SDE I Interview Preparation: The Ultimate Guide

"The smart guide to (de)dupe dirty data" by Suvrat Hiran
https://link.medium.com/3Y3YgFCIE5

System Design Questions

1. Design rideshare system for scooters.

2. Design parking lot

3. Design a collaborative text editor
   Real-time Collaborative Editing with Web Sockets, Node.js & Redis | LakTEK

   Writing: Differential Synchronization

Technical Questions

1. Come up with an algorithm to maximize upload speed of a large file.
2. Implement in memory cache with add,delete and update methods.

OOP

3. Aggregation and Composition design pattern
4. Producer and Consumer design pattern

[How Does the Internet Work?]

DNS(Domain Name System)

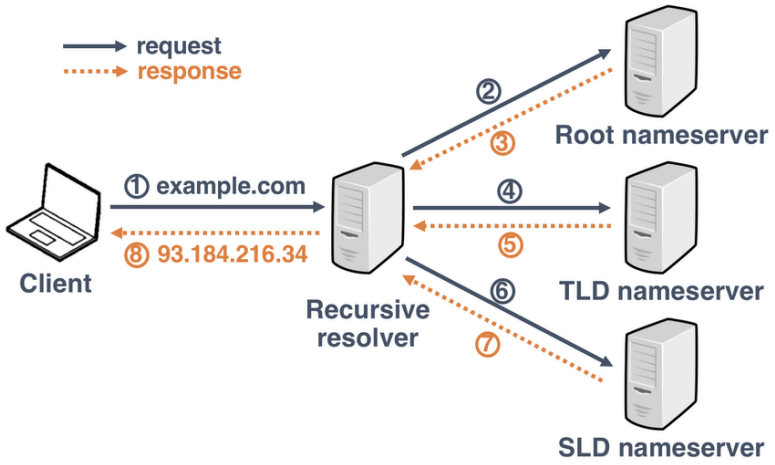
[What is DNS? | How DNS works | Cloudflare]

Components:

- ***DNS Recursive Resolver***: a server designated to receive queries.
- ***Root nameserver(.)***
- ***TLD***(top level domain) ***nameserver(.com, .org)***
- ***Authoritative nameserver(domain names)***

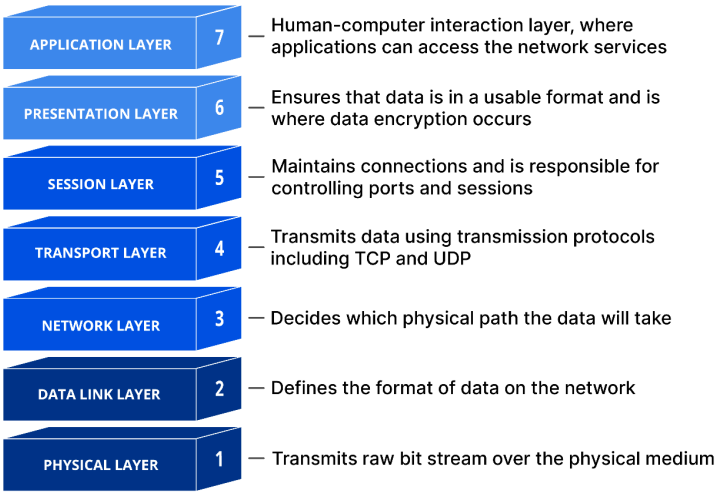
DNS Caching

- DNS data can be cached in a variety of locations, each of which will store DNS records for a set amount of time determined by a time-to-live (TTL).
- ***Browser DNS Caching***: Modern web browsers are designed by default to cache DNS records for a set amount of time. When a request is made for a DNS record, the browser cache is the first location checked for the requested record.
- ***OS level DNS caching:*** second and last local stop before a DNS query leaves your machine.The process inside your operating system that is designed to handle this query is commonly called a “stub resolver” or DNS client. When a stub resolver gets a request from an application, it first checks its own cache to see if it has the record.



OSI Model

[What is the OSI Model? | Cloudflare]



Important layers

1. Physical layer
   deals with time interval voltage and delimiters of the messages

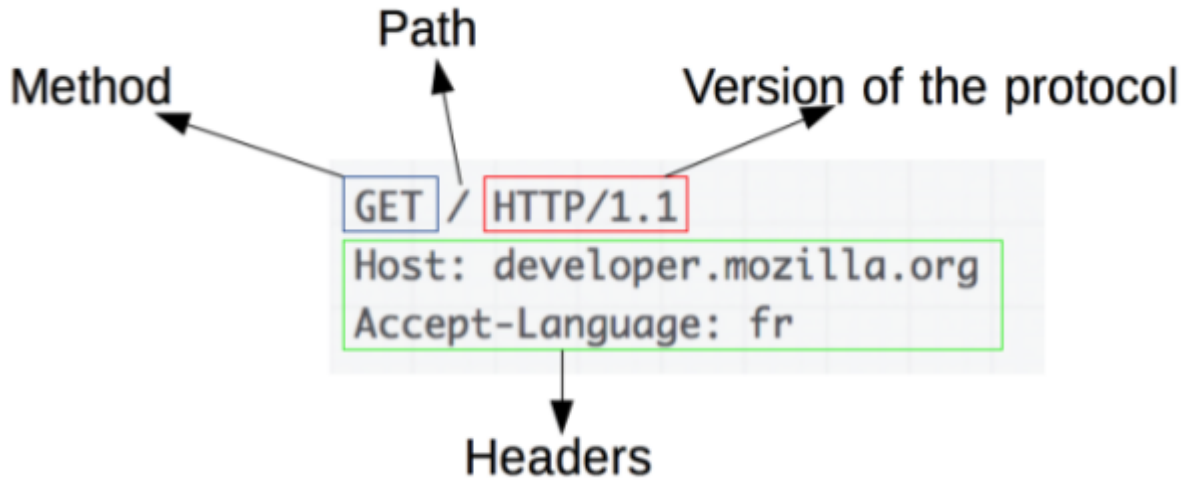
2. Routing layer
   how to route a message from A to B

3. Behavioral layer (Session)
   - frequency and direction of message
   - conversation settings
   - context, i.e. is this message part of the 5mb out of 30 mb file a system was sending

HTTP

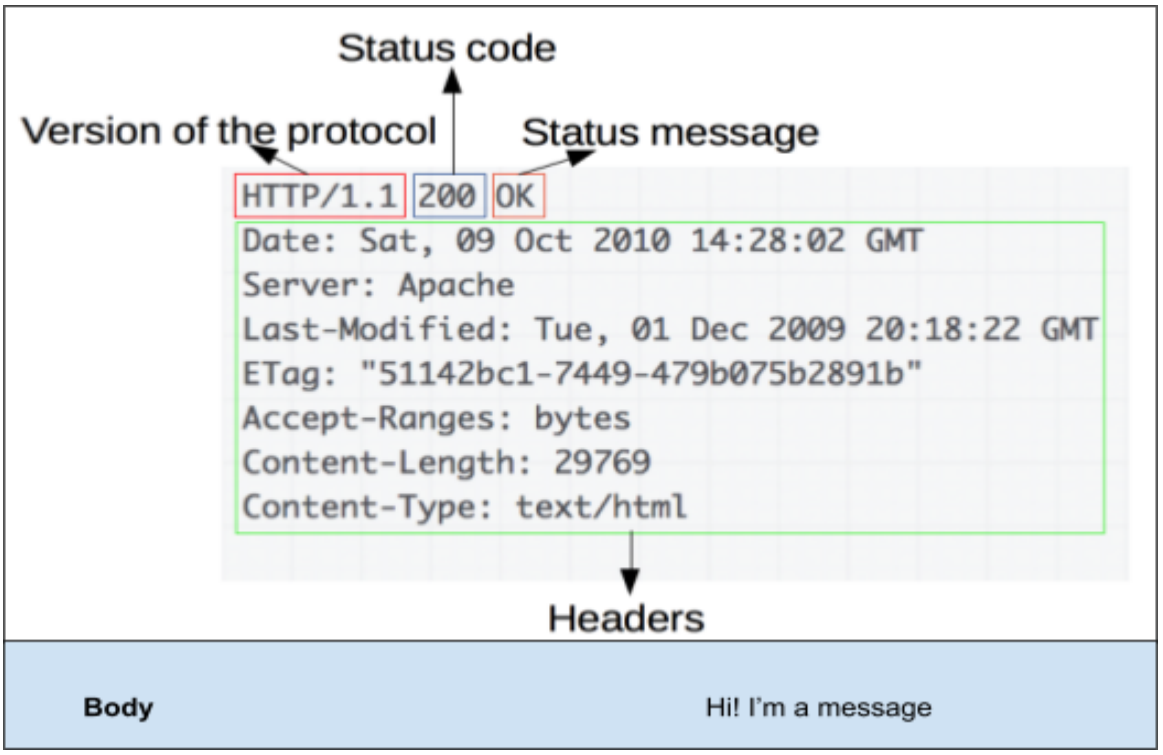
- Stateless

HTTP Request:



1. **Start line:** Method, path and version of protocol
2. **Headers:** For content negotiation(how you want the server to bring the information.)
3. **Body:** (**Not for GET request**) GET requests don't have a request body, so all parameters must appear in the URL or in a header.

HTTP Response:



Analogy:

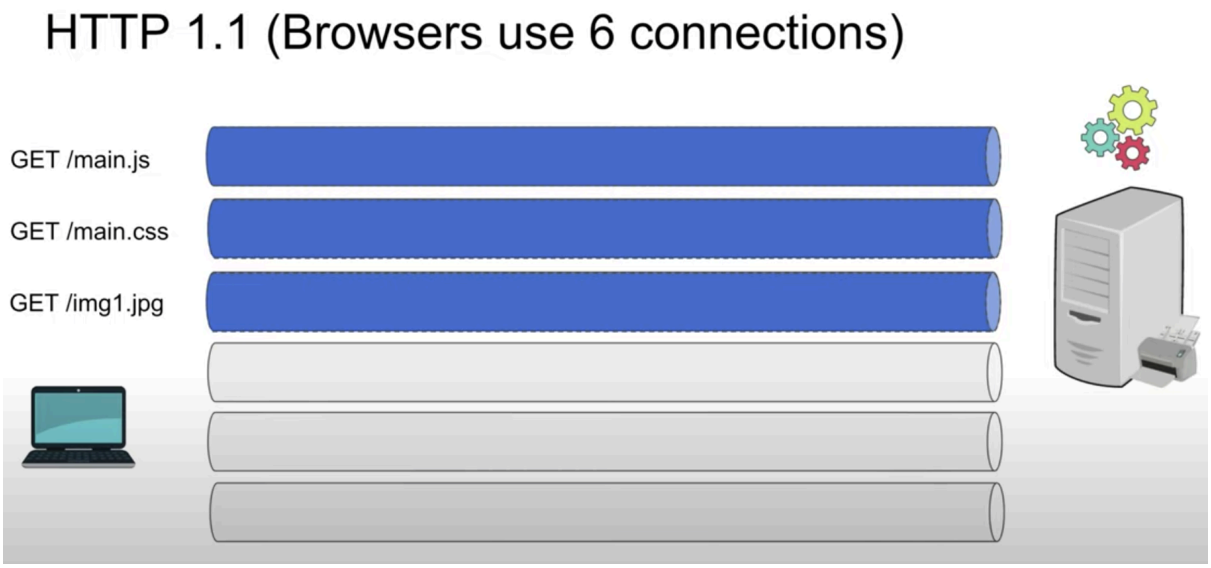
1. **HTTP** is the car.
2. **TCP** acts as a road.
3. **IP** is the coordinate system.
4. **Data link** is the earth.

HTTP Methods:

1. GET
2. POST
3. DELETE
4. PUT: Use PUT APIs primarily to update existing resource. (entire body needs to be sent)
5. PATCH: HTTP PATCH requests are to make partial update on a resource. (only certain fields of the body can be sent)

HTTP 1.1

In HTTP 1.1 as long as the request is processing, the TCP socket is busy. If a client needs multiple files, at most 6 connections are allowed at a time.



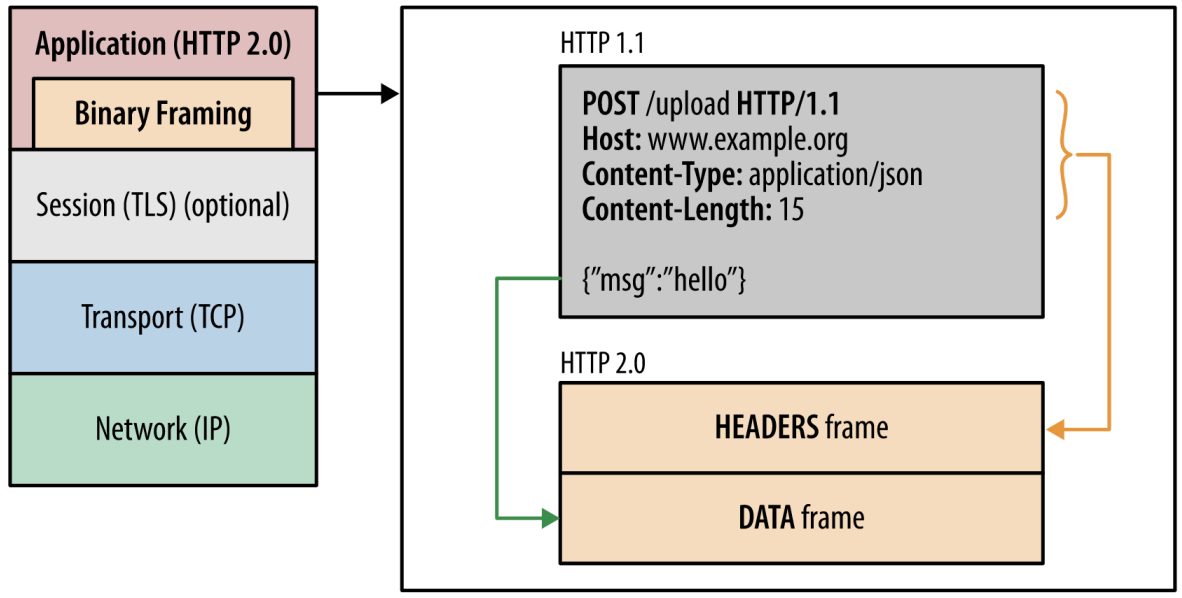
CORS Policy

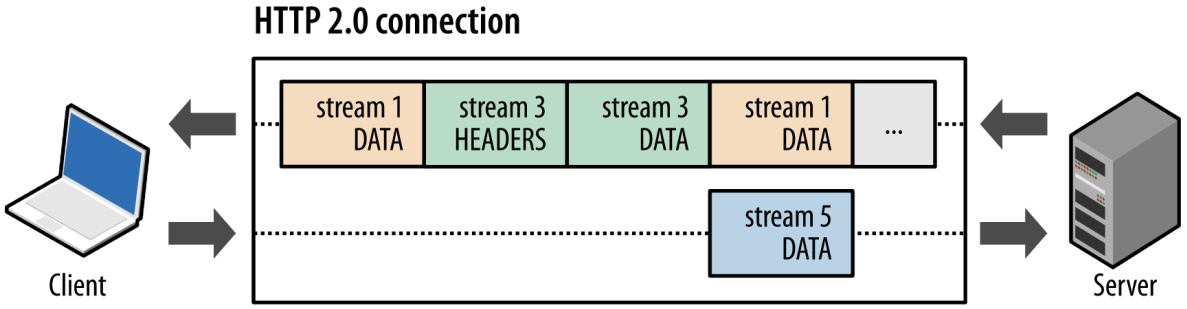
Remember, the same-origin policy tells the browser to block cross-origin requests. When you want to get a public resource from a different origin, the **resource-providing server needs to tell the browser "This origin where the request is coming from can access my resource"**.

1. When the browser is making a cross-origin request, the browser adds an **Origin header** with the current origin (scheme, host, and port).
2. On the server side, when a server sees this header, and wants to allow access, it needs to add an **Access-Control-Allow-Origin header** to the response specifying the requesting origin (or * to allow any origin.)

HTTP 2

HTTP 2 is a binary protocol. Hence we can **multiplex different requests at the same time**. Unlike the newline delimited plaintext HTTP/1.x protocol, all HTTP/2 communication is split into smaller messages and frames, each of which is encoded in binary format. **Stream id is assigned to the frames which helps to distinguish the frames**.





Multiplexing:

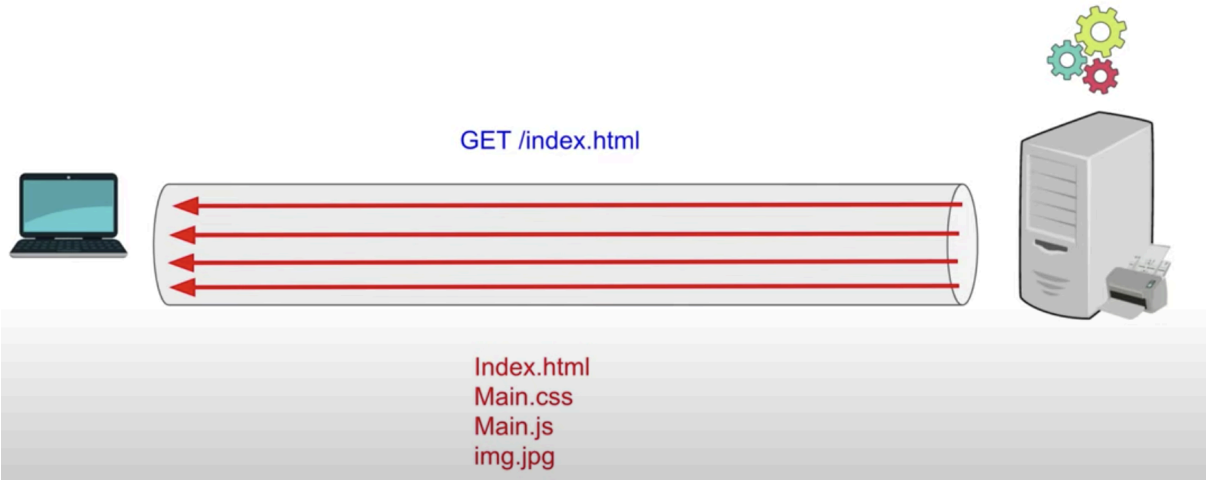
HTTP/2 allows multiple requests to be sent and received simultaneously over a single connection, reducing the number of round trips required to load a webpage.

Binary Protocol:

HTTP/2 uses a binary protocol, which is more efficient than the text-based protocol used in HTTP/1.1. This allows for better compression of headers and data, reducing the size of the data being transmitted over the network.

Server PUSH

Another powerful new feature of HTTP/2 is the ability of the server to send multiple responses for a single client request. That is, in addition to the response to the original request, **the server can push additional resources to the client, without the client having to request each one explicitly**.



Header Compression:

HTTP/2 includes a new header compression algorithm called HPACK, which reduces the size of header data by using dynamic tables to store frequently used headers.

NAT

- Network Address Translation (NAT) is a technique used to map a range of IP addresses in a private network to a range of IP addresses in a public network.
- NAT is used in many home and small office networks, where a **single public IP address is shared among multiple devices**.
- NAT provides security by hiding the private IP addresses of the devices in the network from the internet. It also **conserves IP addresses**, since a single public IP address can be used to support multiple devices.
- However, NAT can cause problems for some applications that require incoming connections from the internet, such as peer-to-peer applications or online gaming. To address these issues, techniques like port forwarding or UPnP (Universal Plug and Play) can be used.

