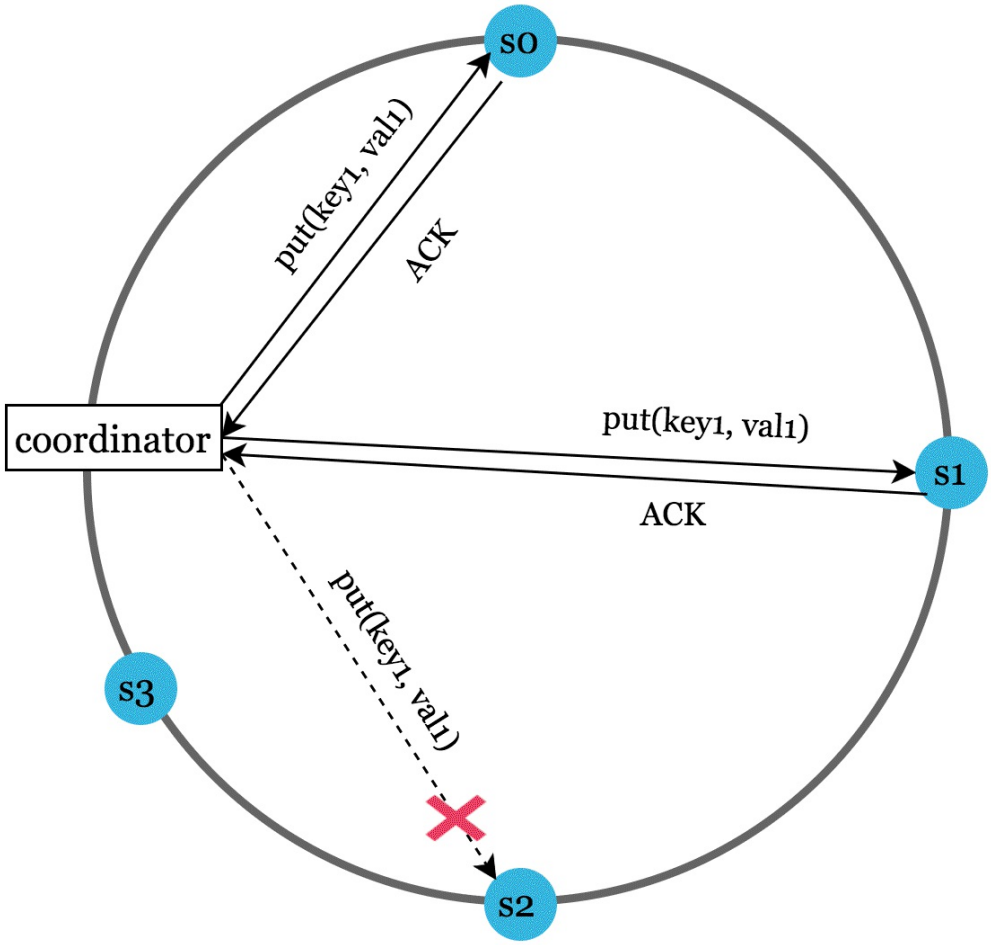

Figure 41

Assume *N* = *3*, *W* = *2*, and *R* = *2* in Figure 41, i.e. *R* + *W* > *N*. Assume operation put(key1, val1) tries to write data to all three replicas: *s0, s1* and *s2*. *s0* and *s1* return success while s2 returns failure. Because *W* = *2*, this write

operation is considered as successful. If *s2* is chosen for get(key1) operation, the system can discover inconsistency by comparing the value in *s2* with the value in either *s0* or *s1*.

If $R + W <= N$, consistency cannot be guaranteed. For instance, assume $N = 3$, $W = 2$, and $R = 1$, i.e. $R + W = N$ in Figure 41. After put(key1, val1) operation, *s0* and *s1* are updated with the latest value while *s2* still contains stale data because the put operation failed. If the subsequent *get* operation is routed to *s2*, *s2* will return stale data immediately because it doesn't need to wait for acknowledge from other replicas.

How to configure *N, W*, and *R* depends on use case requirements that need to be optimized. Here are some of the possible configuration setups:

If $R = 1$ and $W = N$, we optimize for fast read.
If W = 1 and R = N, we optimize for fast write. Durability is not guaranteed in the presence of failures.
If $N = 3$, $W = 2$ and $R = 2$, i.e. $W + R > N$, we focus on fault tolerance.
If $W = N$, we focus on consistency on write. However, this setup may cause slow response and decrease the write success rate as it needs to wait for acknowledgements from all *N* servers.

Next, another important factor to consider for designing a key-value store is consistency models. A wide spectrum of possible consistency models exists. However, knowing the following consistency models should be sufficient for interviews:

Strong consistency. Any read on a data item returns value corresponding to result of the most recent write on that data item (regardless of where the write occurred). A client will never see out-of-date data.

Weak consistency. Every replica will see every update, but possibly in different orders.
Eventual consistency. Given enough time, all updates will propagate through the system.

How to choose between the above consistency models is use case specific. Strong consistency is achieved by forcing a replica not to accept new writes until every replica has agreed on current write. This approach is too heavy weight for highly available systems because it basically blocks new writes. DynamoDB and Cassandra adopt eventual consistency. They allow inconsistent values from concurrent writes to enter the system and force the client which reads the values to reconcile. The next section explains how reconciliation works.