Making API calls in Javascript

XML HttpRequest

- built-in browser object that allows us to make http requests.
- deprecated in ES6, but still widely used.
- works on old as well as new browsers.

AJAX Request(old)
Asynchronously send data to the server without waiting for the response.

An **XMLHttpRequest would need two listeners to be set to handle the success and error cases** and a call to open() and send()

```
// using dummy API from https://jsonplaceholder.typicode.com/
var request= new XMLHttpRequest();
request.open("GET", "https://jsonplaceholder.typicode.com/users");
request.send();
request.onload= function(){
    console.log(JSON.parse(request.response));
}
```

fetch() NEW!

- powerful web api which lets you make asynchronous requests.
- returns a **promise**, it is an object which contains a single value either **response** or an **error**.
- .then() tells the program what to do once the promise is completed.

```
fetch("https://jsonplaceholder.typicode.com/users")
.then(response => {
    return response.json();
}).then(data => {
    console.log(data);
});
```

Axios

- open source library for making http requests.
- works on browser and nodejs

- can be included in html file by using external CDN
- returns promises like fetch API

```
axios.get("https://jsonplaceholder.typicode.com/users")
.then(response =>{
    console.log(response.data);
})
```

jQuery API

- performs asynchronous http request
- can be included in html file by using external CDN
- uses $.ajax() method to make the requests.

```
$(document).ready(function(){
    $.ajax({
        url: "https://jsonplaceholder.typicode.com/users",
        type: "GET",
        success: function(result){
            console.log(result);
        }
    })
})
```

WebSockets

This is the most comprehensive guide to Websockets.

[How JavaScript works: Deep dive into WebSockets and HTTP/2 with SSE + how to pick the right path]

1. **Old structure of web -** pages linked via hyperlinks

2. **Making HTTP "Bidirectional"**: Sneak way to use long-polling to give an impression of bidirectional communication.

3. **Enter websockets** -
   - upgrades an initial HTTP connection to a completely different protocol that has nothing to do with HTTP.
   - designed to work over http ports 80 and 443

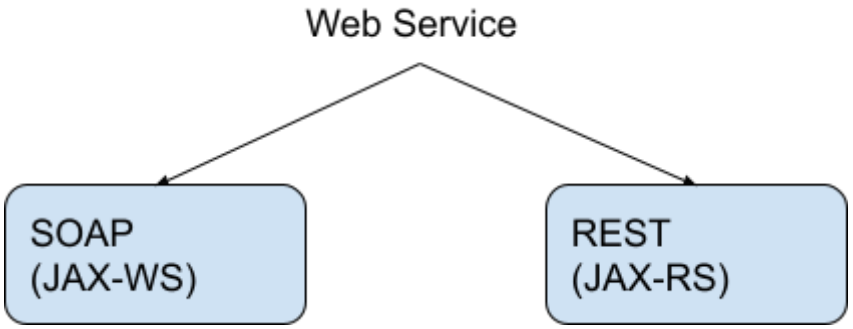
4. **HTTP/2 and Server Push**

5. **Server-Sent Events(SSE)'s EventSource API -** SSE providing the (EventSource)API to enable push up to the applications. This combination of SSE and HTTP/2 makes HTTP purely bidirectional.

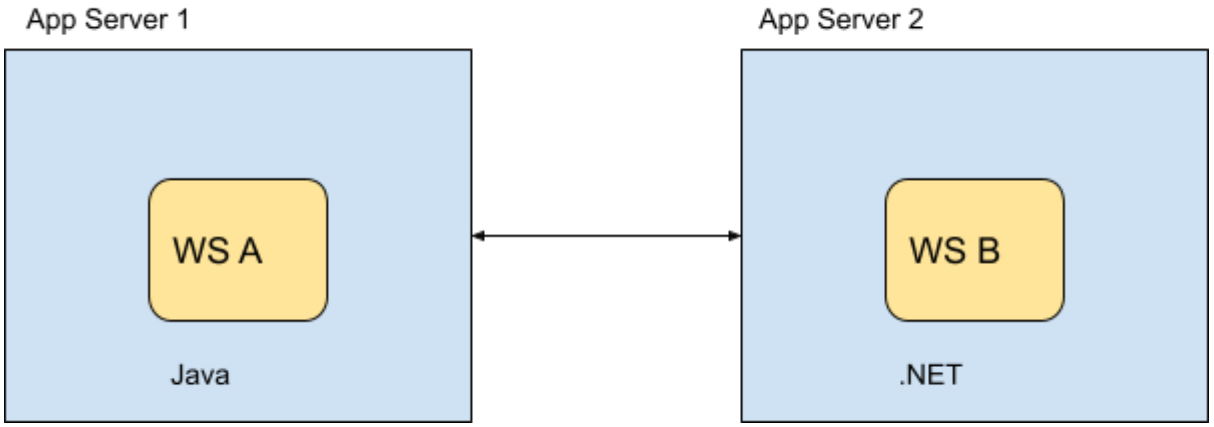
6. **Websocket vs HTTP/2+SSE**

Introduction to Web Services

Web Service is made for code consumption i.e. application level consumption. Unlike a website which is made for human consumption.



Web Service:



Advantage:

1. We can have different applications written in different technologies that communicate with each other.

Let's take a scenario to understand the terminologies associated with web service.
Consider I am writing an **Impl class** and I want to share this class with a Consumer. Best way to do this is by using an Interface which acts like a contract.

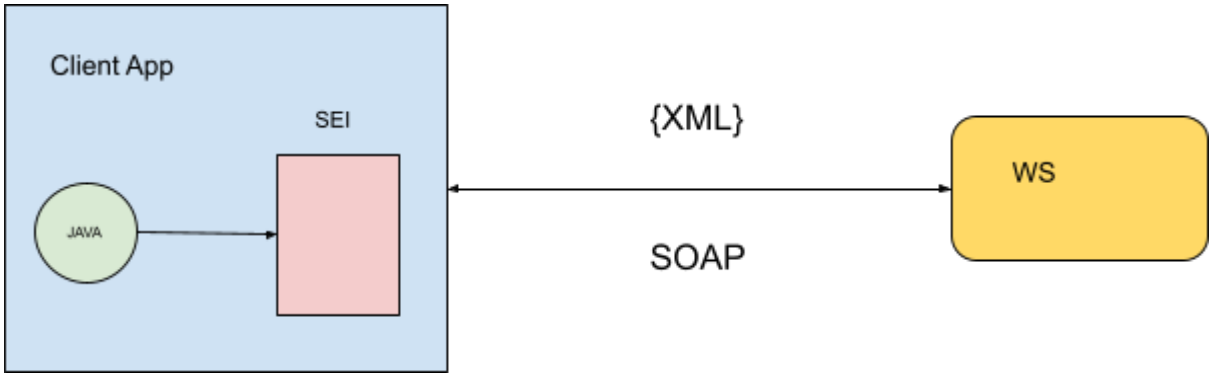
Consumer -> Interface -> Impl class

Now I have to share details of a web service to a consumer. Interface won't work in this scenario as the consumer can be written in a different language. **The contract must be technology independent.**

SOAP

XML was used to share the contract, called as **WSDL**(Web Service Description Language)
UDDI : Yellow pages for web services.(Registry for SOAP based webservices; similar to Eureka/Consul)

Consider you are writing a Client app. and need to exchange data with a web service. How would you send an output list to a web service? Web Service could be in C++. Hence we cannot send a serialized java object. The web service won't understand it. Hence XML(language neutral) is used to exchange information.



SOAP(Simple Object Access Protocol): Way in which different applications can access objects/data.

SEI(Service Endpoint Interface): Converts language specific data to SOAP message. We can have this generated for us.

---

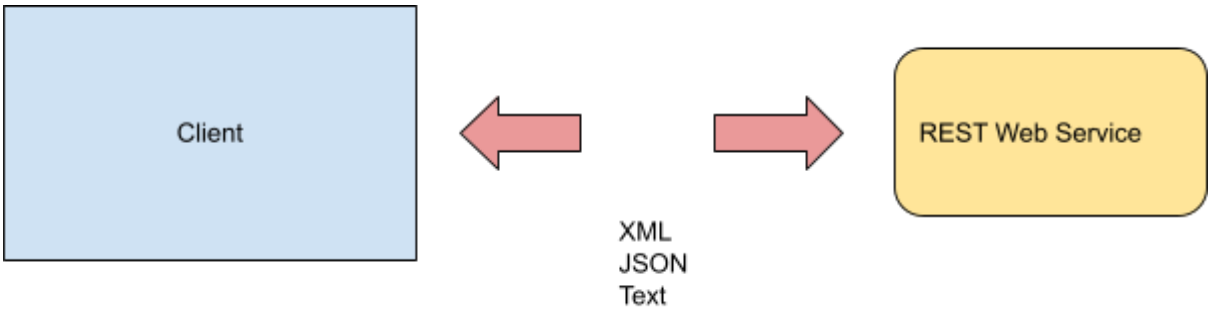
https://stackoverflow.com/questions/2190836/what-is-the-difference-between-http-and-rest

In the SOAP world, WSDL defines the service description, however REST does NOT have any service description as such. SOAP has SOAP Web Service Specification whereas REST does not have any such rules.

REST

REST(Representational State Transfer) is the way HTTP should be used.(it is not a protocol)

- simple and standardized
- scalable and stateless
- performant



Today we only use a tiny bit of the HTTP protocol's methods - namely GET and POST. The REST way to do it is to use all of the protocol's methods. HTTP itself adheres to the REST architectural style, thus enabling you and me to write RESTful applications.

[What is an API?]

- allow applications to talk to each other.

REST API Naming Convention

- A verb in the API is a **design smell**.
  /account/pay ❌
  /account/payment ✅

- A resource can be a singleton or a collection.
  /customers/{customerId}/accounts/{accountId} ✅

Considerations when designing APIs:

1. **Naming**

2. **Return an Object instead of Collection: The example shown below might be an issue**. In the future, if I wanted to add a new variable like "String responseStatus" along with List<Admin>, I can't really do that without making changes in clients. But if I already returned an object, I could add any number of new variables to accommodate in the future.

   Responses that return a List for e.g. List<Admins> should be encapsulated in an Object.

   Consider a function that returns all admins:

   public List<Admins> getAdmins(String groupId){

```
        Errors: groupDoesNotExist
                groupIsDeleted

        Response: List<Admins>
    }

    Return DTOs(Data Transfer Objects)
```

3. **No side effects**: If we are doing multiple things in an API, it is a good idea to split the API.

4. **Avoid large response**: If the response is large. We can use pagination.

HTTP Status Codes

1×× Informational

- **100** Continue
- **101** Switching Protocols
- **102** Processing

2×× Success

- **200** OK
- **201** Created
- **202** Accepted **(Used for Asynchronous APIs i.e. uploading a large file)**
- **203** Non-authoritative Information
- **204** No Content
- **205** Reset Content
- **206** Partial Content
- **207** Multi-Status
- **208** Already Reported
- **226** IM Used

3×× Redirection

- **300** Multiple Choices
- **301** Moved Permanently
- **302** Found
- **303** See Other
- **304** Not Modified
- **305** Use Proxy
- **307** Temporary Redirect
- **308** Permanent Redirect

4×× Client Error

- **400** Bad Request
- **401** Unauthorized: **I don't know who you are. This is an authentication error.**

- `402` Payment Required
- `403` Forbidden: **I know who you are, but you don't have permission to access this resource. This is an authorization error.**
- `404` Not Found
- `405` Method Not Allowed
- `406` Not Acceptable
- `407` Proxy Authentication Required
- `408` Request Timeout
- `409` Conflict
- `410` Gone
- `411` Length Required
- `412` Precondition Failed
- `413` Payload Too Large
- `414` Request-URI Too Long
- `415` Unsupported Media Type
- `416` Requested Range Not Satisfiable
- `417` Expectation Failed
- `418` I'm a teapot
- `421` Misdirected Request
- `422` Unprocessable Entity
- `423` Locked
- `424` Failed Dependency
- `426` Upgrade Required
- `428` Precondition Required
- `429` Too Many Requests
- `431` Request Header Fields Too Large
- `444` Connection Closed Without Response
- `451` Unavailable For Legal Reasons
- `499` Client Closed Request

5×× Server Error

- `500` Internal Server Error
- `501` Not Implemented
- `502` Bad Gateway
- `503` Service Unavailable
- `504` Gateway Timeout
- `505` HTTP Version Not Supported
- `506` Variant Also Negotiates
- `507` Insufficient Storage
- `508` Loop Detected
- `510` Not Extended
- `511` Network Authentication Required
- `599` Network Connect Timeout Error

API Gateway

API Gateway acts as an entry point for the application.

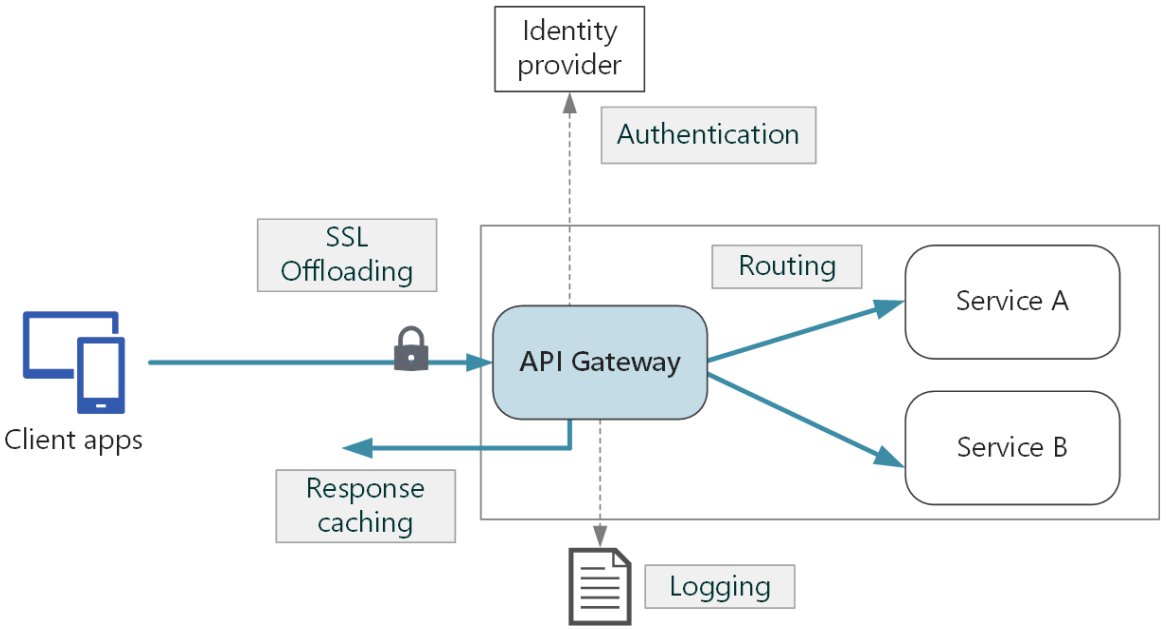
Feature #1
Separate out cross cutting concerns

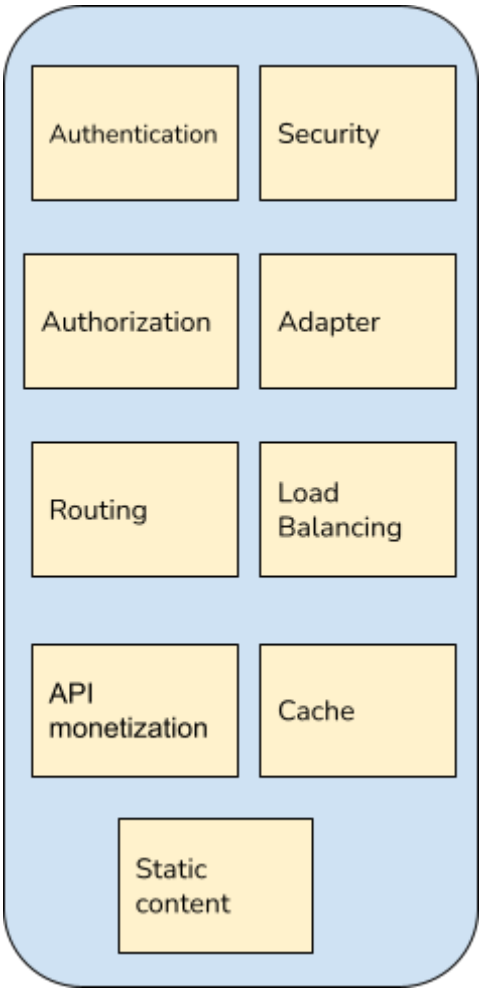
- Authentication
- Authorization
- SSL Termination
- DDOS protection

Feature #2
Routing to a microservice based on path

Feature #3

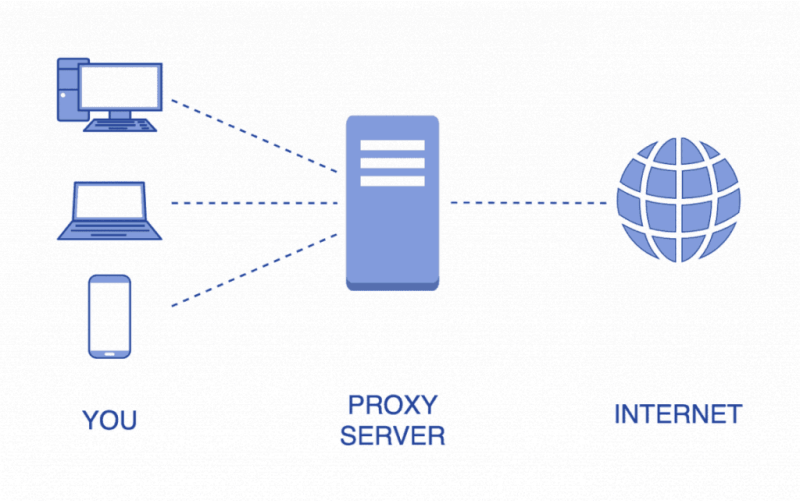
- Instead of clients making multiple calls to fetch trending products, personalized recommendations, API Gateway can handle this with the help of Adapter which can send a consolidated call.
- Static files can be stored on API Gateway.
- Cacheable services can be cached based on timeout on API Gateway.
- Load balancing(in case of multiple copies of a service)
- API monetization





Proxy

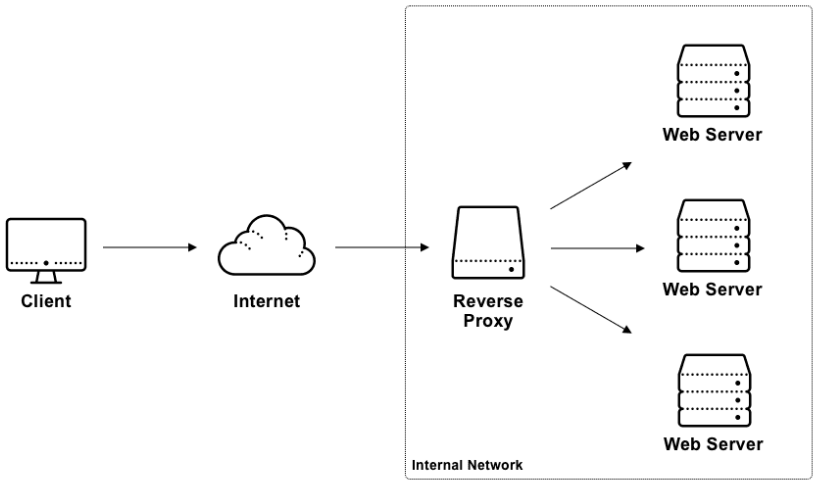
A server that hides the identity of the client. Typically this is ISP server.



Advantages:

1. Anonymity
2. Caching
3. Blocking unwanted sites.
4. Geofencing

Reverse Proxy



Advantages:

1. Load balancing
2. Caching
3. Isolating internal traffic
4. Logging
5. Canary Deployment

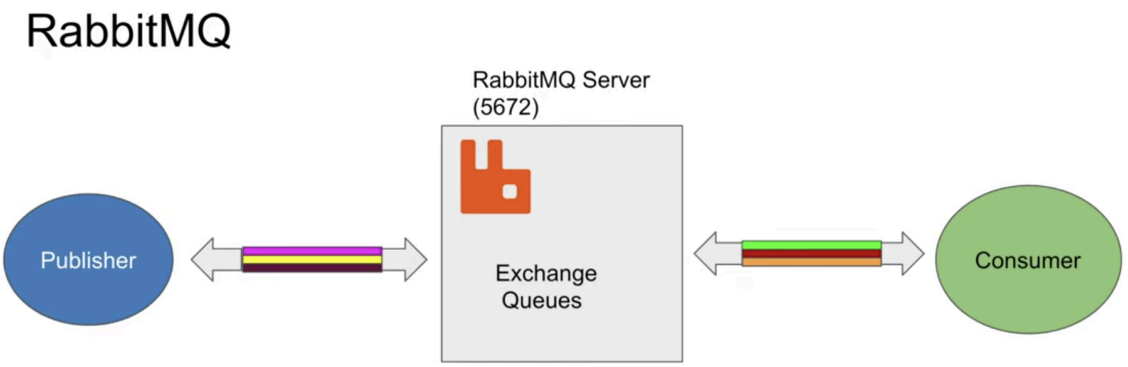
Server side vs Client side Rendering

Server-side rendering

- old way of fetching html pages from server. (think of static html pages linked with anchors or jsps)
- increases load on the server and consumes unnecessary internet bandwidth.

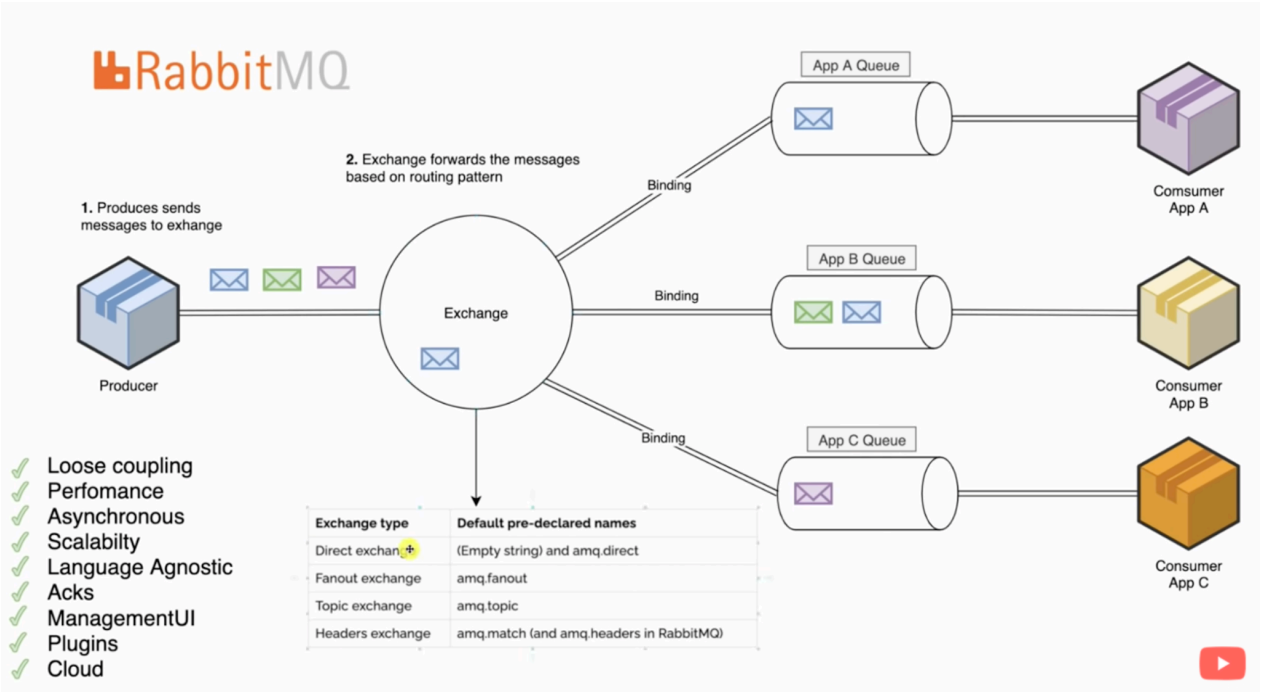
Client-side rendering

- reasonably new approach to rendering websites.
- instead of getting all the content from the HTML document itself, a bare-bones HTML document with a JavaScript file in initial loading itself is received, which renders the rest of the site using the browser.
- initial page load is naturally a bit slow. However, after that, every subsequent page load is very fast.
- possible because of js libraries like Reactjs and Veu.



Producers and Consumers use **AMQP**(Advanced Message Queue Protocol) over a bidirectional connection.
Server **pushes** the messages to the consumer. (different from Kafka)
As RabbitMQ has a **Push Model** there is no guarantee that the consumer will receive that message as the messages grow in number.
There are channels which are nothing but an abstraction in the connection. **Channels**(multi-colored bands) are a way to separate multiple consumers inside that consumer (Multiplexing).

Once the consumer sends **ACK** to the server, the message is dequeued from the exchange queue.



- Each message goes through the exchange.
- There are different types of exchanges: direct, fanout, topic, headers.

- Exchanges behave differently based on config in application.properties

Handling failure in RabbitMQ

In message queueing the **dead letter queue** is a service implementation to store messages that meet one or more of the following failure criteria:

- Message that is sent to a queue that does not exist.
- Queue length limit exceeded.
- Message length limit exceeded.
- Message is rejected by another queue exchange.
- Message reaches a threshold read counter number, because it is not consumed. Sometimes this is called a "back out queue".

RabbitMQ based configuration in application.yml:

```
spring:
  rabbitmq:
    listener:
      simple:
        retry:
          enabled: true
          initial-interval: 3s
          max-attempts: 6
          max-interval: 10s
          multiplier: 2

server:
  port: 8081
```

Consider a scenario where an exception is thrown for a message. The message is sent for 6 times as we have specified max-attempts to 6 and only then our message is sent to the dead letter queue.

Load Balancing

Idea is to balance the load on the servers uniformly.

1. We can use distributed hashing i.e. using a hash table to route a particular request id to a server, the problem with this approach is adding and removing servers. All the requests need to be rehashed. As requests are routed

to a different server the local cache data needs to be rebuilt.

Taking the existence of a good hash function h for granted, we can solve the problem of mapping URLs to caches. Say there are n caches, named $\{0, 1, 2, \ldots, n-1\}$. Then we can just store the Web page with URL x at the cache server named

$$h(x) \bmod n. \tag{1}$$

Note that $h(x)$ is probably something like a 32-bit value, representing an integer that is way way bigger than n — this is the reason we apply the "$\bmod n$" operation to recover the name of one of the caches.

The solution (1) of mapping URLs to caches is an excellent first cut, and it works great in many cases. To motivate why we might need a different solution, suppose the number n of caches is not static, but rather is changing over time. For example, in Akamai's early days, they were focused on adding as many caches as possible all over the Internet, so n was constantly increasing. Web caches can also fail or lose connection to the network, which causes n to decrease. In a peer-to-peer context (see Section 1.5), n corresponds to the number of nodes of the network, which is constantly changing as nodes join and depart.

Suppose we add a new cache and thereby bump up n from 100 to 101. For an object x, it is very unlikely that $h(x) \bmod 100$ and $h(x) \bmod 101$ are the same number, Thus, *changing n forces almost all objects to relocate.* This is a disaster for applications where n is constantly changing.[4]

2. **Consistent Hashing** (What is Consistent Hashing and Where is it used?): Our criticism of simple hashing motivates the goal of consistent hashing. Hashing takes place in a ring. Only few servers are affected because of rehashing.

Data Structure: Balancing BSTs like Red-Black trees can be used for this operation:

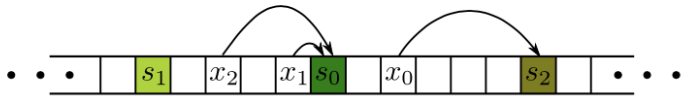


Figure 2: Each element of the array above is a bucket of the hash table. Each object x is assigned to the first cache server s on its right.

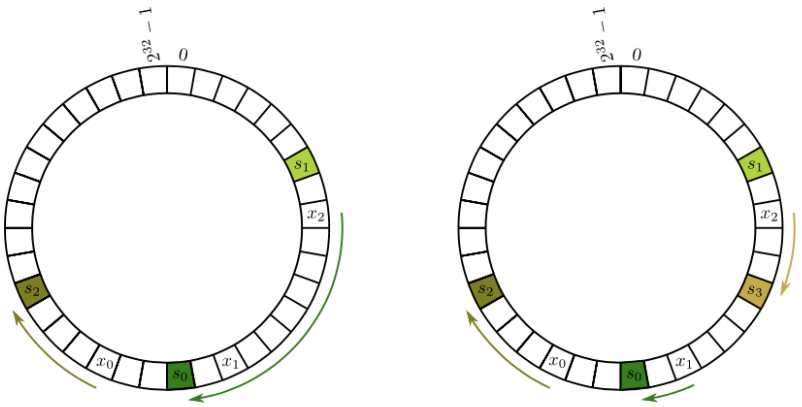


Figure 3: (Left) We glue 0 and $2^{32}-1$ together, so that objects are instead assigned to the cache server that is closest in the clockwise direction. This solves the problem of the last object being to the right of the last cache. (Right) Adding a new cache server s_3. Object x_2 moves from s_0 to s_3.

The key idea is: in addition to hashing the names of all objects (URLs) x, like before, *we also hash the names of all the cache servers* s. The object and cache names need to be hashed to the same range, such as 32-bit values.

To understand which objects are assigned to which caches, consider the array shown in Figure 2, indexed by the possible hash values. (This array might be very big and it exists only in our minds; we'll discuss the actual implementation shortly.) Imagine that we've already hashed all the cache server names and made a note of them in the corresponding buckets. Given an object x that hashes to the bucket $h(x)$, we scan buckets to the right of $h(x)$ until we find a bucket $h(s)$ to which the name of some cache s hashes. (We wrap around the array, if need be.) We then designate s as the cache responsible for the object x.

This approach to consistent hashing can also be visualized on a circle, with points on the circle corresponding to the possible hash values (Figure 3(left)). Caches and objects both hash to points on this circle; an object is stored on the cache server that is closest in the clockwise direction. Thus n caches partition the circle into n segments, with each cache responsible for all objects in one of these segments.

This simple idea leads to some nice properties. First, assuming reasonable hash functions,

by symmetry, the expected load on each of the n cache servers is exactly a $\frac{1}{n}$ fraction of the objects. (There is non-trivial variance; below we explain how to reduce it via replication.) Second, and more importantly, suppose we add a new cache server s — which objects have to move? *Only the objects stored at s.* See Figure 3(right). Combined, these two observations imply that, in expectation, adding the nth cache causes only a $\frac{1}{n}$ fraction of the objects to relocate. This is the best-case scenario if we want the load to be distributed evenly — clearly the objects on the new cache have to move from where they were before. By contrast, with the naive solution (1), on average only a $\frac{1}{n}$ fraction of the objects *don't* move when the nth cache is added![6]

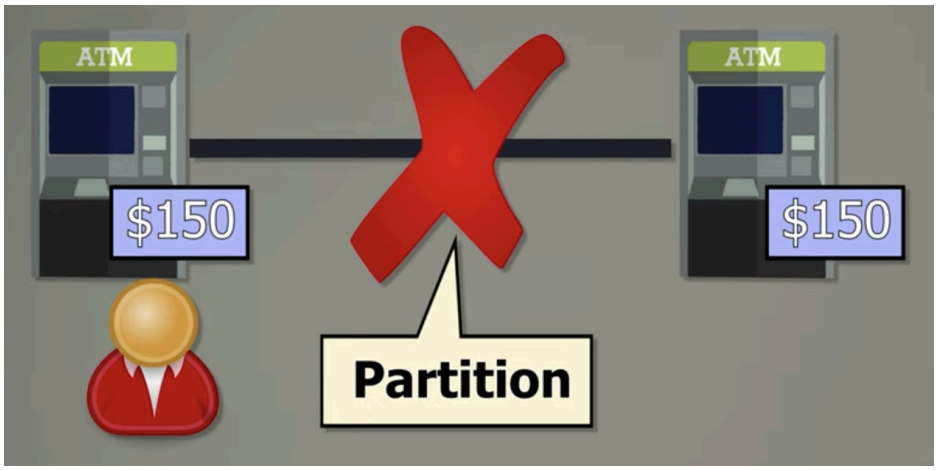
So how do we actually implement the standard hash table operations Lookup and Insert? Given an object x, both operations boil down to the problem of efficiently implementing the rightward/clockwise scan for the cache server s that minimizes $h(s)$ subject to $h(s) \geq h(x)$.[7] Thus, we want a data structure for storing the cache names, with the corresponding hash values as keys, that supports a fast *Successor* operation. A hash table isn't good enough (it doesn't maintain any order information at all); a heap isn't good enough (it only maintains a partial order so that identifying the minimum is fast); but recall that *binary search trees*, which maintain a total ordering of the stored elements, do export a Successor function.[8] Since the running time of this operation is linear in the depth of the tree, it's a good idea to use a balanced binary search tree, such as a Red-Black tree. Finding the cache responsible for storing a given object x then takes $O(\log n)$ time, where n is the number of caches.[9]

CAP Theorem

It states that in the presence of partitions (i.e. network failures), a system cannot be both consistent and available, and must choose one of the two.

CAP stands for

- **Consistency** - data is consistent across all machines i.e. no discrepancies.
- **Availability** - you allow updates to the data, hence two machines can have different views of the data.
- **Partition Tolerance**

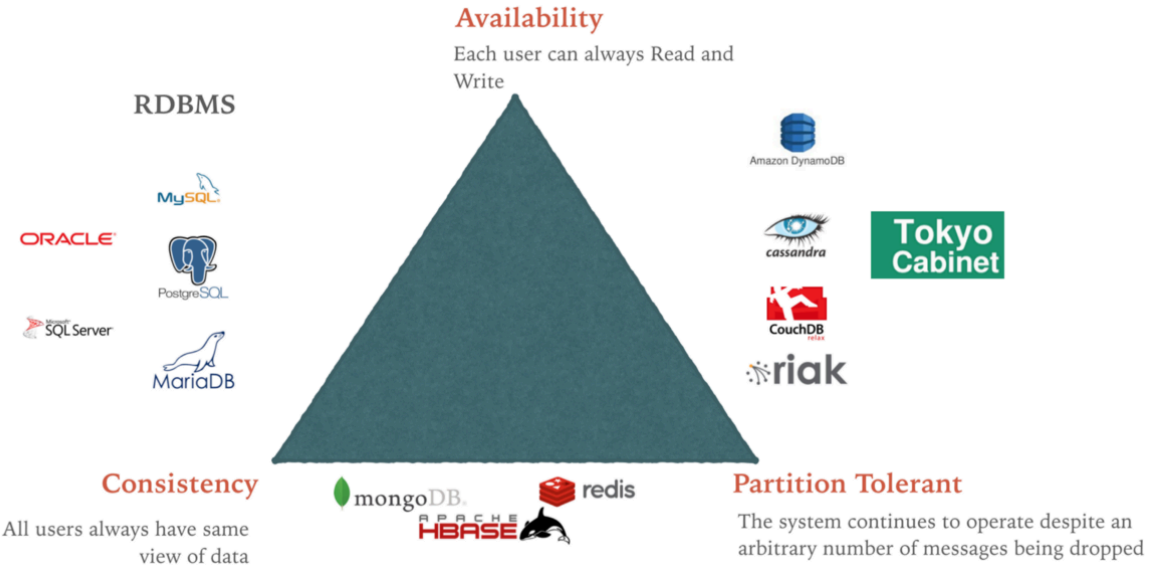


In the above scenario what do the two machines do next? This is the design decision that CAP theorem talks about: **It can either be consistent or available.**

The Limits of the CAP Theorem -> Explained by CockroachDB

How cockroach db serve read and write?
CockroachDB is a CAP-Consistent (CP) system: each piece of data lives on at least three replicas, and **writes** require that a majority of those replicas are able to communicate with each other. **For reads**, one of those replicas is granted a lease, or temporary ownership of a range of data, that allows it to serve reads without communicating with the others for a few seconds. In the event that the leaseholder is partitioned away from the other replicas, it will be allowed to continue to serve reads (but not writes) until its lease expires (leases currently last 9 seconds by default), and then one of the other two replicas will get a new lease (after waiting for the first replica's lease to expire). This ensures that the system recovers quickly from outages, maximizing availability even though it does not satisfy the CAP theorem's all-or-nothing definition of availability.

How does it compare to a traditional RDBMS and also to other distributed (NoSQL databases)?
In order to deal with the sheer volume of data today, businesses have been moving to distributed architectures in recent years. Adapting to this new approach is a major challenge companies currently face. **The most widely used relational databases are difficult to scale out and are unable to handle the dynamic nature of cloud environments**. NoSQL databases, on the other hand, have come of age alongside the cloud. While these NoSQL solutions can take advantage of the elastic scale cloud deployments offer, they do so at the cost of basic features such as ACID transactions and the consistency and correctness of SQL. CockroachDB is a cloud native database that is built specifically to work in the cloud, scales well, and does so without sacrificing the inherent advantages of a SQL database. It is the best of both worlds.


Availability
Each user can always Read and Write
RDBMS
MySQL
ORACLE
PostgreSQL
SQL Server
MariaDB
Amazon DynamoDB
cassandra
Tokyo Cabinet
CouchDB
riak
Consistency
All users always have same view of data
mongoDB.
redis
APACHE HBASE
Partition Tolerant
The system continues to operate despite an arbitrary number of messages being dropped

Data Replication Strategies

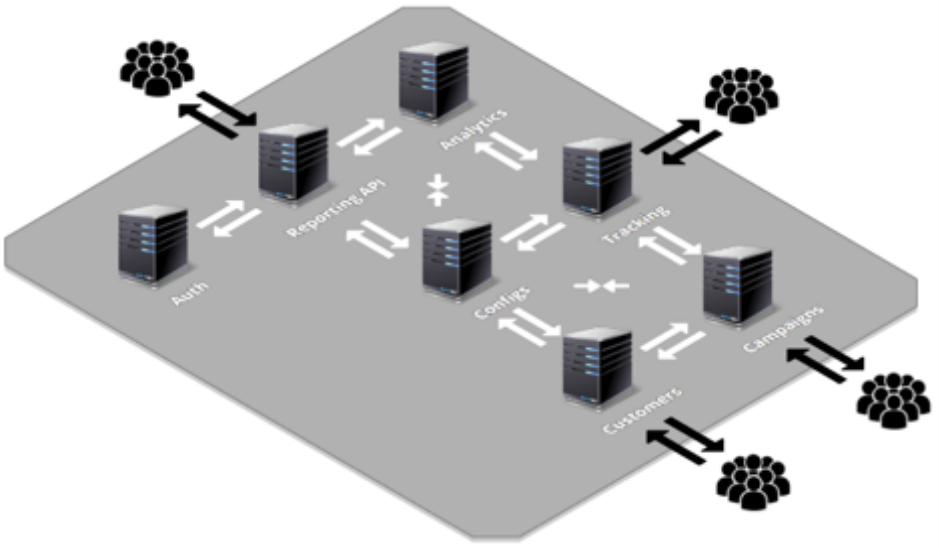
Distributed Consensus and Data Replication strategies on the server

- Master slave architecture (async and sync replication): Here master becomes single point of failure.
- Master master architecture (leads to split brain problem)
- Distributed consensus (Algorithms: 2Phase Commit, MVCC)

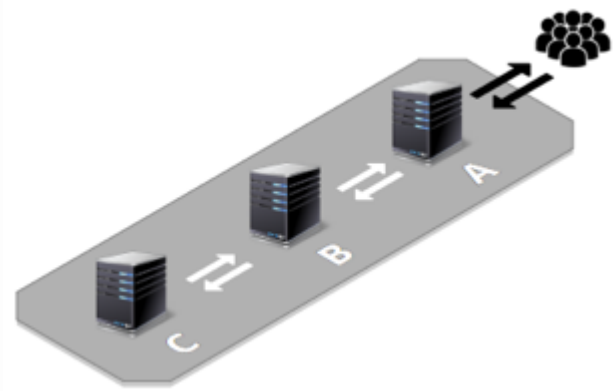
Microservices

Are Your Microservices Overly Chatty? – The New Stack

In practice, using **synchronous communications, like HTTP(REST)**, across the entire system makes it behave like a monolith or more precisely a **distributed monolith** that prevents the full benefits of microservices from being enjoyed.