Versioning

Versioning means treating each modification of data as a new immutable version of the data. Before we talk about versioning, let's consider the following sequence of actions that could cause data conflict first:

```
# two clients simultaneously fetch the same value
[client 1] get(user1) => {"name":"john"}
[client 2] get(user1) => {"name":"john"}

# client 1 modifies the name and performs a put operation
[client 1] put(user1, {"name":"johnSanFrancisco"})

# client 2 modifies the email and performs a put operation
[client 2] put(user1, {"name":"johnNewYork"})
```

We now have the following conflicting versions:
Original value in the database: {"name":"john"}
From client1: {"name":"johnSanFrancisco"}
From client 2: {"name":"johnNewYork"}

In this example, the original value could be ignored because the modifications were based on it. However, there is no clear way to resolve the conflict for the last two versions. To resolve this, we need a versioning system that allows us to detect overwrites and throw away the old version, but also allows us to detect conflicts and let the client reconcile. DynamoDB [18] uses vector clocks to solve this problem. Let's examine how vector clocks work.

A vector clock is basically a *{server:counter}* pair. For each writing server, it keeps a counter and allows us to see if one version precedes, succeeds or in conflict with the other.

Using vector clock, you can tell that a version X is an ancestor (i.e. no conflict) of version Y if the counters for each participant in the vector clock of Y is greater than or equal to the vector clock of X. For example, vector clock *{s0:1,s1:1}* is an ancestor of *{s0:1,s1:2}* and therefore there is not conflict.

Similarly, you can tell that a version X is a sibling (i.e. conflict exists) of Y if there exists any participant in Y's vector clock who has a counter that is less than his corresponding counter in X. For example, those two vector clocks are conflicting: *{s0:1,s1:2}* and *{s0:2,s1:1}*.

Figure 42 shows how vector clocks work in more detail.

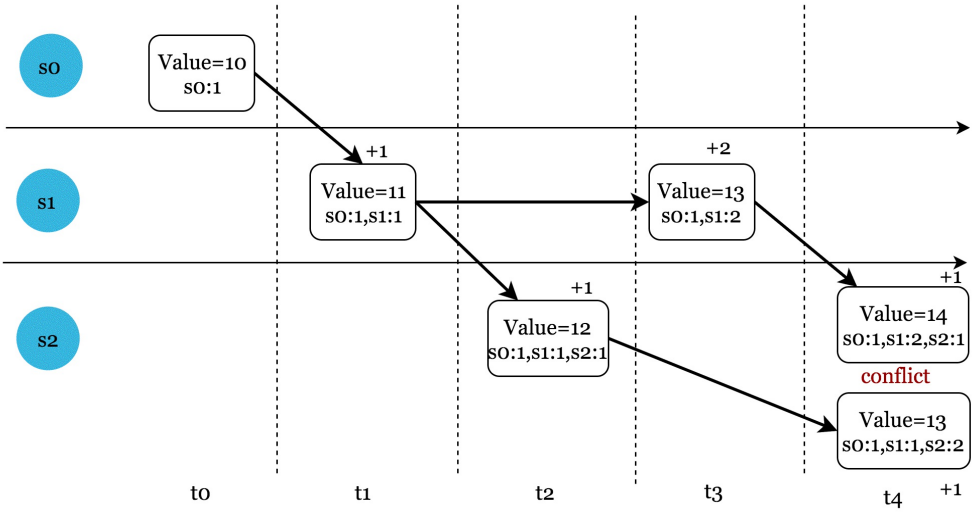

Figure 42

At *t0*, server *s0* updates "Value" to 10. This is the first time *s0* updates this value, so it returns counter 1, i.e. *{s0:1}* using vector clock notation.

At *t1*, *s1* reads the data (written by *s0*) and updates "Value" to 11. *s1* adds it's own vector clock *{s1:1}* and appends it, so the vector clock becomes *{s0:1,s1:1}*.

At *t2*, *s2* reads the data (written by *s1*) and updates "Value" to 12. *s2* adds it's own vector clock *{s2:1}* and appends it, so the vector clock becomes *{s0:1,s1:1,s2:1}*.

At *t3*, *s1* reads the data (written by s1) and updates "Value" to 13. The vector clock becomes *{s0:1,s1:2}*. The counter is 2 because it's the second time *s1* updates "Value".

At *t4*, *s2* reads the data and tries to apply +1 operation. However, when *s2* reads the data, s2 gets conflicted values 13 (stored in s1) and 12 (stored in s2). If you apply +1 operation directly as shown in Figure 42, it would result in conflicted values 14 and 13, with vector clocks equal to {s0:1,s1:2,s2:1} and

{s0:1,s1:1,s2:2} respectively. In this scenario, client could use vector clocks to detect and resolve conflict. What resolution strategies to use is up to the client. For instance, client could use a resolution strategy that largest value wins.

There are two downsides of using vector locks. First, it adds complexity to the client because the client needs to implement certain logic to resolve conflicts. Second, the *{server:counter}* pairs in vector clock could grow rapidly. One potential fix for this problem is to set a threshold for the length and if the number of *{server:counter}* pairs exceeds the limit, the oldest one will be removed. This can lead to inefficiencies in reconciliation because the descendant relationship cannot be determined accurately. However, dynamo paper [18] says Amazon has not yet encountered this problem in production and therefore it is an acceptable solution.