An illustration of overly chatty microservices (AKA spaghetti)

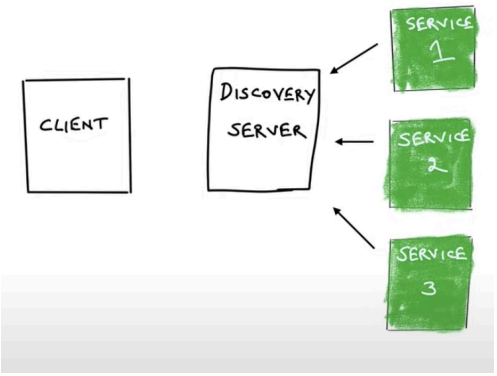


In the above picture, the services are **tightly coupled**. The main risk to pay attention to when using this pattern is that each “core” service (e.g. service C in the above illustration) becomes a **single point of failure**. Meaning, it can potentially create a performance bottleneck – or worse, a downtime of the dependent services.

In order to untangle that mess, we’re moving many core services to communicate using an **asynchronous event-driven architecture**.

Service Registry and Discovery

(*Spring Cloud’s Netflix Eureka, Hashicorp Consul, Apache Zookeeper*)

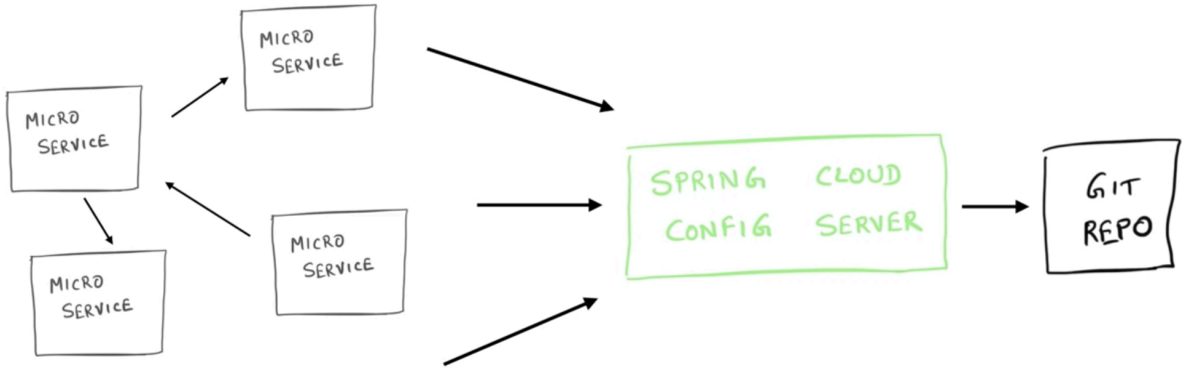


In microservices,
- hardcoded urls are bad.
- dynamic urls are generated in the cloud.
- load balancing of these microservices
- multiple environments.

Hence we need some sort of discovery to register the services.

Config as a service

1. Apache Zookeeper
2. Hashicorp Consul
3. Spring Cloud Configuration Server



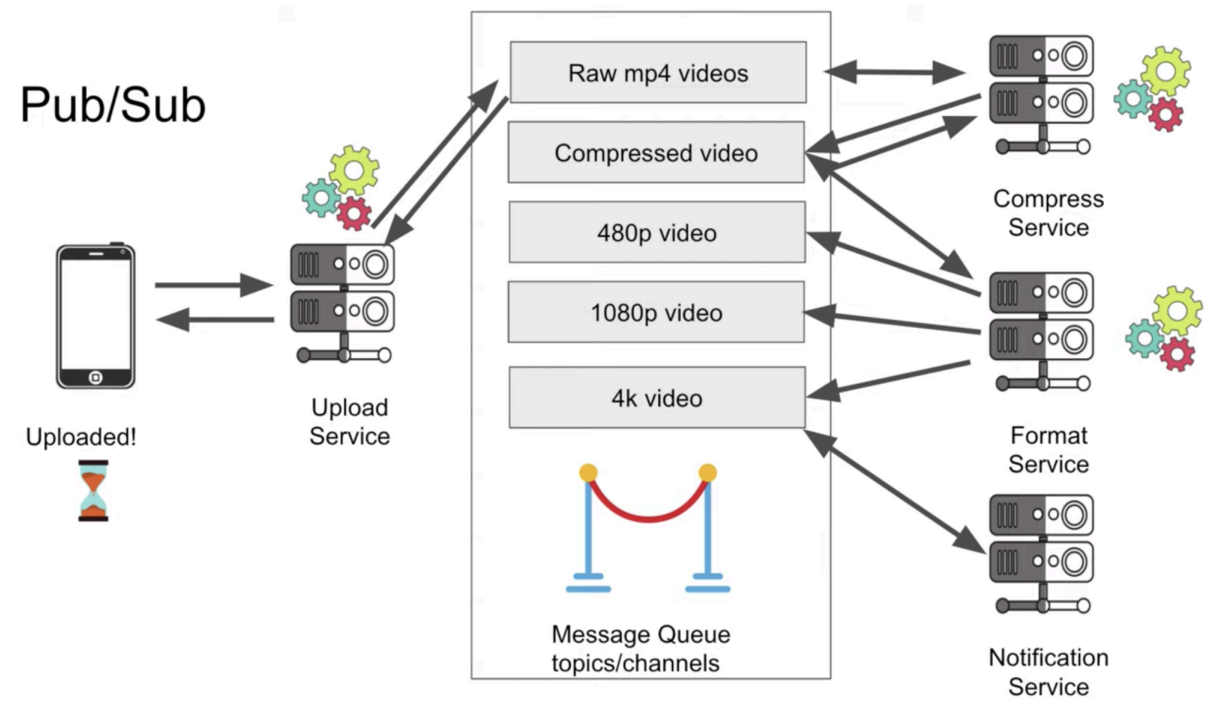
Kafka

Distributed stream processing software.

General use-case: If the cost of calculating the response is high OR if the same response is required by multiple services having a pub/sub model helps.

Idea: In asynchronous communications, a service may still rely on another service, meaning the API and dependency between the parties still exists. But if one service fails or is overloaded and slow to respond, it will not affect the other services since **they're now loosely coupled from each other** and contain everything they need to respond.

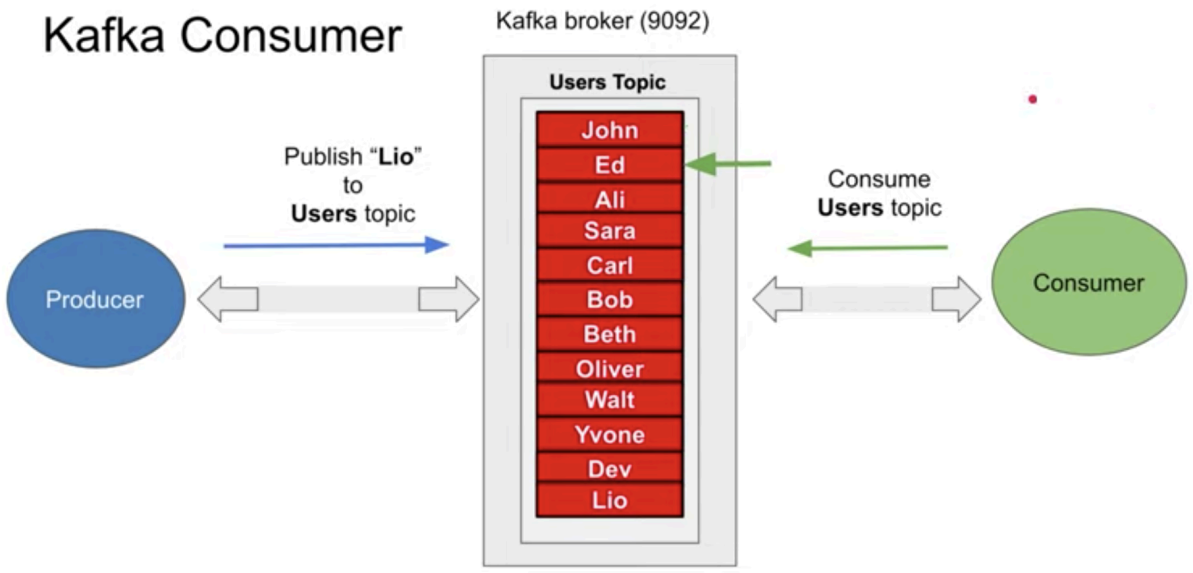
So the benefit of a common event bus is that **it eliminates the single point of failure and performance bottlenecks** we had when our core services communicated synchronously – e.g. the queue can still keep the messages sent to Service B until it's back up and able to consume them.



Producer:
Publishes messages to the topic. A message is referred to by the topic and the position.

Consumer:
Consumes messages from the topic. Consumers use **long polling** to get the message.

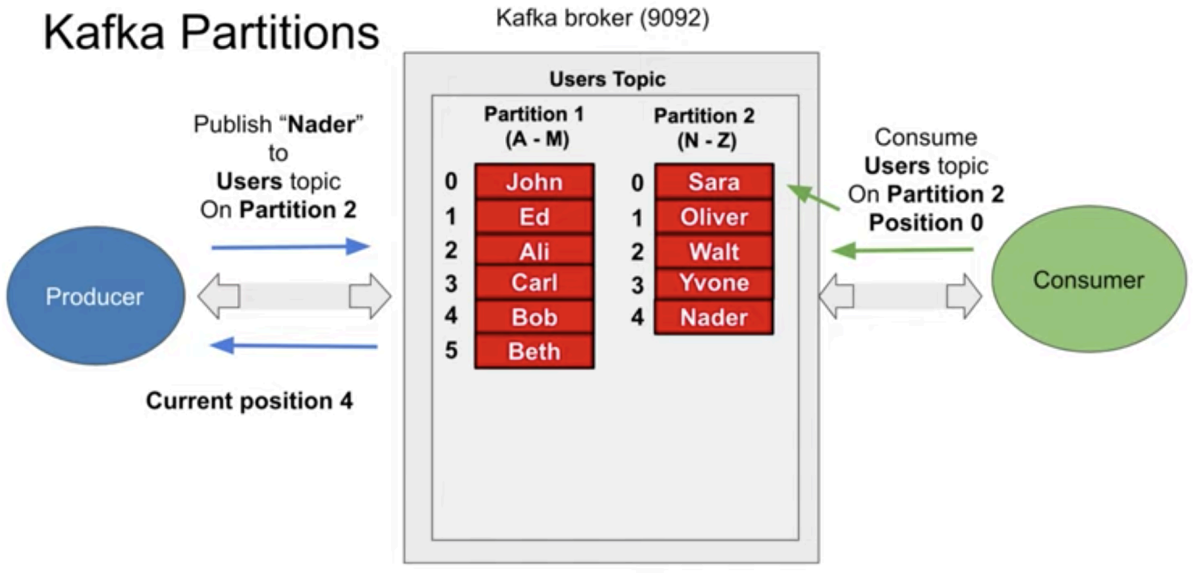
Cluster of brokers:
Responsible for managing and storing topics(messages).



As topics grow large, we shard them.

Partitions:
A topic is distributed across a number of brokers so that each broker manages subsets of messages for each topic - these subsets are called partitions.

To get all messages in a topic:
To make sure an application gets all the messages in a topic, ensure the application has **its own consumer group.**



Queue vs Pub-Sub

Queue: Message published once, consumed once.

Pub-Sub: Message published once, consumed many times.

Kafka tries to achieve both.
To act like a **queue**, put all consumers(=partitions) in one group. (because the partition is only responsible for that particular consumer.)

Kafka's unique characteristic is that it does not track acknowledgments from consumers the way many JMS queues do. Instead, it **allows consumers to use Kafka to track their position (offset)** in each partition.

The action of updating the current position in the partition a **commit**.

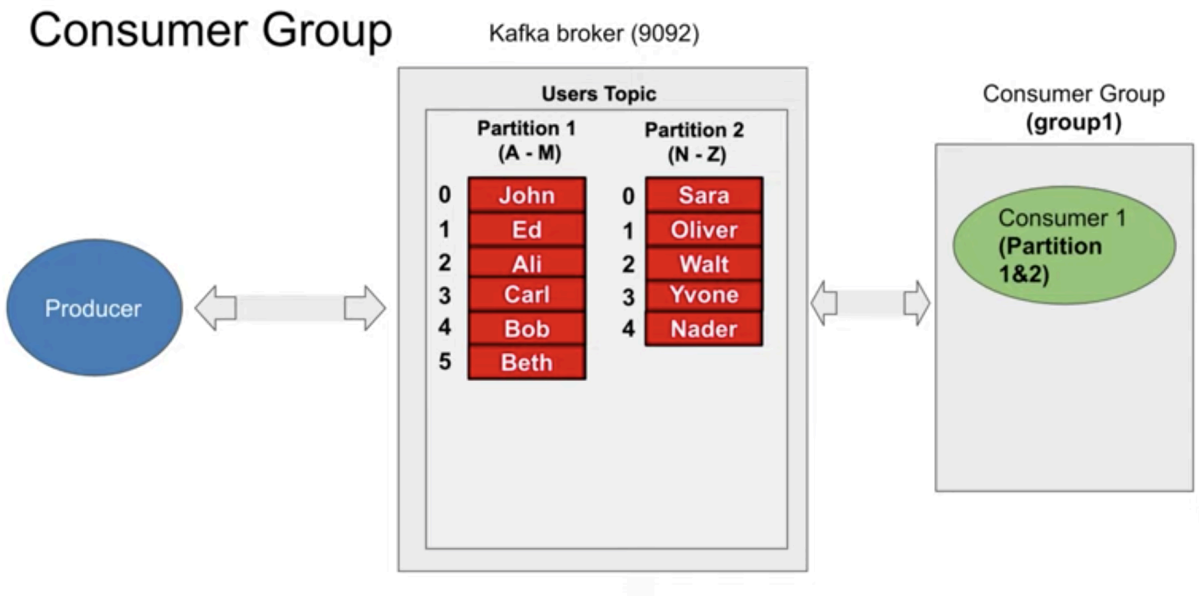
How to exit long polling?
When you decide to exit the poll loop, you will need another thread to call **consumer.wakeup()**.

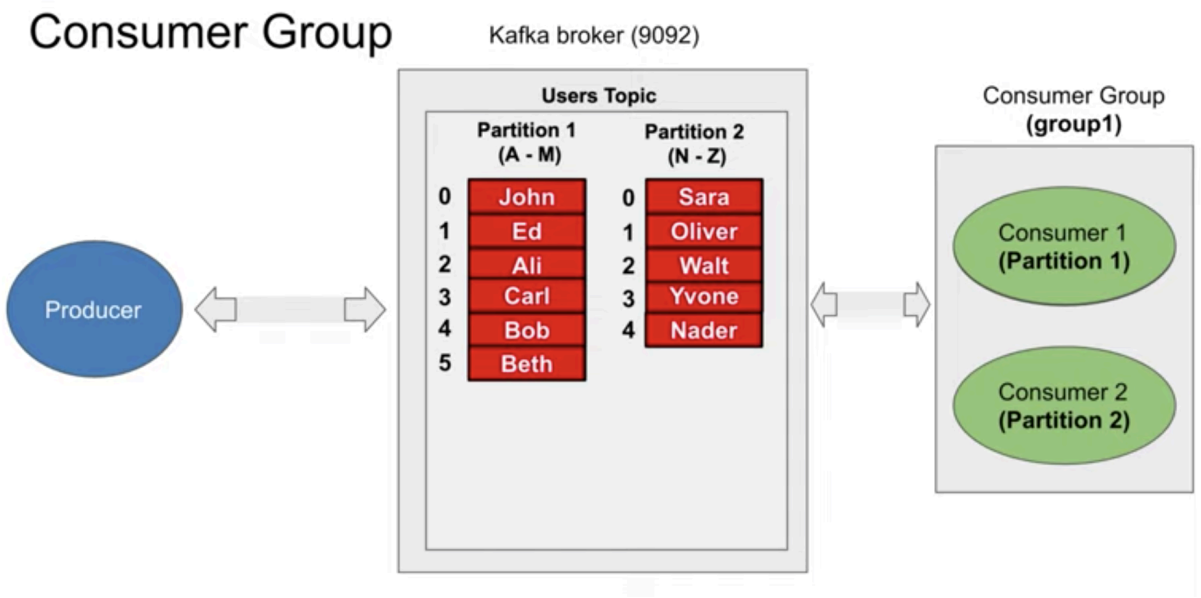
```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup();
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

...

try {
    // looping until ctrl-c, the shutdown hook will cleanup on
exit
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(1000);
        System.out.println(System.currentTimeMillis() +
            "--  waiting for data...");
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("offset = %d, key = %s, value =
%s\n",
                record.offset(), record.key(), record.value());
```

```
        }
        for (TopicPartition tp: consumer.assignment())
            System.out.println("Committing offset at position:" +
                consumer.position(tp));
            movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) {
    // ignore for shutdown
} finally {
    consumer.close();
    System.out.println("Closed consumer and we are done");
}
```

Kafka rebalances the partitions between the group members using **Rebalance protocol.**

Maintaining membership:
The way consumers maintain membership in a consumer group and ownership of the partitions assigned to them is by **sending heartbeats** to a Kafka broker designated as the group coordinator.
Heartbeats are sent when the consumer polls (i.e., retrieves records) and when it commits records it has consumed.