Failure Scenarios

In any large systems, it's common that a small number of servers or network components fail at any given time. Knowing how to handle failure scenarios is very important and should be part of the system design process. First, let's learn some techniques to detect failures. Next, we'll go over common failure scenarios and failure resolution strategies.

Failure Detection

In a distributed system, it's not sufficient to say a server is down because another server says so. It needs at least two independent sources of information to mark a server down. It's because the other server might be the one actually has the problem. But if multiple servers say a server is down, you can say with high confidence that this server is down. How does one server know whether other servers are down or become alive?

A straightforward solution is through all-to-all multicasting. It's not efficient when there are lots of servers in the system.

A better solution is to use decentralized failure detection methods like gossip protocol for inter-node communication. Gossip protocol works like this: Every second, each server sends a message to a set of random servers, who in turn propagate to another set of servers. After certain period of time, data propagate to every server in the system [21]. The gossip process tracks state from other servers both directly (servers gossiping directly to it) and indirectly (non-firsthand info). During gossip exchanges, every server maintains global server status info about other servers in the system. Failure detection is determined by checking gossip state and history.

Handling transient failures

After failures have been detected through gossip protocol, the system deploys certain mechanisms to achieve high availability. In the strict quorum approach, the write operation could be blocked because the system does not receive acknowledgements from a pre-defined amount of servers, as illustrated in quorum consensus. Therefore, the system wouldn't be available during server or network failures. For instance in Figure 43, if $W = 3$, operation put(key1, val1) fails because it couldn't receive acknowledgement from *s2* (*s2* is down).

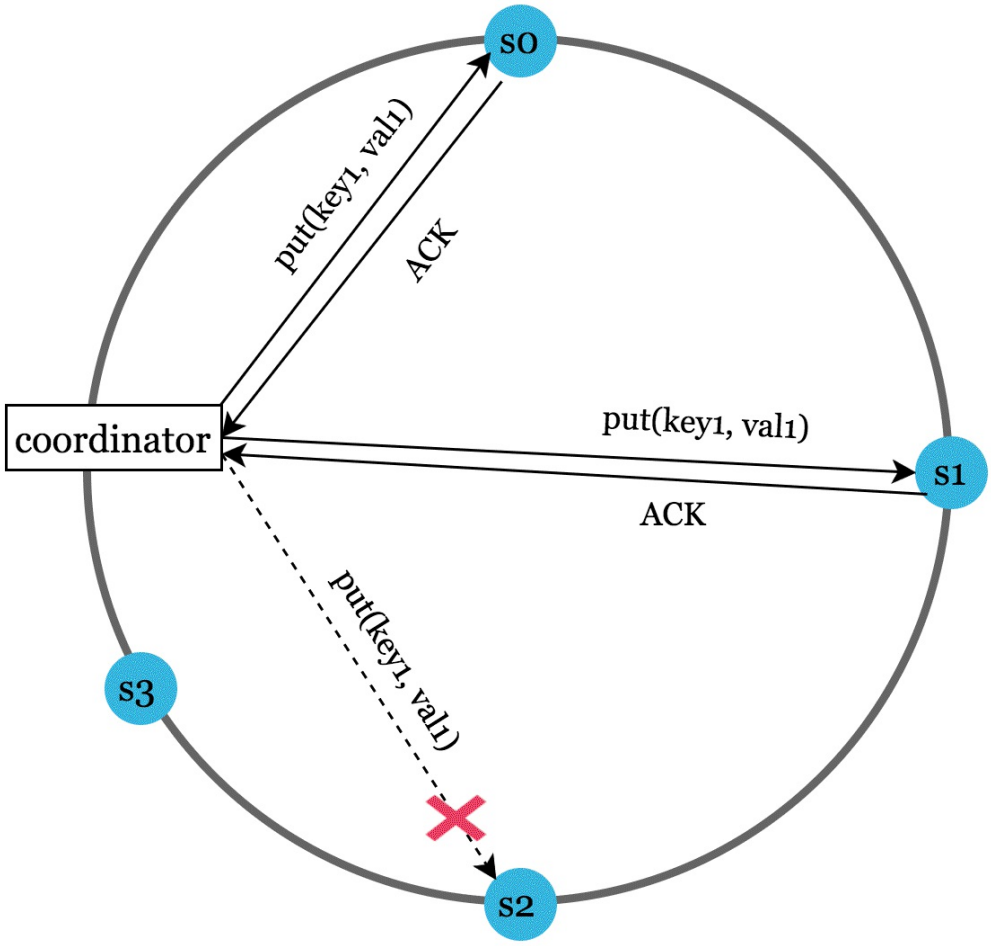

Figure 43

To improve write availability, a technique "sloppy quorum" [18] is used. Instead of strictly enforcing the quorum requirement, the system chooses the first *W* healthy servers for writes and *R* healthy servers for read on the hash ring.

To make the system always writable, a technique called "hinted handoff" is used. If a server where data is written to is unavailable due to network or server failures, another server will process that write operation temporarily. Then when the down server comes back up, the change will be pushed back to make the data consistent. For instance, since *s2* is unavailable in Figure 43,

reads and writes will be handled by *s3* instead temporarily. When *s2* comes back online, *s3* will hand the data back to *s2*.