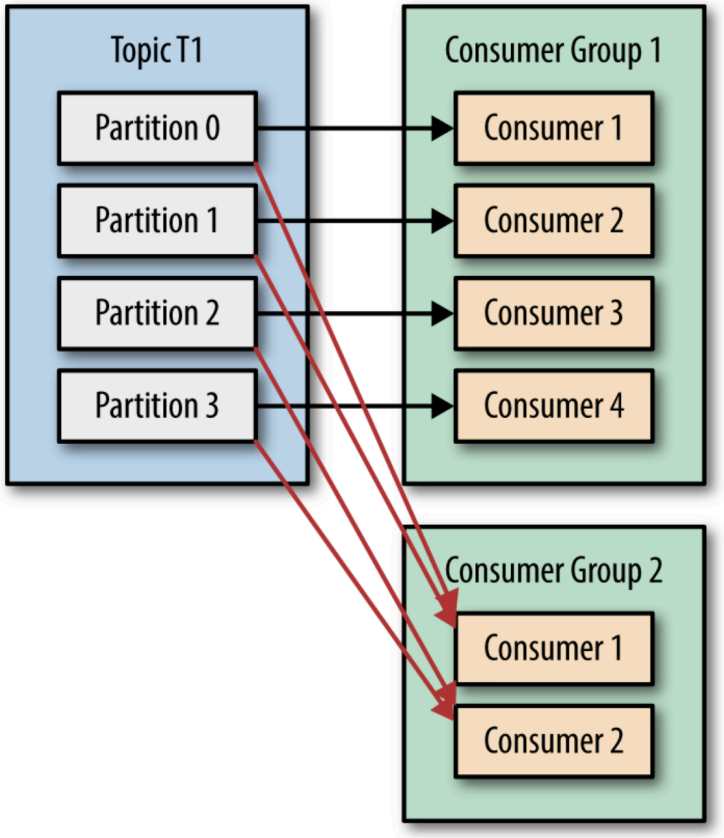
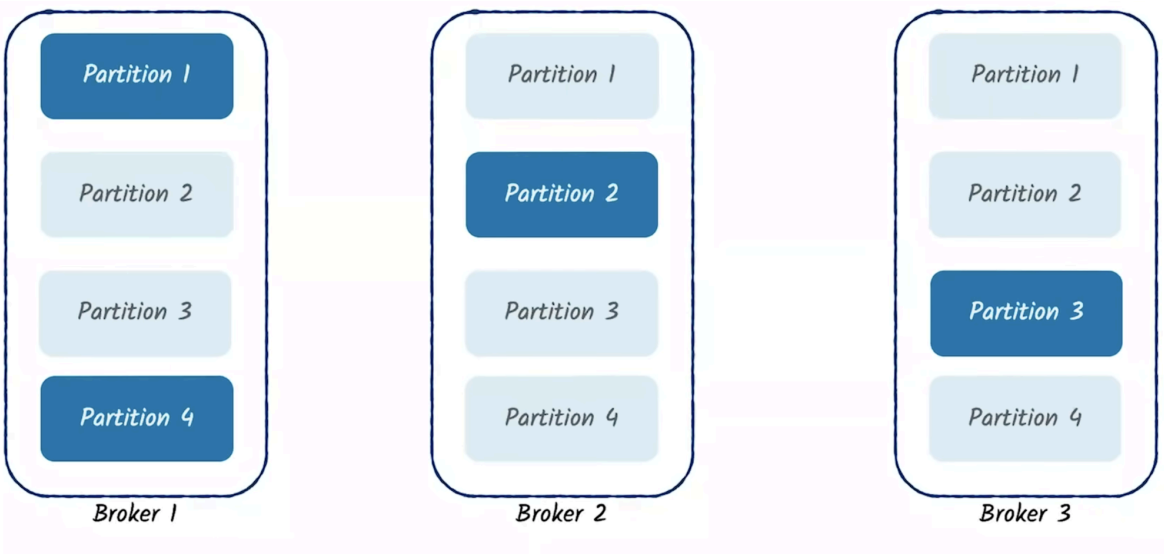


Figure 4-5. Adding a new consumer group, both groups receive all messages

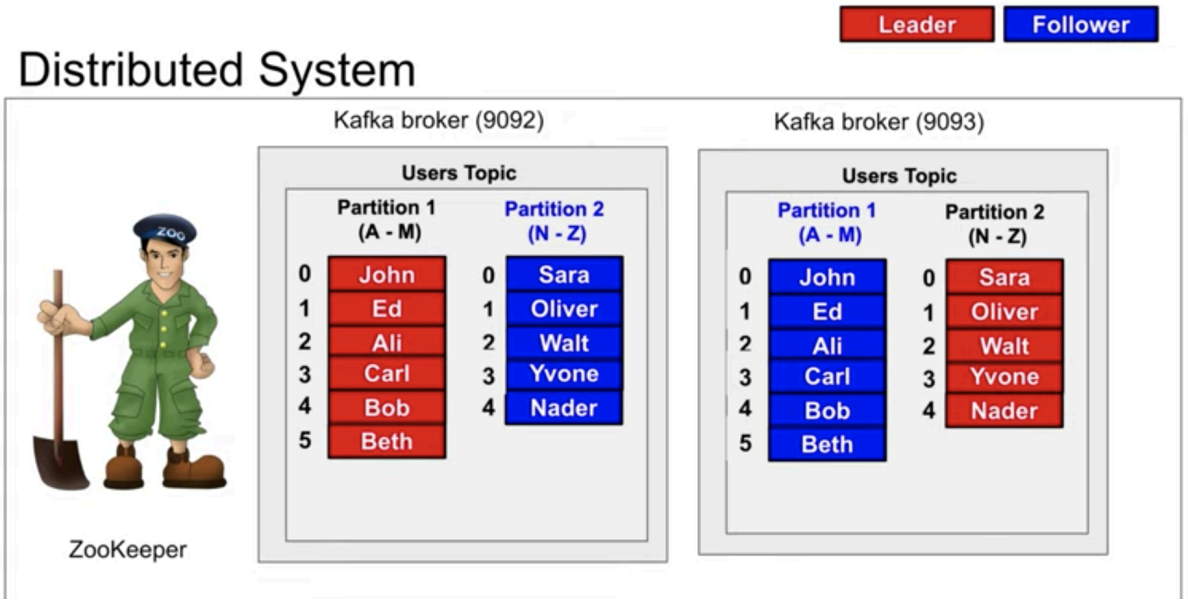
To make the system more fault tolerant, instead of having a leader broker and a follower broker. **Kafka maintains a leader and follower for partitions.**

Partition leadership and replication



Dark blue partitions are the leaders and the others are replicas which can take over in case of failures of a broker.

How does a consumer commit an offset?
It produces a message to Kafka, to a *special* ***__consumer_offsets*** **topic**, with the committed offset for each partition. As long as all your consumers are up, running, and churning away, this will have no impact. However, if a consumer crashes or a new consumer joins the consumer group, this will trigger a **rebalance**. After a rebalance, each consumer may be assigned a new set of partitions than the one it processed before. In order to know where to pick up the work, the consumer will read the latest committed offset(from **commit log**) of each partition and continue from there.



(*hypothesis*)When a producer writes a message to the broker, the cluster of brokers gossip between themselves to find the leader of the partition.

This information is stored by **Zookeeper**.

Zookeeper
keeps track of:

1. Which node is the master?
2. What tasks are assigned to the workers?
3. Which workers are currently available?

Zookeeper API is like a file system. Each element is called a znode.

Databases

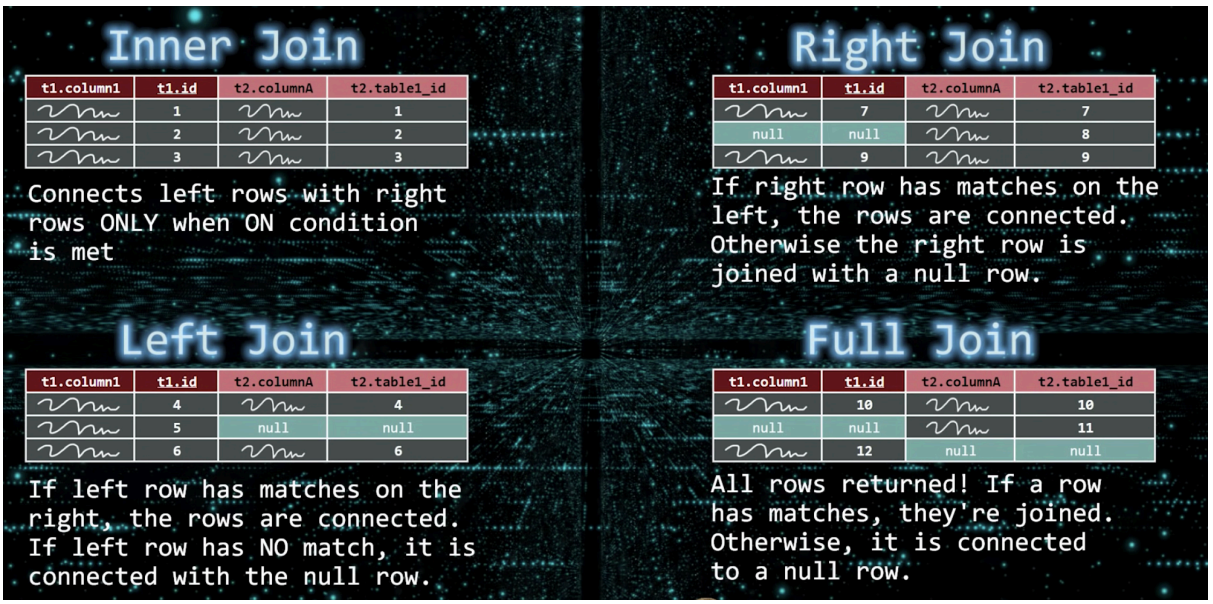
SQL

Databases:

- MySQL
- PostgreSQL
- Oracle
- IBM DB2

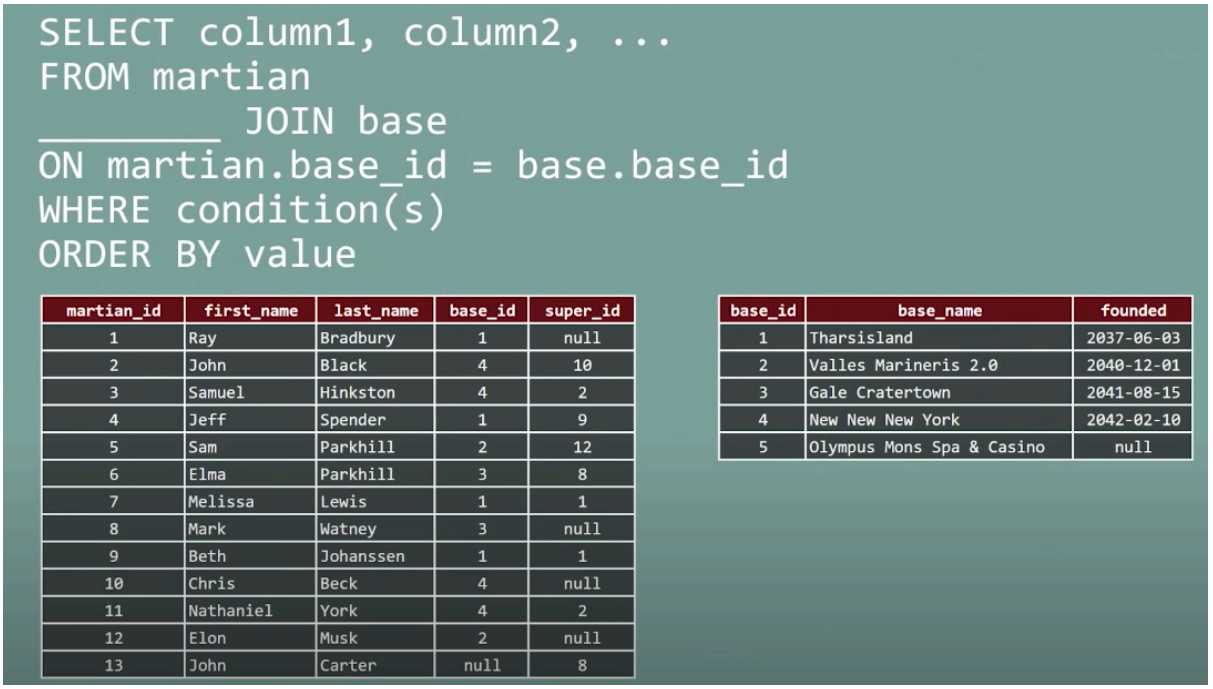
SQL Joins: https://www.youtube.com/watch?v=Jh_pvk48jHA

Note! MySQL and SQLite do not support Full Join (Outer Join)



In this example the match is based on the id of the two tables.

Sample Query of the Join, we just have to specify inner, left, right or full join in place of blank



martian_id	first_name	last_name	base_id	super_id
1	Ray	Bradbury	1	null
2	John	Black	4	10
3	Samuel	Hinkston	4	2
4	Jeff	Spender	1	9
5	Sam	Parkhill	2	12
6	Elma	Parkhill	3	8
7	Melissa	Lewis	1	1
8	Mark	Watney	3	null
9	Beth	Johanssen	1	1
10	Chris	Beck	4	null
11	Nathaniel	York	4	2
12	Elon	Musk	2	null
13	John	Carter	null	8

base_id	base_name	founded
1	Tharsisland	2037-06-03
2	Valles Marineris 2.0	2040-12-01
3	Gale Cratertown	2041-08-15
4	New New New York	2042-02-10
5	Olympus Mons Spa & Casino	null

Self Join: As the martian table has super_id which is the id of its superior we have to join the same table in order to generate a report of martian and their superior.

Joins

① Natural join :

- joins automatically based on common attribute (column)
- we don't specify `ON` clause.
- in the result the repeated column isn't shown.
  (in inner join the same columns are repeated)

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE
1132445	Walker	32145	231
1217782	Adares	32145	125
1312243	Rakowski	34129	167
1321242	Rodriguez	37134	125
1542311	Smithson	37134	421
1657399	Vanloo	32145	231

Table name: AGENT

AGENT_CODE	AGENT_PHONE
125	6152439887
167	6153426778
231	6152431124
333	9041234445

Result of natural join :

```
SELECT   select_list
FROM     T1
NATURAL [INNER, LEFT, RIGHT] JOIN T2
```

if nothing is specified default is INNER

```
select * from customer NATURAL JOIN agent
```

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	6152439887
1321242	Rodriguez	37134	125	6152439887
1312243	Rakowski	34129	167	6153426778
1132445	Walker	32145	231	6152431124
1657399	Vanloo	32145	231	6152431124

JOIN CLASSIFICATION	JOIN TYPE	SQL SYNTAX EXAMPLE	DESCRIPTION
CROSS	CROSS JOIN	SELECT * FROM T1, T2	Returns the Cartesian product of T1 and T2 (old style)
		SELECT * FROM T1 CROSS JOIN T2	Returns the Cartesian product of T1 and T2
INNER	Old-style JOIN	SELECT * FROM T1, T2 WHERE T1.C1=T2.C1	Returns only the rows that meet the join condition in the WHERE clause (old style); only rows with matching values are selected
	NATURAL JOIN	SELECT * FROM T1 NATURAL JOIN T2	Returns only the rows with matching values in the matching columns; the matching columns must have the same names and similar data types
	JOIN USING	SELECT * FROM T1 JOIN T2 USING (C1)	Returns only the rows with matching values in the columns indicated in the USING clause
	JOIN ON	SELECT * FROM T1 JOIN T2 ON T1.C1=T2.C1	Returns only the rows that meet the join condition indicated in the ON clause

all yield the same result & perform same

can be <, >, etc.

JOIN CLASSIFICATION	JOIN TYPE	SQL SYNTAX EXAMPLE	DESCRIPTION
OUTER	LEFT JOIN	SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C1	Returns rows with matching values and includes all rows from the left table (T1) with unmatched values
	RIGHT JOIN	SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T1.C1=T2.C1	Returns rows with matching values and includes all rows from the right table (T2) with unmatched values
	FULL JOIN	SELECT * FROM T1 FULL OUTER JOIN T2 ON T1.C1=T2.C1	Returns rows with matching values and includes all rows from both tables (T1 and T2) with unmatched values

— INNER JOIN : results in intersection of two tables

— OUTER JOIN : results in union of two tables.

② Left outer join :

output all rows of left (customer) table.

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	125	6152439887
1321242	Rodriguez	37134	125	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1132445	Walker	32145	231	231	6152431124
1657399	Vanloo	32145	231	231	6152431124
1542311	Smithson	37134	421	null	null

all vals from left

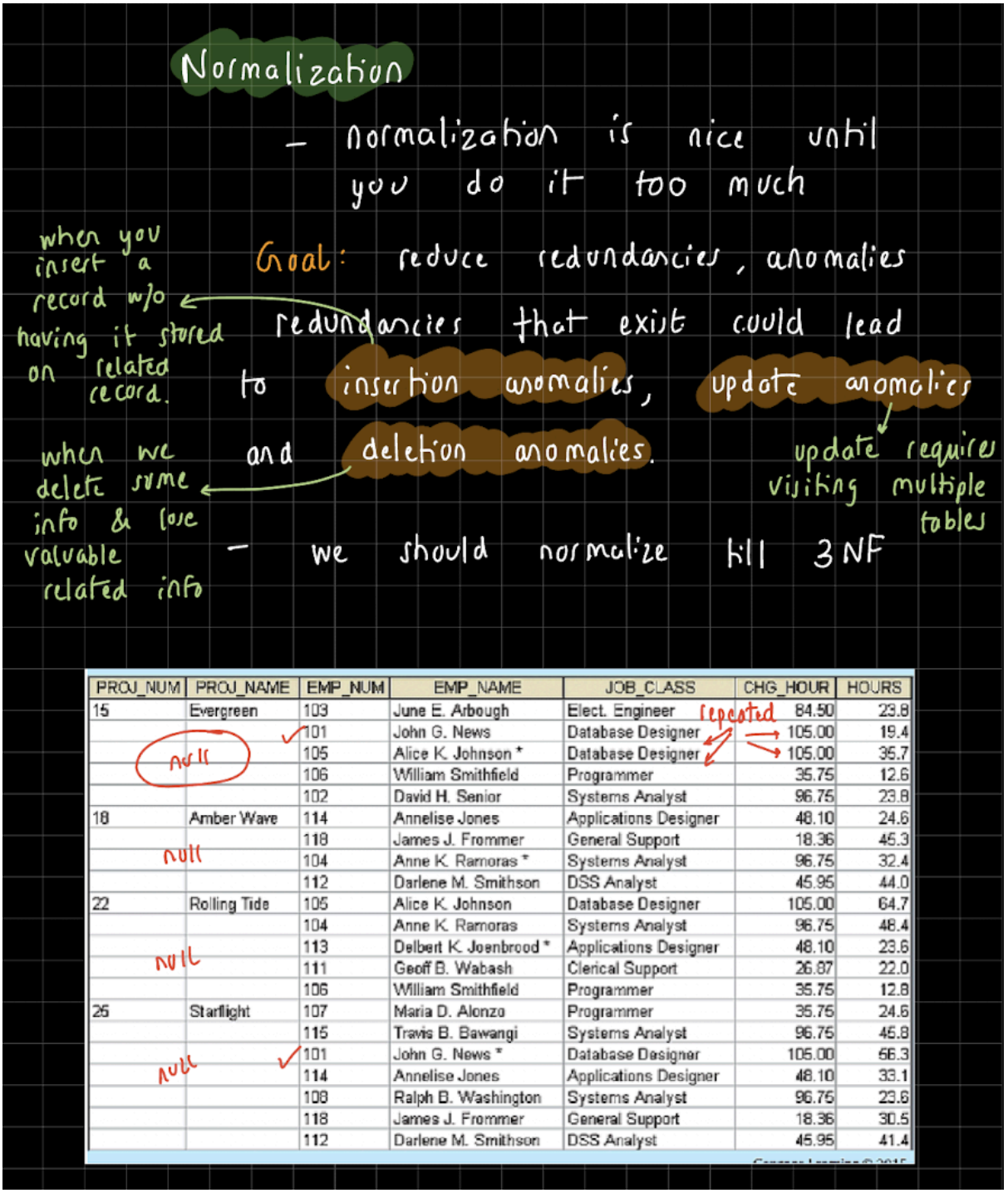
③ Right outer join :

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	125	6152439887
1321242	Rodriguez	37134	125	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1132445	Walker	32145	231	231	6152431124
1657399	Vanloo	32145	231	231	6152431124
null	null	null	null	333	9041234445

④ Self join (aliasing becomes important here as same fields are used to compare)

Normalization

Refer Notability > Database > Joins



PROJ_NUM	PROJ_NAME	EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR	HOURS
15	Evergreen	103	June E. Arbough	Elect. Engineer	84.50	23.8
		101	John G. News	Database Designer	105.00	19.4
		105	Alice K. Johnson *	Database Designer	105.00	35.7
		106	William Smithfield	Programmer	35.75	12.6
		102	David H. Senior	Systems Analyst	96.75	23.8
18	Amber Wave	114	Annelise Jones	Applications Designer	48.10	24.6
		118	James J. Frommer	General Support	18.36	45.3
		104	Anne K. Ramoras *	Systems Analyst	96.75	32.4
		112	Darlene M. Smithson	DSS Analyst	45.95	44.0
22	Rolling Tide	105	Alice K. Johnson	Database Designer	105.00	64.7
		104	Anne K. Ramoras	Systems Analyst	96.75	48.4
		113	Delbert K. Joenbrood *	Applications Designer	48.10	23.6
		111	Geoff B. Wabash	Clerical Support	26.87	22.0
		106	William Smithfield	Programmer	35.75	12.8
25	Starflight	107	Maria D. Alonzo	Programmer	35.75	24.6
		115	Travis B. Bawangi	Systems Analyst	96.75	45.8
		101	John G. News *	Database Designer	105.00	56.3
		114	Annelise Jones	Applications Designer	48.10	33.1
		108	Ralph B. Washington	Systems Analyst	96.75	23.6
		118	James J. Frommer	General Support	18.36	30.5
		112	Darlene M. Smithson	DSS Analyst	45.95	41.4

Functional dependency

$A \longrightarrow B$

B is functionally dependent on A

i.e (if we know the value of A, there can only be one value for B.)

Functional dependency

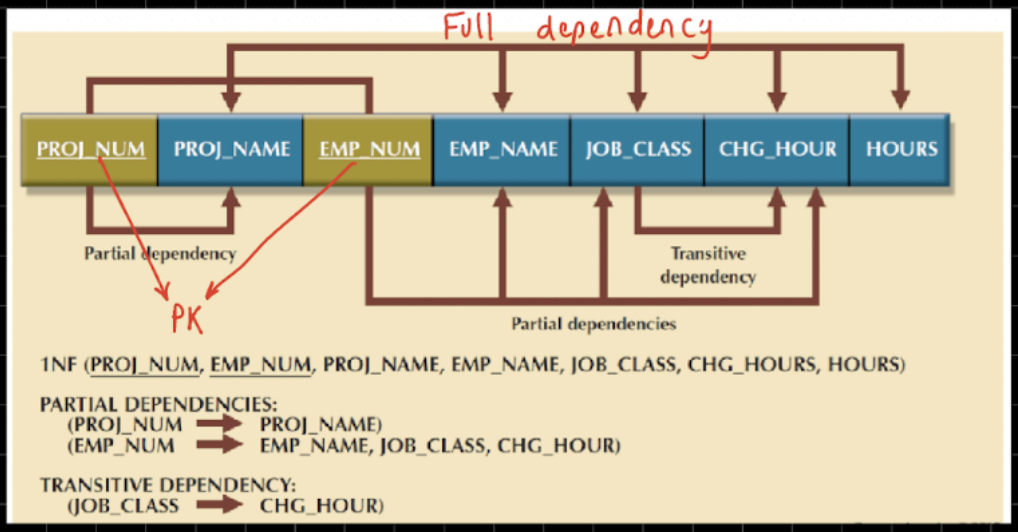
STU_ID[determinant] ->[functionally determines] STU_LNAME[dependent]

STU_ID,STU_LNAME -> GPA is NOT a 'full functional dependency' because the determinant contains an extra (unwanted) attr (STU_LNAME)

STU_LNAME,STU_FNAME -> GPA is a 'full functional dependency' (assuming lastname,firstname is unique)

Dependency diagram :

Top arrows good, bottom bad



① PROJ_NAME has only partial dependency on the PK (since it is only dependent on PROJ_NUM, which is a part of the PK)

② CHG_HOUR is dependent on JOB_CLASS (which is not a PK i.e a non-prime attribute).

Hence this dependency is transitive.

Issues with our db

PROJ_NUM	PROJ_NAME	EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR	HOURS
15	Evergreen	103	June E. Arbough	Elect. Engineer	84.50	23.8
		101	John G. News	Database Designer	105.00	19.4
		105	Alice K. Johnson *	Database Designer	105.00	35.7
		106	William Smithfield	Programmer	35.75	12.6
		102	David H. Senior	Systems Analyst	96.75	23.8
18	Amber Wave	114	Annelise Jones	Applications Designer	48.10	24.6
		118	James J. Frommer	General Support	18.36	45.3
		104	Anne K. Ramoras *	Systems Analyst	96.75	32.4
		112	Darlene M. Smithson	DSS Analyst	45.95	44.0
22	Rolling Tide	105	Alice K. Johnson	Database Designer	105.00	64.7
		104	Anne K. Ramoras	Systems Analyst	96.75	48.4
		113	Delbert K. Joenbrood *	Applications Designer	48.10	23.6
		111	Geoff B. Wabash	Clerical Support	26.87	22.0
		106	William Smithfield	Programmer	35.75	12.8
25	Starflight	107	Maria D. Alonzo	Programmer	35.75	24.6
		115	Travis B. Bawangi	Systems Analyst	96.75	45.8
		101	John G. News *	Database Designer	105.00	56.3
		114	Annelise Jones	Applications Designer	48.10	33.1
		108	Ralph B. Washington	Systems Analyst	96.75	23.6
		118	James J. Frommer	General Support	18.36	30.5
		112	Darlene M. Smithson	DSS Analyst	45.95	41.4

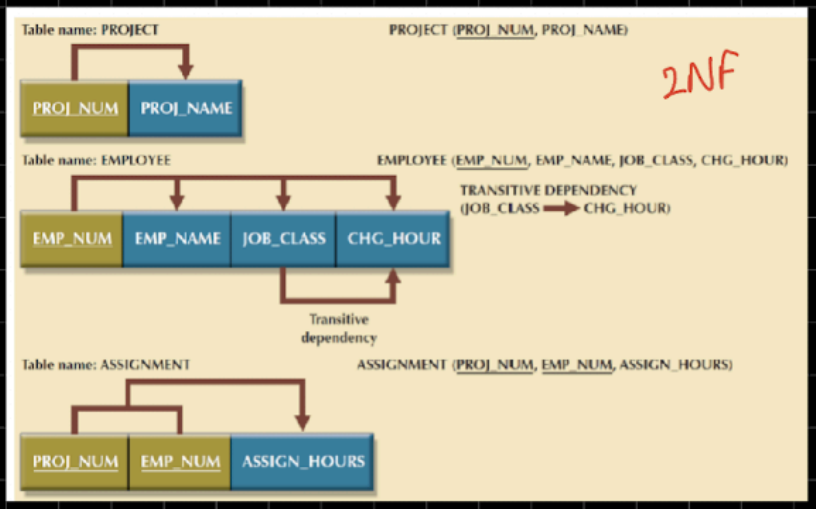
- PROJ_NUM could be used as PK (part of it) but it contains null.
- possibilities of data inconsistencies
- redundancies that exist, could lead to anomalies
- wasted storage space due to redundancies.

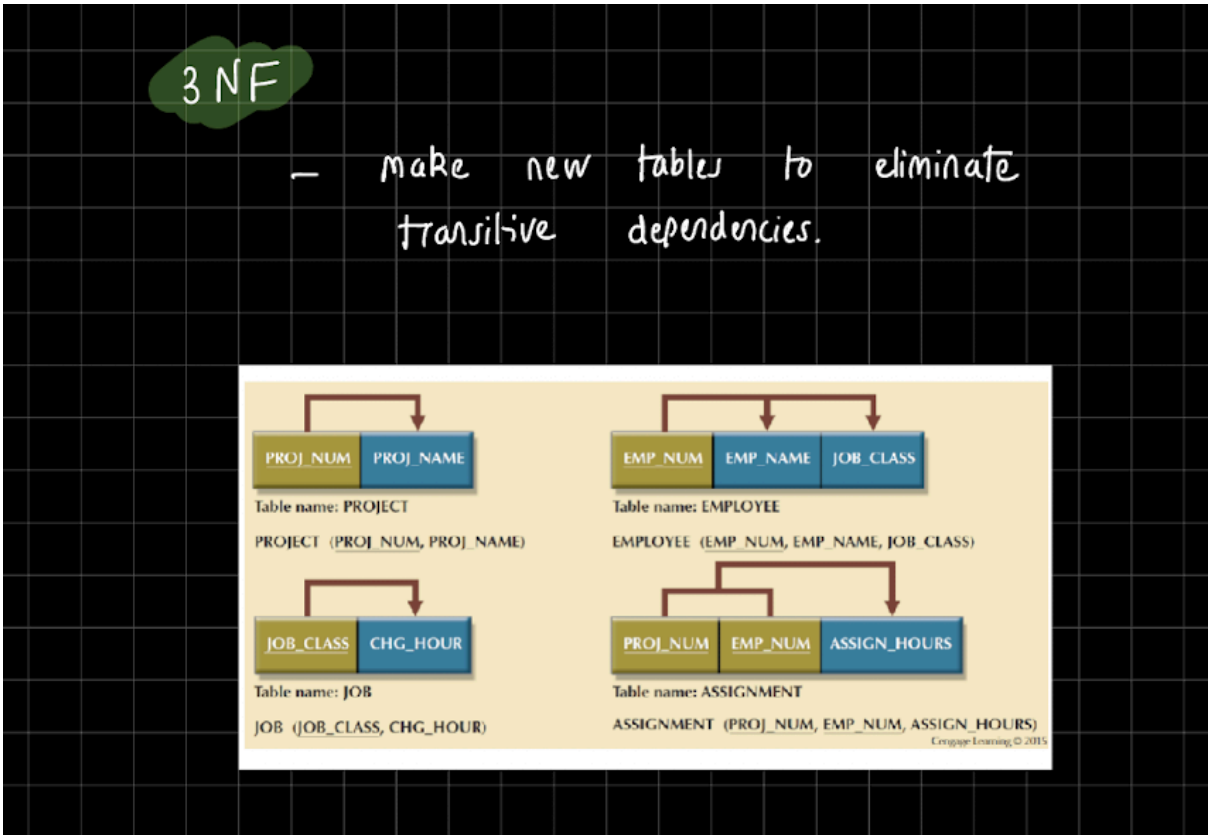
1NF

- All key attributes are defined
- There are no repeating groups in the table
- All attributes are dependent on primary key.

2NF

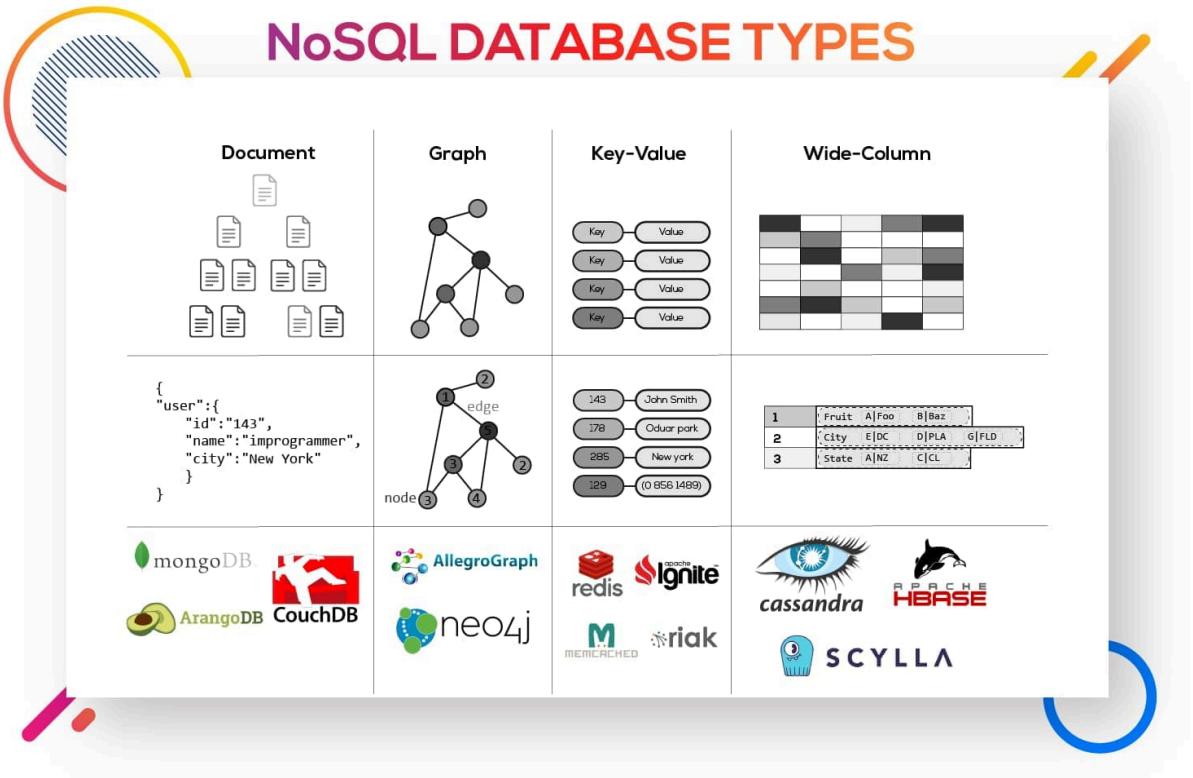
- make new tables to eliminate partial dependencies




3NF
– make new tables to eliminate transitive dependencies.
PROJ_NUM
PROJ_NAME
Table name: PROJECT
PROJECT (PROJ_NUM, PROJ_NAME)
EMP_NUM
EMP_NAME
JOB_CLASS
Table name: EMPLOYEE
EMPLOYEE (EMP_NUM, EMP_NAME, JOB_CLASS)
JOB_CLASS
CHG_HOUR
Table name: JOB
JOB (JOB_CLASS, CHG_HOUR)
PROJ_NUM
EMP_NUM
ASSIGN_HOURS
Table name: ASSIGNMENT
ASSIGNMENT (PROJ_NUM, EMP_NUM, ASSIGN_HOURS)
Cengage Learning © 2015

Sharding

NoSQL (Not only SQL)



Advantages:

1. Data is contained together. Joins are not required, which are expensive.
2. Adding a new attribute is easier. In RDBMS, we have to lock the dbs to add new attributes(columns).
3. Built for aggregations.

Disadvantages:

1. ACID is not guaranteed.

Some general points related to NoSQL:

- NoSQL is typically good for unstructured/"schemaless" data - usually, you **don't need to explicitly define your schema** up front and can just **include new fields without any ceremony**
- NoSQL typically favours a **denormalized schema** due to no support for JOINs per the RDBMS world. So you would usually have a flattened, denormalized representation of your data.
- Using NoSQL doesn't mean you could lose data. Different DBs have different strategies. e.g. MongoDB - you can essentially choose what level to trade off performance vs potential for data loss - best performance = greater scope for data loss.

- It's often very **easy to scale out** NoSQL solutions. Adding more nodes to replicate data is one way to a) offer more scalability and b) offer more protection against data loss if one node goes down. But again, it depends on the NoSQL DB/configuration. NoSQL does not necessarily mean "data loss" like you infer.
- IMHO, **complex/dynamic queries/reporting are best served from an RDBMS**. Often the query functionality for a NoSQL DB is limited.
- It doesn't have to be a 1 or the other choice. My experience has been using RDBMS in conjunction with NoSQL for certain use cases.
- NoSQL DBs often **lack the ability to perform atomic operations** across multiple "tables".

SQL vs NoSQL

SQL	NoSQL
Pre-defined Schema	Dynamic Schema
Good for complex queries	Not a best fit for complex queries
Vertically Scalable	Horizontally Scalable
ACID complaint	Most NoSQL DBs compromise ACID properties.

ACID is valid for a standalone SQL db.

NoSQL databases

.

MongoDB

MongoDB picks Consistency and Partition Tolerance, hence **mongodb has master-slave architecture**. In this approach, there might be many servers handling requests, but only one server

can actually perform writes so as to maintain data in a consistent state.

- As mongodb is a CP system, it has a single master that handles all the write requests. (unlike the nodes in cassandra cluster)

MongoDB performs dirty reads by default.

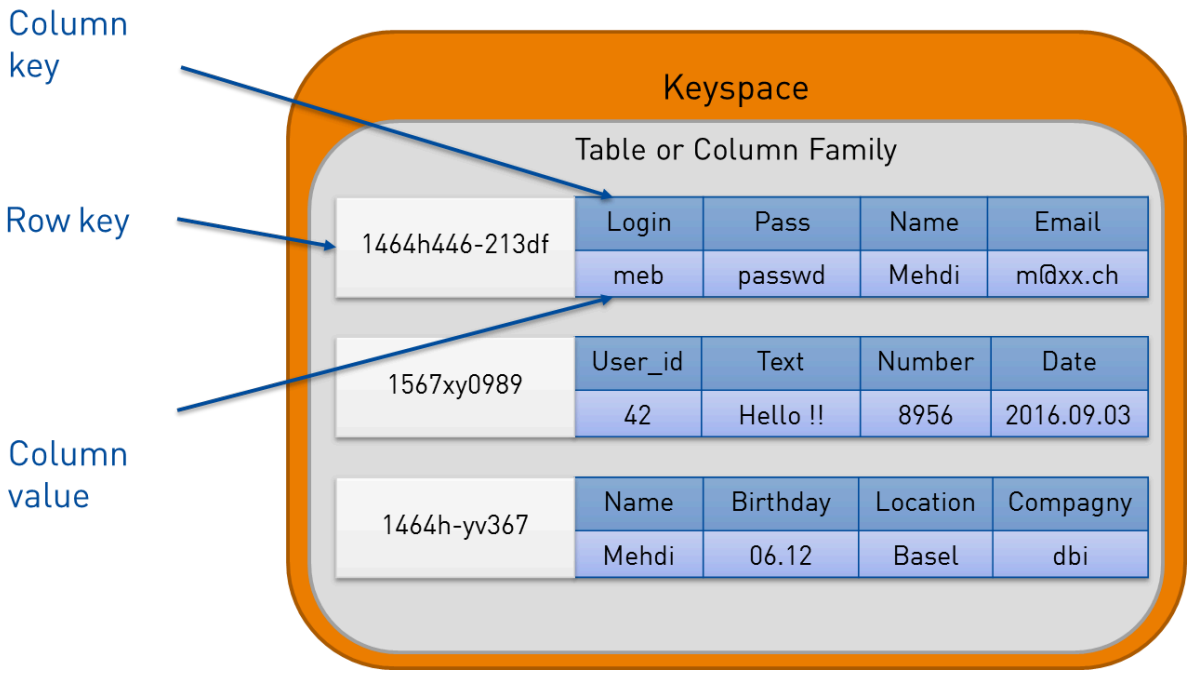
Cassandra

Cassandra is a **highly available**, **distributed**, partitioned row store. In Cassandra, rows of data are stored based on the hashed value of the partition key, called a **token**. **Each node in the cluster is assigned multiple token ranges**, and rows are stored on nodes that are responsible for their tokens.