Handling permanent failures

Hinted handoff is used to handle transient failures. What if a hinted replica becomes permanently unavailable? To handle such situation, we need to implement an anti-entropy protocol to keep the replicas in sync. Anti-entropy means comparing all the replicas of each piece of data and updating each replica to the newest version. A Merkle Tree is used for the purpose of inconsistency detection and minimizing the amount of data transferred.

According to Wikipedia: "A hash tree or Merkle tree is a tree in which every non-leaf node is labelled with the hash of the labels or values (in case of leaves) of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures."

The following steps show how to build a Merkle tree. Assume key space is from 1 to 12. Red boxes indicate inconsistency.

Step 1: Divide keys into four buckets as shown in Figure 44.

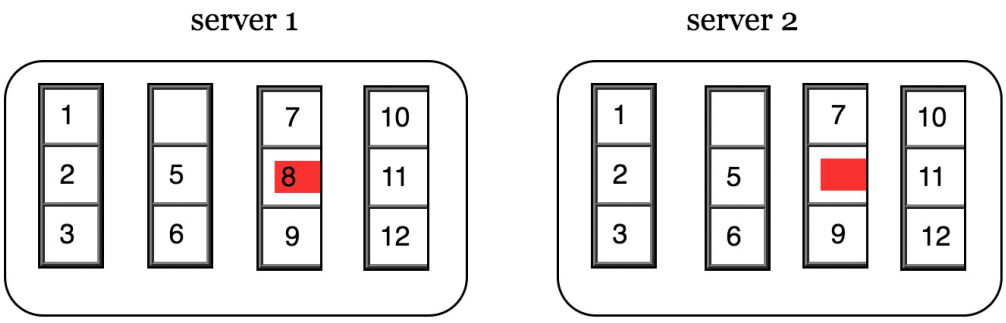

Figure 44

Step 2: Once the buckets are created, hash each key in a bucket using a uniform

hashing method (Figure 45).

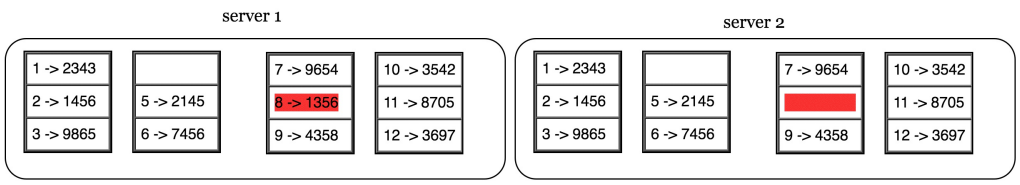

Figure 45

Step 3: Create a single hash node per bucket (Figure 46)

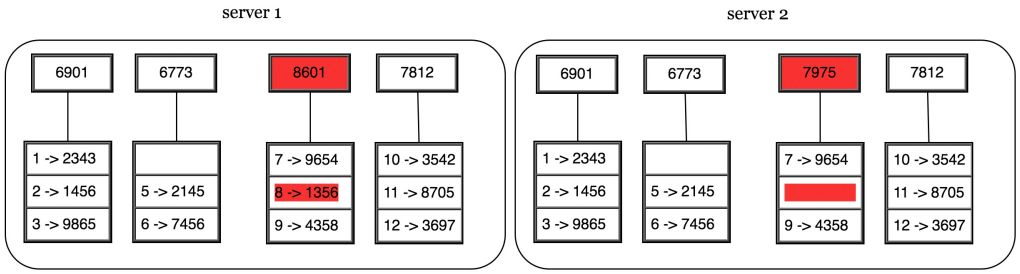

Figure 46

Step 4: Build the tree upwards till root by calculating hashes of children (Figure 47).

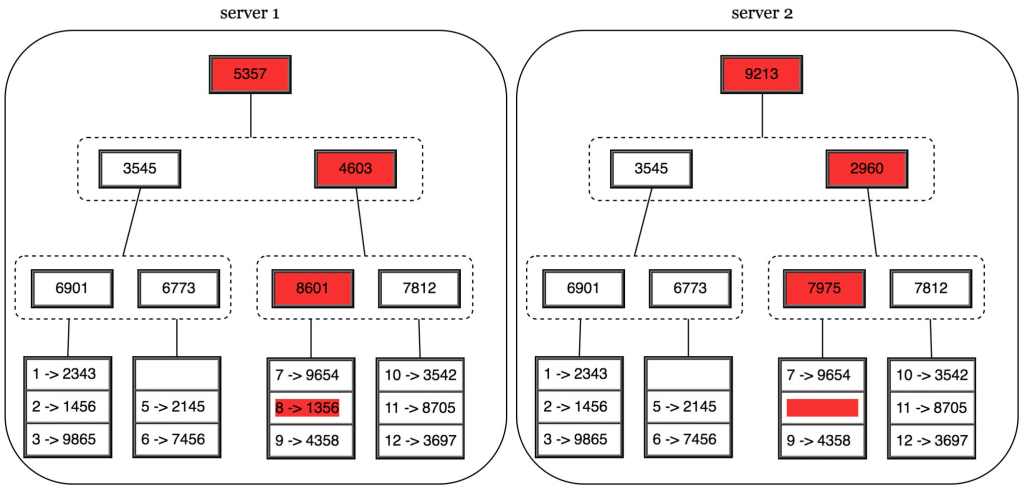

Figure 47

To compare two Merkle trees, start by comparing the root hashes. If root hashes match, then both servers have the same data. If root hashes disagree, then the left child hashes are compared followed next by right child hashes. You can traverse the tree to find out which buckets are not synchronized and synchronize those buckets only.