- Cassandra is designed such that it has **no master or slave nodes (i.e. peer to peer).**
- Data is automatically distributed across all the nodes.
- Similar to HDFS, data is replicated across the nodes for redundancy.
- Data is kept in memory and lazily written to the disk.

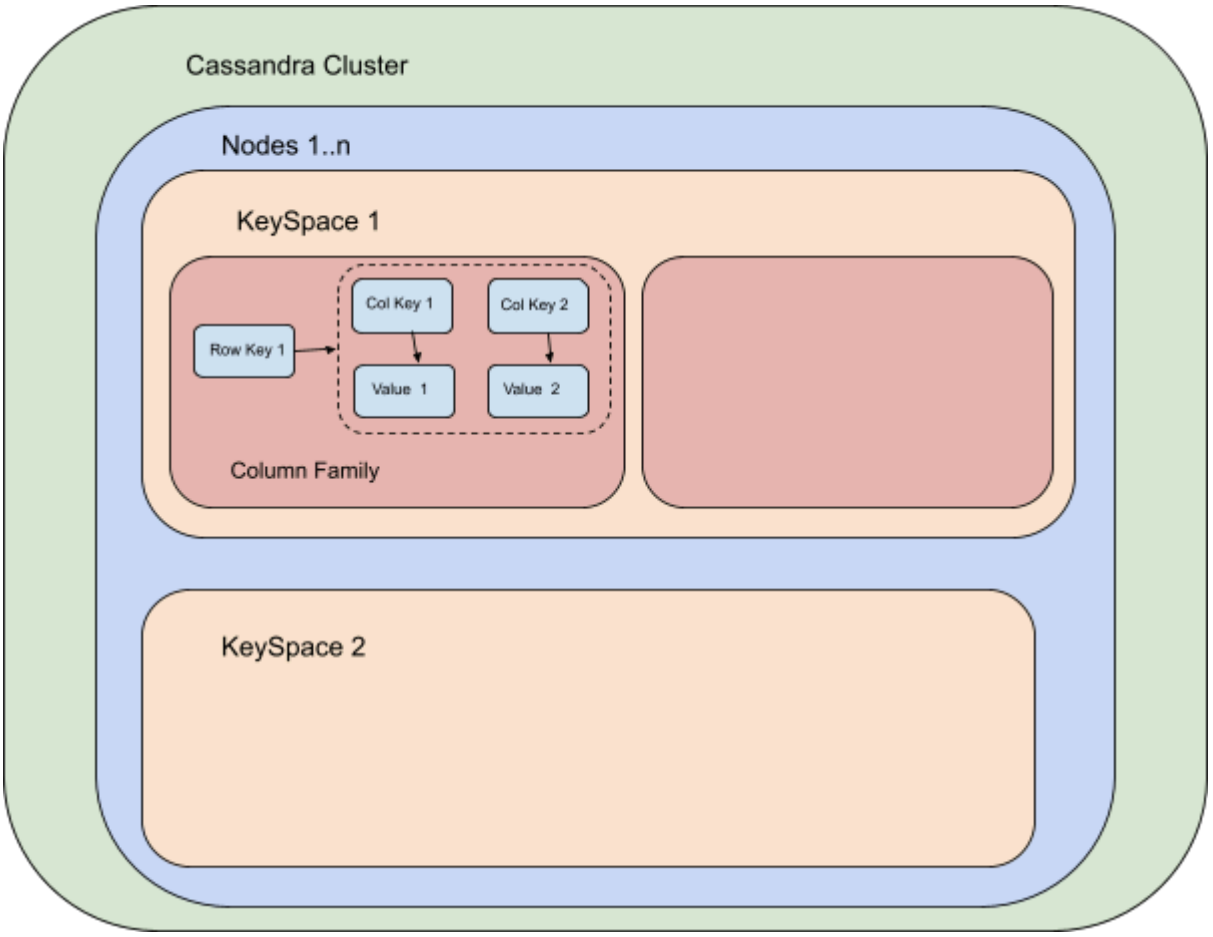
Cassandra Data Model

Column : A single piece of data.
Row: Stores related data
Column Family: A set of rows
Keyspace



Cassandra data model is a Nested Map:

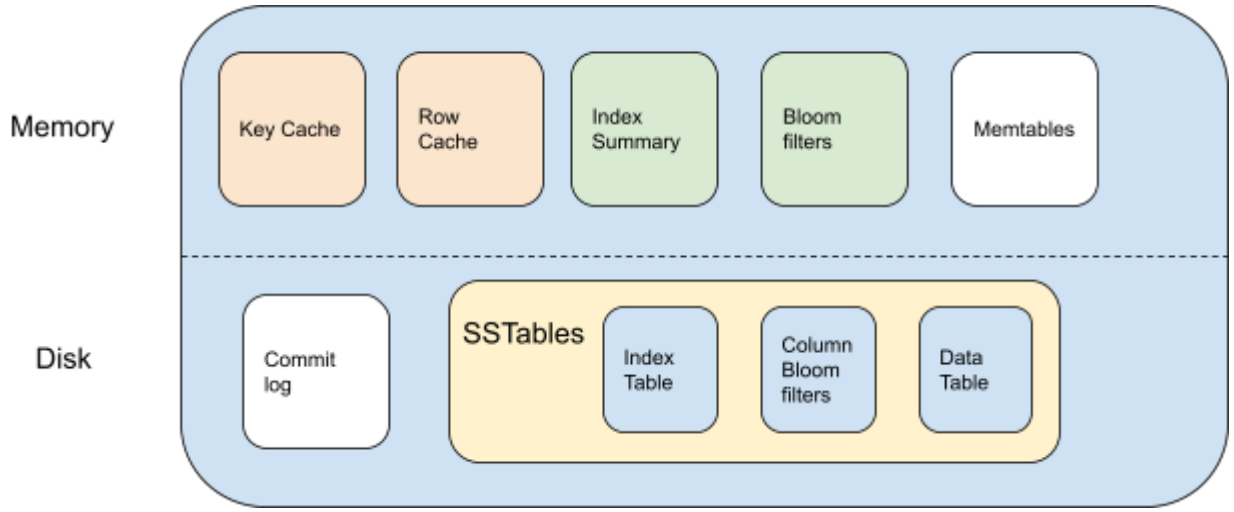
Map<row_key,SortedMap<col_key,col_value>>



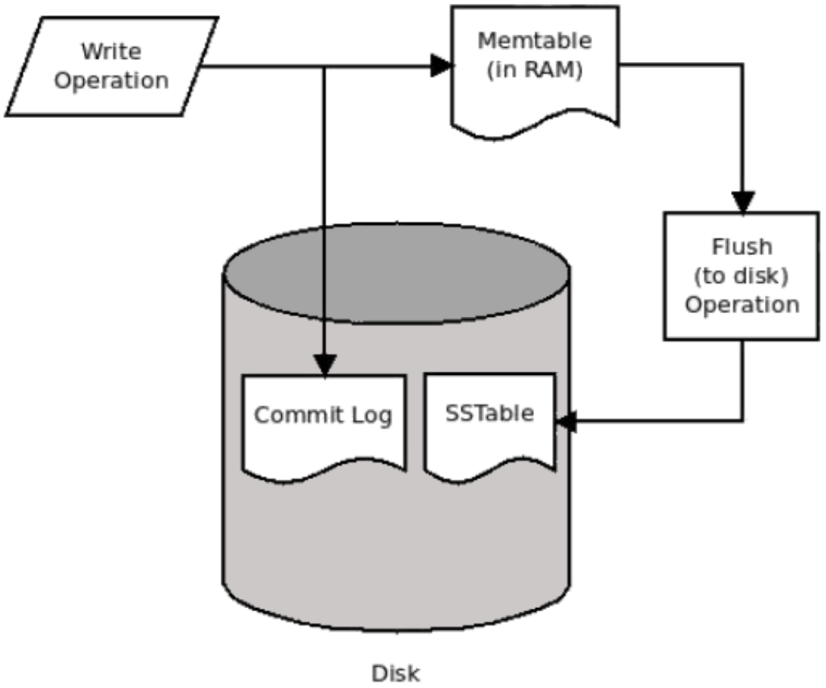
In the image above, col key 1 is the first *name* and value 1 is *John.*
Row key uniquely identifies a row in a list of rows.

When data is written, it is replicated to multiple instances. During this process, it is possible that one (or more) of the replica writes could fail. **Reads for this data would then return stale or dirty results.** However, one of the design choices for Cassandra was that the possibility of serving stale data outweighed the risk of a request failing.

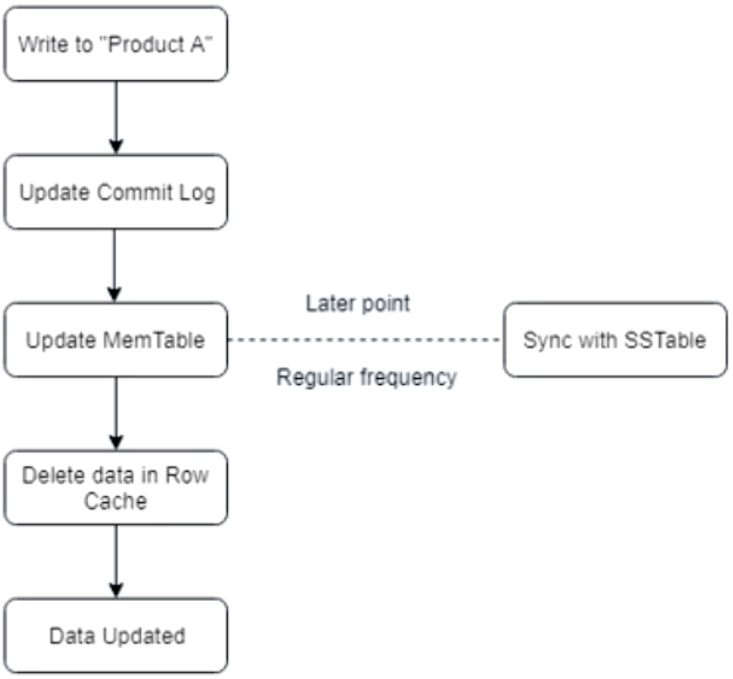
Storage Components:



Cassandra Data Write Path



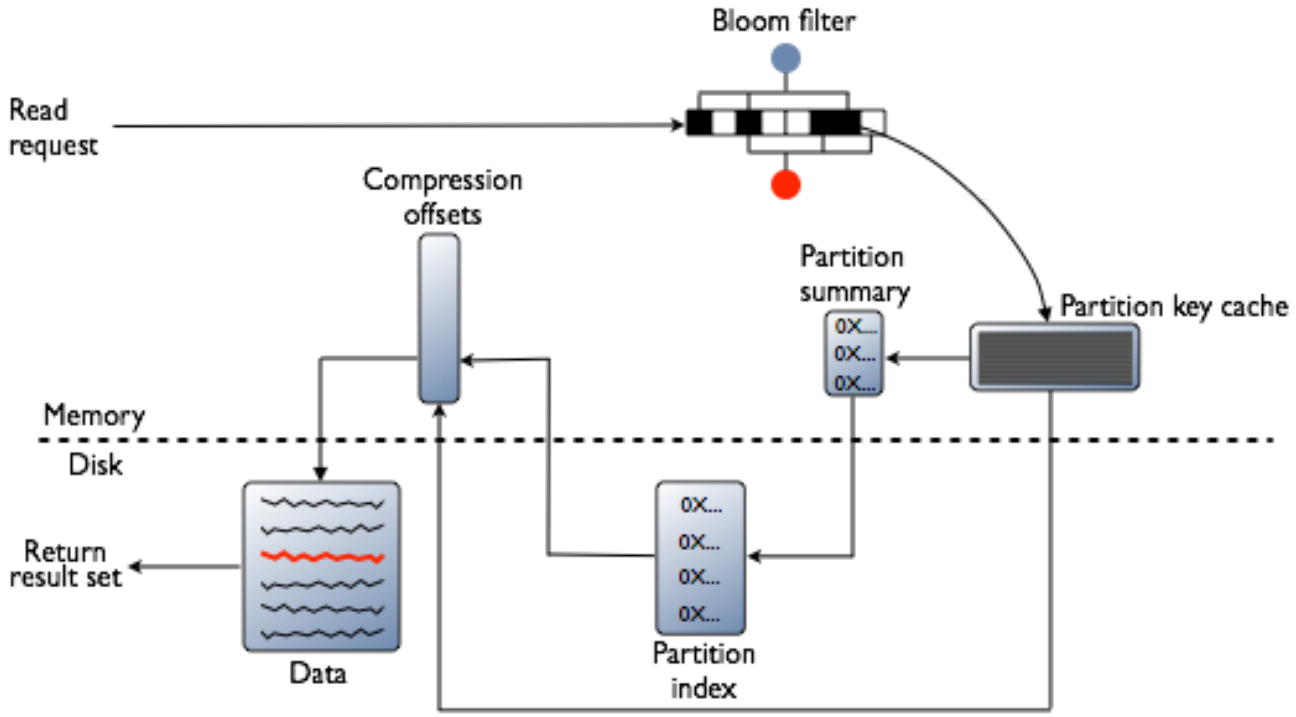
1. There is an in-memory structure known as a **memtable**(resembles a write-back cache), which gets the write. Additionally, the new data is written to the **commit log(happens first)**, which is on-disk.(*The commit log is Cassandra's way of enforcing durability in the case of a plug-out-of-the-wall event. When a node is restarted, the commit log is verified against what is stored on-disk and replayed if necessary.*)
2. Once a flush of the memtable is triggered, the data stored in memory is written to the sorted string table files (**SSTables**) on-disk.



Delete

Cassandra has a soft delete type of policy. Add a **tombstone** to the commit log and eventually delete at compaction.

Cassandra Read Path



Note: A new SSTable is created when the memtable is flushed to the disk. Hence a key can reside in multiple SSTables. Those SStables are merged(compacted) periodically.

A write is passed to other nodes using **Gossip Protocol**.

Writes in a custer:

- Need to be lock-free for heavy workloads. Preferably no disk access at all for the write in the critical write path.
- Client sends a write request to a coordinator node in Cassandra cluster. Coordinator uses Partitioner to send a query to all replica nodes responsible for the key. Coordinator is elected by the Zookeeper.
- When X replicas respond, the coordinator returns an acknowledgement to the client.
- Always writable: **Hinted Handoff mechanism:** (When a node reports another as down, the node that is still up will store structures, known as **hints**, for the down node. Hints are essentially writes meant for one node that are temporarily stored on another. When the down node returns to a healthy status, the hints are then streamed to that node in an attempt to re-sync its data.)

Reads in a cluster:

- Coordinator sends read requests to X replicas.
- When X replicas respond, coordinator returns the latest-timestamped value from among those X.
- Coordinator also fetches values from other replicas, then initiates a **read repair** if any two values are different.

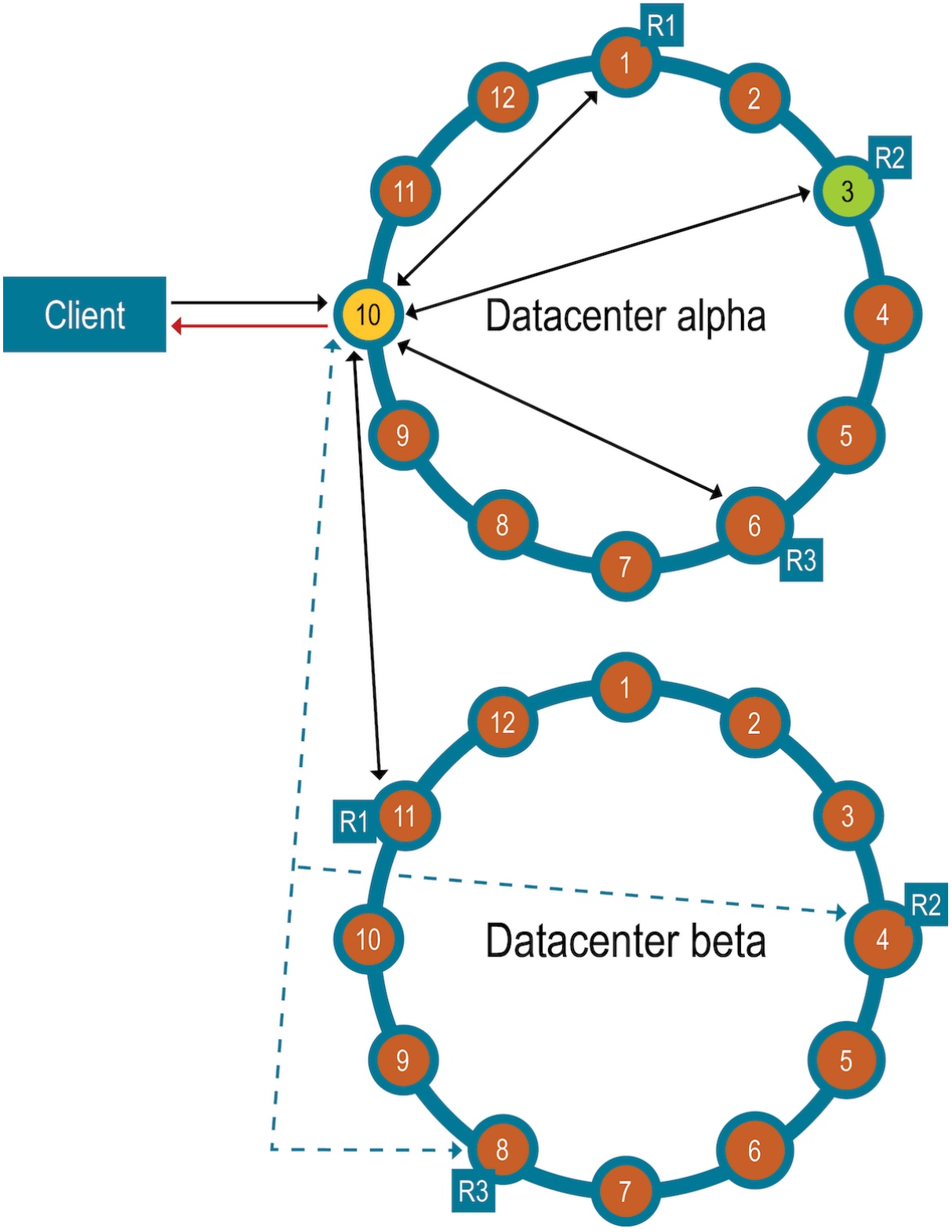
Mystery of X (in Cassandra): Client may choose consistency level for each operation (read/write)
ANY: any server (may not be replica), fastest
ALL: all replicas, ensures strong consistency but slowest
ONE: at least one replica, faster than ALL but cannot tolerate a failure
QUORUM: quorum across all replicas in all DCs
LOCAL_QUORUM: quorum in coordinator's DC
EACH_QUORUM: quorum in every DC

e.g. A two-datacenter cluster with a consistency level of QUORUM

In a two-datacenter cluster with a replication factor of 3 and a read consistency of **QUORUM**, 4(2+2) replicas for the given row must respond to fulfill the read request. The 4 replicas can be from any datacenter. In the background, the remaining replicas are checked for consistency with the first four. If

needed, a read repair is initiated for the out-of-date replicas.