Using Merkle trees, the amount of data needs to be synchronized is proportional to the differences between two replicas, not the amount of data they contain. In production systems, the bucket size is generally quite big. For instance, a possible configuration is one million buckets for one billion keys. Each bucket only contains 1000 keys, which is a pretty small number. Synchronizing data in a bucket across servers are fast and efficient.

Handling data center outage

Data center outage could happen due to power outage, network outage, nature disaster, etc. To build a system that's capable of handling data center outage, it's important to replicate data across multiple data centers. This way, even if a data center is completely down, we could avoid data outage.

Summary

The following table (Table 1) summaries the goals we want to achieve and corresponding techniques when designing a distributed key-value store.

Goal/Problems	Technique
Ability to store "big data"	Use consistent hashing to spread load across servers.

High availability reads	Data replication Multi-datacenter replication
Highly available writes	Versioning with vector clock
Dataset partition	Consistent Hashing
Incremental scalability	Consistent Hashing
Heterogeneity	Consistent Hashing
Tunable consistency	Quorum consensus
Low latency	Set R to small number like 1 in the quorum system
Handling temporary failures	Sloppy quorum and hinted handoff
Handling permanent failures	Merkle tree
Handling data center outage	Cross-datacenter replication

Table 1

CHAPTER FOUR: DESIGN A URL SHORTENER

Problem

How to design a URL shortener like tinyurl is a very frequently asked system design question. Assume an interviewer gives you the following task:

"Please design a URL shortening service like tinyurl, a web service that provides short aliases for redirection of long URLs."

Clarify and scope out the task

System design questions are usually intentionally left open ended, so you need to ask questions and find out requirements in order to design the proper system. Let's start asking the interviewer some questions to clarify the task.

Question: Could you give me an example that shows how a URL shortener works?
Answer: For example, URL https://en.wikipedia.org/wiki/Systems_design is the original URL. Your service creates an alias with shorter length for it – http://tinyurl.com/zn9edcu. If you click the alias, it'll redirect you to the original URL.

Question: What is the traffic volume?
Answer: 10 million URLs are generated per day.

Question: How long is the shortened URL?
Answer: As short as possible.

Question: What characters are allowed in the shortened URL?
Answer: Shortened URL can be combinations of numbers (0-9) and characters (a-z, A-Z).

Question: Can shortened URLs be deleted or updated?
Answer: For simplicity's sake, let's assume shortened URLs cannot be deleted or updated.

Now the use cases to support are clear:
1)- URL shortening: given a long URL => return a much shorter URL

2)- URL redirecting: given a shorter URL => redirect to the original URL
3)- High availability, scalability and fault tolerance considerations

Next, let's make some simple calculations with the information we get.

Write operation: 10 million URLs are generated per day
Per second: 10 million / 24 /3600 = 115

Read operation: Assume read operation to write operation ratio is 10:1.
Per second: 115 * 10 = 1150

Assume the URL shortener service will run for 100 years. This means we need to support 10 million * 365 * 100 = 465 billion records.

It's important that you walk through the calculation and assumptions with your interviewer so that both of you are on the same page.

Abstract design

Once you've gathered all the requirements and been clear about design goals, it's time to move on to high level design. At first glance, hash table or hash map is perfect for such a system. If you set short URL as key and long URL as value, you can look up long URL by short URL easily and perform the redirect. This means use case "URL redirecting" is supported by the design.

What about "URL shortening" use case? How does that work? To make things easier, let's assume the short URL looks like this https://www.tinyurl.com/$hashValue. We need to find out a hash function *f* that maps a long URL to *$hashValue*.

f(longURL) = $hashValue

The hash function needs to satisfy the following conditions:

• Each *longURL* must be hashed to one and only one *$hashValue*.
• Each *$hashValue* must be mapped back to one and only one *longURL*.

Detailed design

Let's break down the high level design and examine each component in more depth.

Length calculation

Since the domain name https://www.tinyurl.com is fixed, we only need to find the length of *$hashValue*. Considering *$hashValue* can only contain [0-9,a-z,A-Z], there are 10 + 26 + 26 = 62 possible characters. The task of finding the length of $hashValue becomes finding the smallest *n* such that 62^n ≥ 465 billion. When *n = 6, 62^n ≈ 56 billion*. When *n = 7, 62^n = 5.2 trillion*. 5.2 trillion is bigger than 465 billion, so the length of *$hashValue* is 7.

Data model

To begin with, assume everything is stored in a single relational database. The table design is as simple as illustrated in Figure 48. Only 3 columns are needed for the table: *shortURL, longURL* and *id*. *Id* auto-increments by 1 for every new entry.

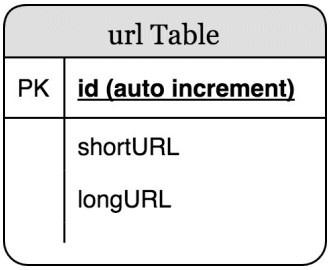

Figure 48

Implement the hash function

Next, we need to implement a hash function that hashes a long URL to a 7-character string. A straightforward solution is reusing well known hash

functions like CRC32, MD5 or SHA-1. The following table compares the hash results of applying these hash functions on URL "https://en.wikipedia.org/wiki/Systems_design".

Hash Function	Hash Value (Hex)
CRC32	5cb54054
MD5	5a62509a84df9ee03fe1230b9df8b84e
SHA-1	0eeae7916c06853901d9ccbefbfcaf4de57ed85b

Table 2

As shown in Table 2, even the shortest hash function, CRC32, is too long for us (more than 7 characters long). If hashing string (*longURL*) to string (*shortURL*) directly is hard, what else could be used as the hash key? As shown in Figure 49, a good alternative is to use the *id* field (integer) to generate *shortURL*. This technique is called base conversion.

id	shortURL	longURL
2009215674938	zn9edcu	https://en.wikipedia.org/wiki/Systems_design

Figure 49

Convert an id to shortURL

Steps to convert *id* field to *shortURL*:

1)- If we map each distinct character possible in *shortURL* to a number, it looks like this:

e.g. 0-0, ..., 9-9, 10-a, 11-b, ..., 35-z, 36-A, ..., 61-Z.

Here is an example of converting *id* to the base 62 representation. 11157_{10} means 11157 with a base of 10.

$11157_{10} = 2 \times 62^2 + 55 \times 62^1 + 59 \times 62^0 = [2,55, 59] = [2, T, X]$

2)- With 2 mapping to 2, 55 mapping to T and 59 mapping to X, we get 2TX as the shortened URL. Therefore, the short URL looks like this:

https://tinyurl.com /2TX

The code snippet below converts an *id* to *shortURL*.

```
// Function to generate a short url from integer id
String idToShortURL(long n) {
   // Map to store 62 possible characters
   char map[] =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
         .toCharArray();

   StringBuilder shorturl = new StringBuilder();

   // Convert given integer id to a base 62 number
   while (n > 0) {
      // use above map to store actual character
      // in short url
      shorturl.append(map[(int) n % 62]);
      n = n / 62;
   }

   // Reverse shortURL to complete base conversion
   return shorturl.reverse().toString();
}
```

Flow for URL shortening

After tackling the detailed components, let's switch angle and design the flow of URL shortening service as a whole.

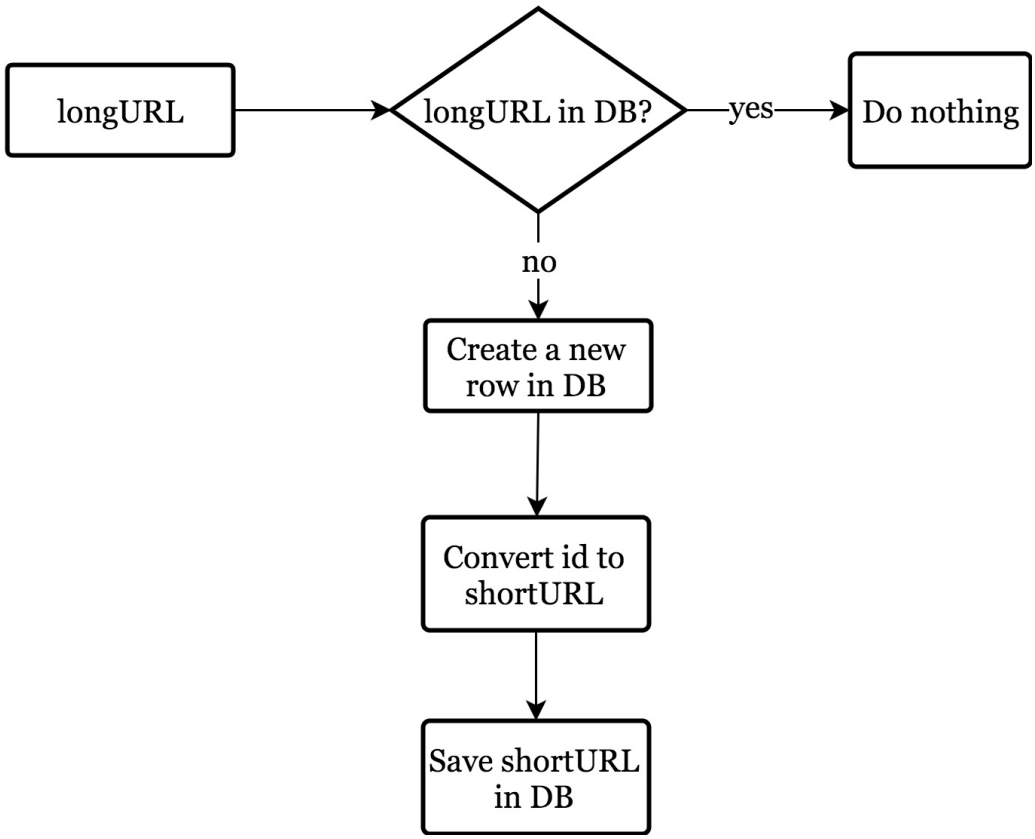

Figure 50

As shown in flowchart Figure 50, if longURL exists in DB, there is no need to perform the URL shortening operation because it's already done in the past.

If longURL doesn't exist in DB, we have to do a few things. Assume "https://en.wikipedia.org/wiki/Systems_design" is the *longURL*.

1)- Create a new row (Figure 51) in the database. *id* and *longURL* fields are filled with corresponding data, but shortURL field is empty.

id	shortURL	longURL
2009215674938		https://en.wikipedia.org/wiki/Systems_design

Figure 51

2)- Convert *id* to *shortURL* using base 62 conversion. In our case, 2009215674938 (id field value) is converted to *zn9edcu* in base-62 representation.

3)- Save *shortURL* in DB as shown in Figure 52.

id	shortURL	longURL
2009215674938	zn9edcu	https://en.wikipedia.org/wiki/Systems_design

Figure 52

Flow for URL redirecting

The flow for URL redirecting can be summarized as the following 3 steps:

1)- Convert *shortURL* back to *id* using base conversion (from 62-base to 10-base). Using the wikipedia link again as an example (Figure 52), *zn9edcu* is converted to *2009215674938*. The code snippet below shows how to convert *shortURL* to *id*.

```
// Function to get integer id back from a shortURL
long shortURLtoID(String shortURL)
{
   long id = 0;

   // A simple base conversion logic
   for (int i = 0; i < shortURL.length(); i++)
   {
      if ('0' <= shortURL.charAt(i) && shortURL.charAt(i) <= '9') {
         id = id * 62 + shortURL.charAt(i) - '0';
      }
      if ('a' <= shortURL.charAt(i) && shortURL.charAt(i) <= 'z') {
         id = id * 62 + shortURL.charAt(i) - 'a' + 10;
      }
      if ('A' <= shortURL.charAt(i) && shortURL.charAt(i) <= 'Z') {
         id = id * 62 + shortURL.charAt(i) - 'A' + 36;
      }
```

```
    }
    return id;
}
```

2)- Locate the database row by *id*. Assume "url_mapping" is the database table name. The query looks like this: *select *from url_mapping where id = 2009215674938*.

3)- Find the *longURL* from the query result and redirect.

Scale your system

When the system gets more traffic, we need to distribute data across multiple servers. One possible high level system looks like this (Figure 53).

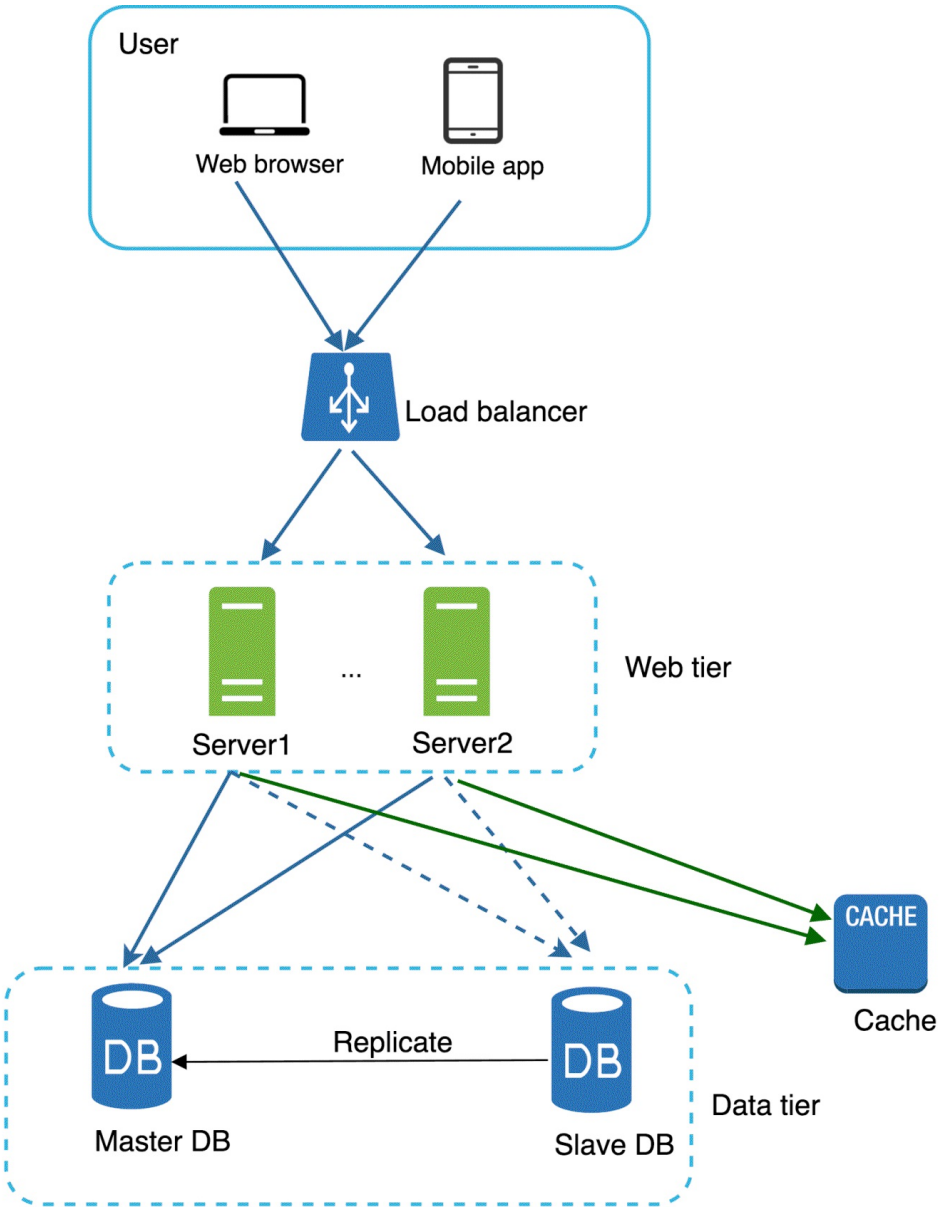

Figure 53

This should look familiar to you as we have talked about similar design intensively in Chapter 1. The cache tier is used to enhance read speed. Popular *<shortURL, longURL>* pairs are cached to reduce database load and at the same time enhance read speed.

Components of the system in Figure 53:

• Clear separation of web tier, data tier and cache tier so they can be scaled independently.
• Load balancer to distribute the load across servers
• Stateless web tier
• Data replication to improve availability and reliability
• Cache tier to improve read performance

The previous chapter covers important techniques to consider while designing distributed key-value store. These techniques are essential and applicable to any distributed system design, thus I list them again as below in Table 3.

Goal/Problems	Technique
Ability to store “big data”	Use consistent hashing to spread load across servers.
High availability reads	Data replication Multi-datacenter replication
Highly available writes	Versioning with vector clocks
Dataset partition	Consistent Hashing
Incremental scalability	Consistent Hashing
Heterogeneity	Consistent Hashing
Tunable consistency	Quorum consensus
Low latency	Set R to small number like 1 in the quorum system
Handling temporary failures	Sloppy quorum and hinted handoff
Handling permanent failures	Merkle tree
Handling data center outage	Cross-datacenter replication

Table 3

AFTERWORD

Congratulations! You are at the end of this interview guide. You are now well equipped with both knowledge and process. Not everyone has the discipline to do what you have done, to learn what you have learned. Take a moment and give yourself a pat on the back. Your hard work will be paid off.

A great resource to further improve your system design ability is http://highscalability.com/. This high scalability blog provides many real-life architectures such as Uber, Instagram, Facebook, etc. I highly recommend you to pay attention to both the shared principles and the technologies that are used. Researching each technology and see what problems it solves is a great way to strengthen your knowledge base and refine the design process.

Landing the dream job is a long journey, requiring lot of time and efforts. Practice makes perfect. Best luck!

Finally, thank you for buying and reading the book. Without readers like you, our work would not exist. Hope you have enjoyed the book!

BIBLIOGRAPHY

[1] A. Ejsmont, Web Scalability for Startup Engineers, McGraw-Hill Education, 2015.

[2] [Online]. Available: http://dev.mysql.com/doc/refman/5.5/en/replication.html.

[3] "wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Multi-master_replication.

[4] mysql, "NDB Cluster Replication: Multi-Master and Circular Replication," [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster-replication-multi-master.html.

[5] A. Homer, J. Sharp, L. Brader, M. Narumoto and T. Swanson, Cloud Design Patterns, Microsoft patterns & practices , 2014.

[6] R. Nishtala, "Facebook, Scaling Memcache at," 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’13).

[7] Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/best-practices-cdn.

[8] "amazon," amazon, [Online]. Available: https://aws.amazon.com/cloudfront/dynamic-content/.

[9] amazon, amazon, [Online]. Available: http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-sticky-

sessions.html.

[10] R. Meshenberg, "netflix," netflix, [Online]. Available: http://techblog.netflix.com/2013/12/active-active-for-multi-regional.html.

[11] M. Tocker, " www.percona.com," [Online]. Available: https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/.

[12] "highscalability.com," [Online]. Available: http://highscalability.com/blog/2010/10/15/troubles-with-sharding-what-can-we-learn-from-the-foursquare.html.

[13] "amazon," [Online]. Available: https://aws.amazon.com/rds/pricing/.

[14] N. Craver. [Online]. Available: http://nickcraver.com/blog/2013/11/22/what-it-takes-to-run-stack-overflow/.

[15] "wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Consistent_hashing.

[16] "highscalability.com," [Online]. Available: http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html.

[17] wikipedia, "wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Consistent_hashing.

[18] D. Karger, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," STOC '97 Proceedings of the twenty-ninth annual ACM symposium on Theory of computing.

[19] W. Vogels, "allthingsdistributed," [Online]. Available: http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html.

[20] "CAP_theorem," [Online]. Available: https://en.wikipedia.org/wiki/CAP_theorem.

[21] G. DeCandia, "Dynamo: Amazon's Highly Available Key-value Store," SOSP'07, 2007.

[22] "cassandra," [Online]. Available: http://cassandra.apache.org/.

[23] "datastax," [Online]. Available: http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/architecture/arcl

[24] I. Gupta, "On Scalable and Efficient Distributed Failure Detectors," Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, 2001.

[25] "amazon.com," Amazon.com Inc., [Online]. Available: http://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/BestPractic

[26] Akamai, [Online]. Available: https://blogs.akamai.com/2015/10/dynamic-page-caching-beyond-static-content.html.

[27] M. Tocker, "www.percona.com," [Online]. Available: https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/.

[28] T. White, "tom-e-white.com," [Online]. Available: http://www.tom-e-white.com/2007/11/consistent-hashing.html.

[29] "imaginea.com," [Online]. Available:

https://blog.imaginea.com/consistent-hashing-in-cassandra/.