Multiple datacenter cluster with 3 replica nodes and consistency level set to **QUORUM**



Coordinator node

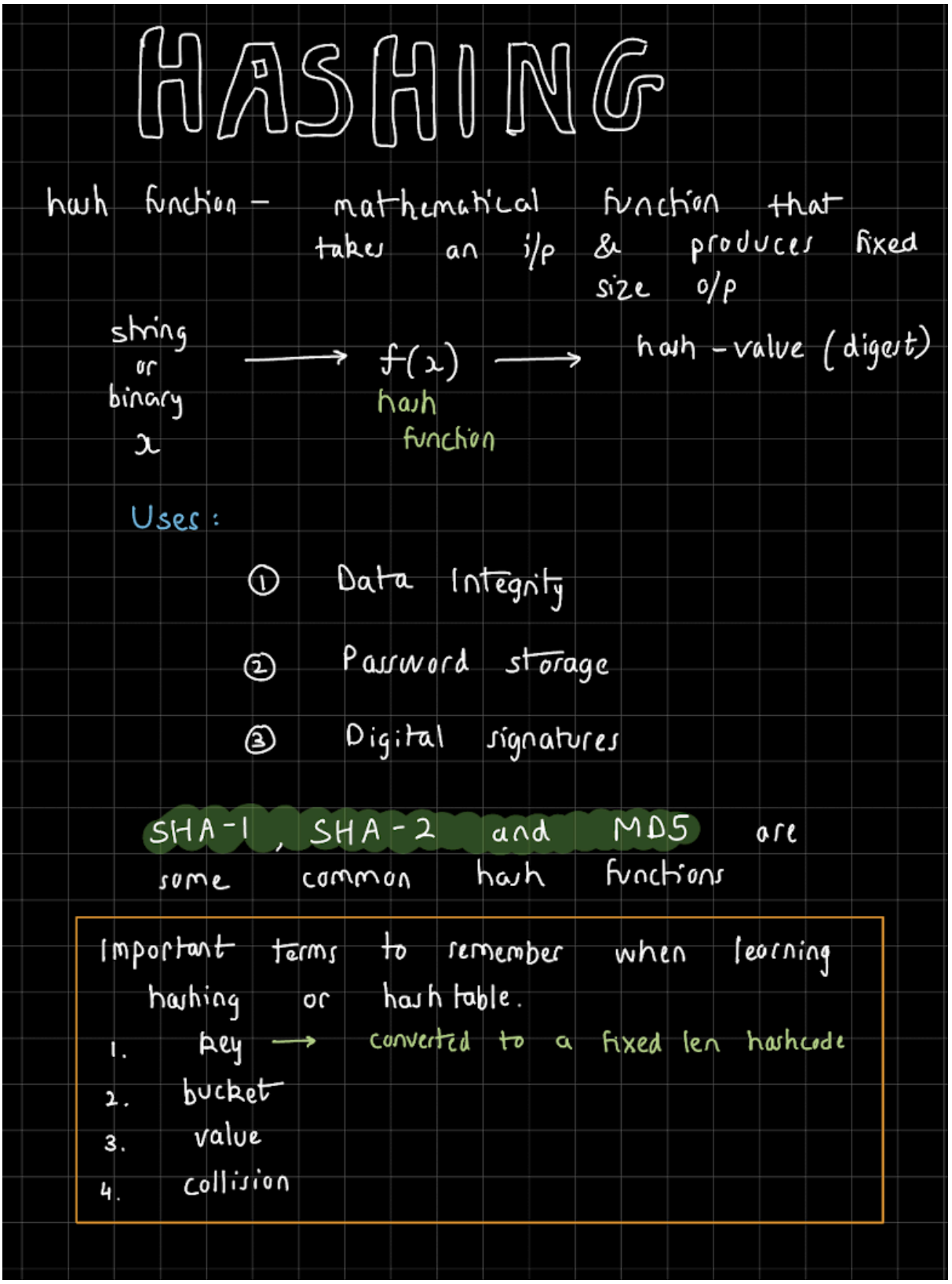
Chosen node

Read response

Read repair

Hashing

Awesome article on hashing: [CS106B Hashing]



If no. of collisions are high the worst case runtime is $O(N)$

key $\xrightarrow{\text{prehash}}$ hash code $\xrightarrow{\text{compression}}$ $[0, N-1]$

- Can we perform findMax(k) or findMin(k) ?
  → No, we would have to search entire table.

- We can't have two keys sharing the same bucket, so we have a collision.

Handling collision

1. Chaining : instead of storing key in the bucket, we store a pointer to a linked list which stores $\langle K, V \rangle$ pairs.
   (we have to be careful that L.Ls are not too long)

A good hash-table can be quantified using 'load - factor'

$$\text{Load Factor} = \frac{n}{N}$$

n → no. of elements

N → no. of buckets

WARNING If the load factor gets too big!
→ we go through the entire table and re-hash all keys to the new-table
(this is a big-penalty, but worth it to keep load factor low)

Bloom filters

- Probabilistic data structure

[Bloom Filters by Example]

Applications:

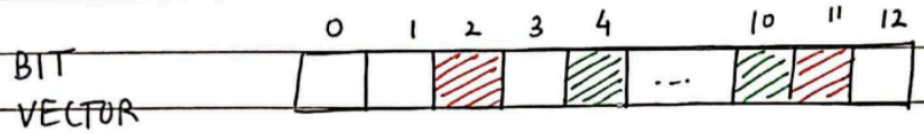
1. Used to query data in Hbase, Cassandra.(as indexes on distributed chunks of data.They use the filter to keep track of which rows or columns of the database are stored on disk, and avoid a (costly) disk access for non-existent attributes.)
2. Username present or not.
3. Malicious url?
4. Medium uses Bloom filters to avoid recommending articles a user has previously read.
5. Ethereum uses Bloom filters for quickly finding logs on the Ethereum blockchain.

Testing membership in bloom filter:

You simply hash the string with the same hash functions, then see if those values are set in the bit vector. If they aren't, you know that the element isn't in the set. If they are, you only know that it might be, because another element or some combination of other elements could have set the same bits.

BLOOM FILTERS

- space efficient probabilistic data structure that is used to test whether an element is member of set.



hash_1("CAT") → 2
hash_2("CAT") → 11

hash_1("DOG") → 4
hash_2("DOG") → 10

is_present("RAT") → MAY BE 87%
is_present("MAN") → NO

① If the bloom filter outputs NO, it is 100% definite that the element is not a part of the set.

② As the bit array becomes full, number of false positives increase. Hence we need to increase the len(bit vector).

③ How many hash functions? More the hash functions, slower the filter and quicker it fills up. If you have too few hash functions, you may suffer too many false positives.

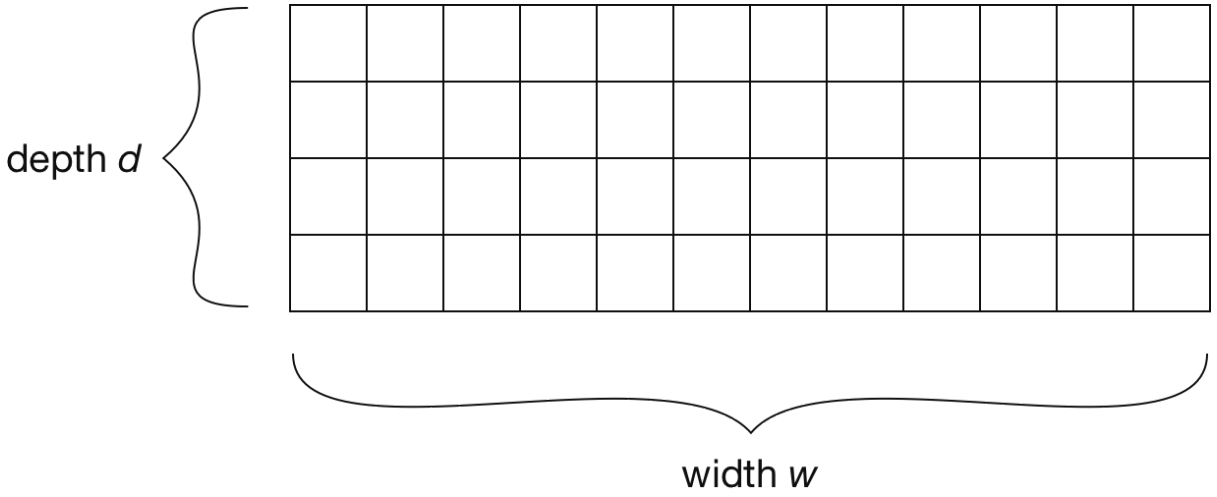
Count Min Sketch

- Another probabilistic data structure.

Applications:

1. Finding heavy hitters.
2. Database query planning: *Part of finding good execution strategies is estimating the table sizes yielded by certain subqueries. For example, given a join, such as the one below, we want to find out* ***how many rows the result will have****.*

```
SELECT *
FROM a, b
WHERE a.x = b.x
```



d are the number of hash functions.
w is usually in the 1000s.(calculated on how accurate we want the results)

1. Initialization: $\forall i \in \{1,\dots,d\}, j \in \{1,\dots,w\}$,:**count[i,j]=0**
2. Increment count (of element a): $\forall i \in \{1,\dots,d\}$: **count[i,hi(a)]+=1**
3. Retrieve count (of element a): **min=count[i,hi(a)]**

We take the min(count) to minimize the count of collisions.

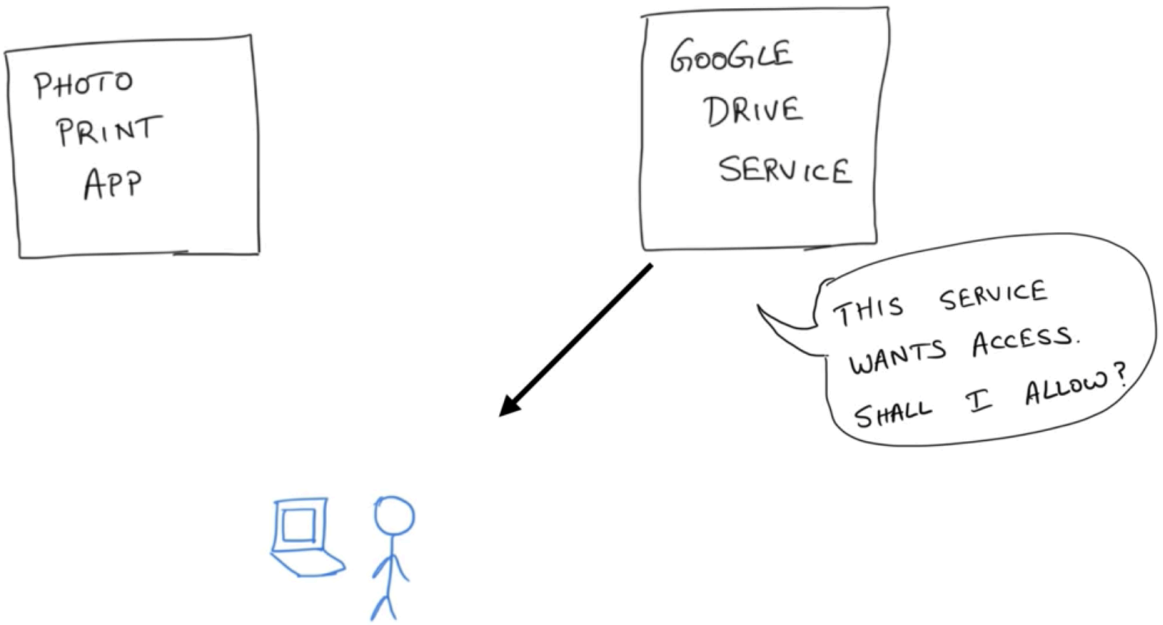
Microservices Authentication and Authorization Solutions

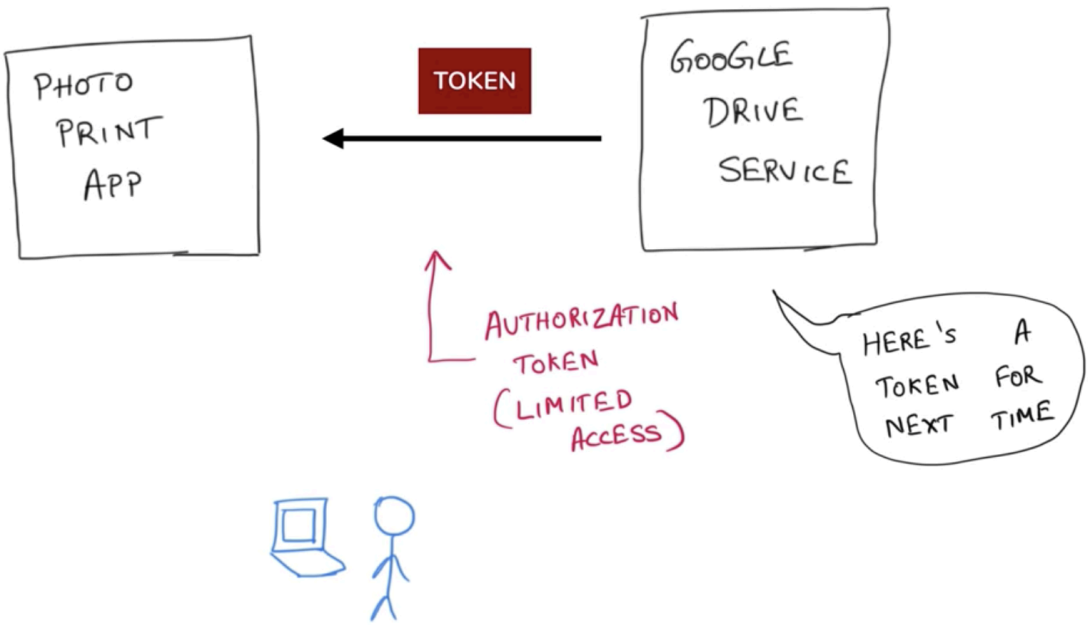
[Microservices Authentication and Authorization Solutions | by Mina Ayoub | tajawal](#) (Best article so far)

OAuth

OAuth is a **delegation** (or usually be misled to be an authorization) **protocol**, in other words, **OAuth is about how to get token and how to use a token.** OAuth doesn't define the token format, JWT is one of the token format largely used inside of the OAuth ecosystem.

- Problem solved is **Authorization and NOT Authentication.**
- Authorization mechanism where services can authorize on our behalf.





Now, everytime **Photo Printing Service** needs access to the Drive, it **hands over the token** to authorize itself.

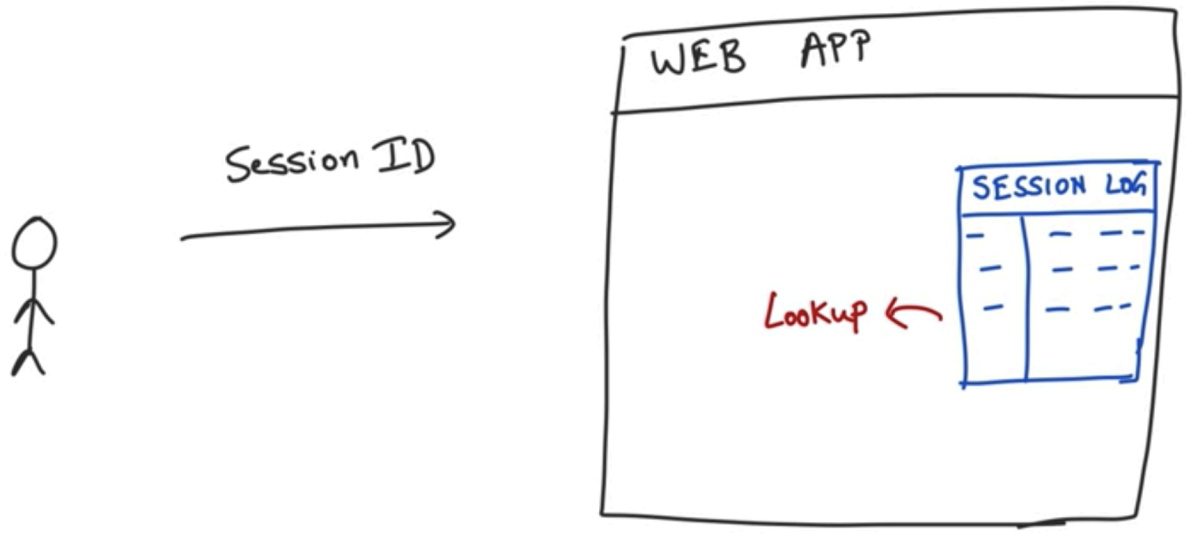
Authorization Strategies

1. JWT(JSON Web tokens) -> Value tokens
2. Session tokens -> Reference tokens

Traditional Way!

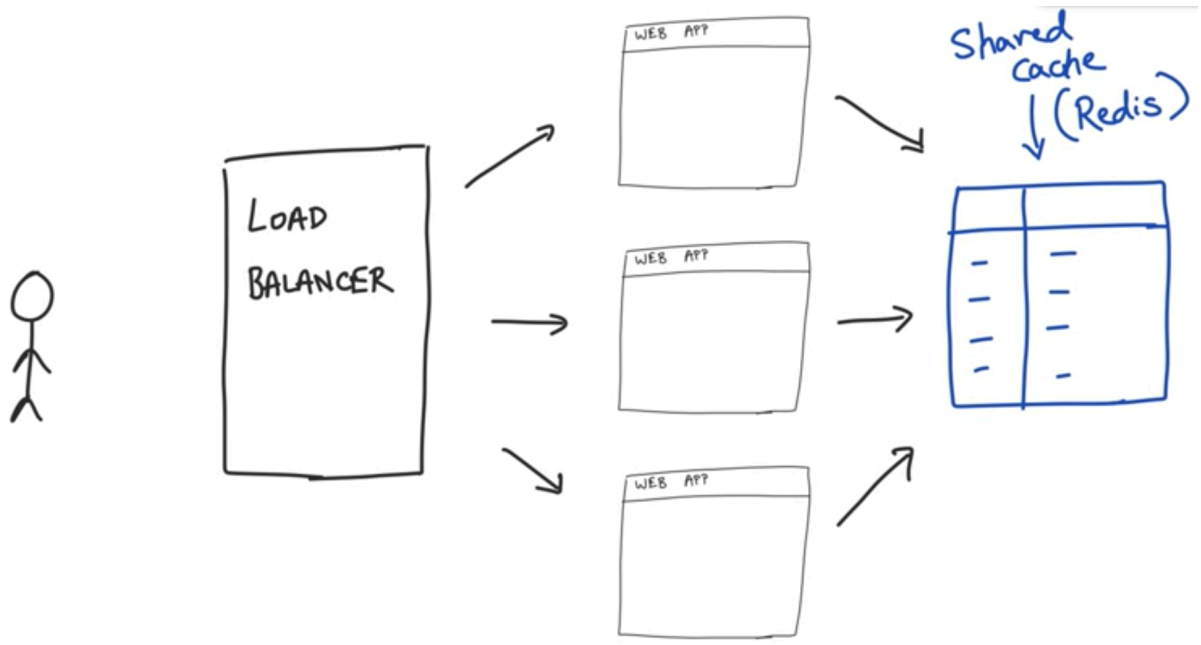
As HTTP is stateless it does not remember any previous information. In order for a user to authorize, the web server can maintain session logs for the users and it can return **Session ID** as the reference.

Session ID can be saved in the cookie and sent as a header everytime we access a resource.We use **cookies** which are special http headers that once set, the browser will keep sending along with every request.



There are drawbacks to this approach:

1. **locality of session log on the server is not guaranteed** as there may be multiple instances.
2. We can implement a **Shared Session cache using Redis** but it can be a SPOF.

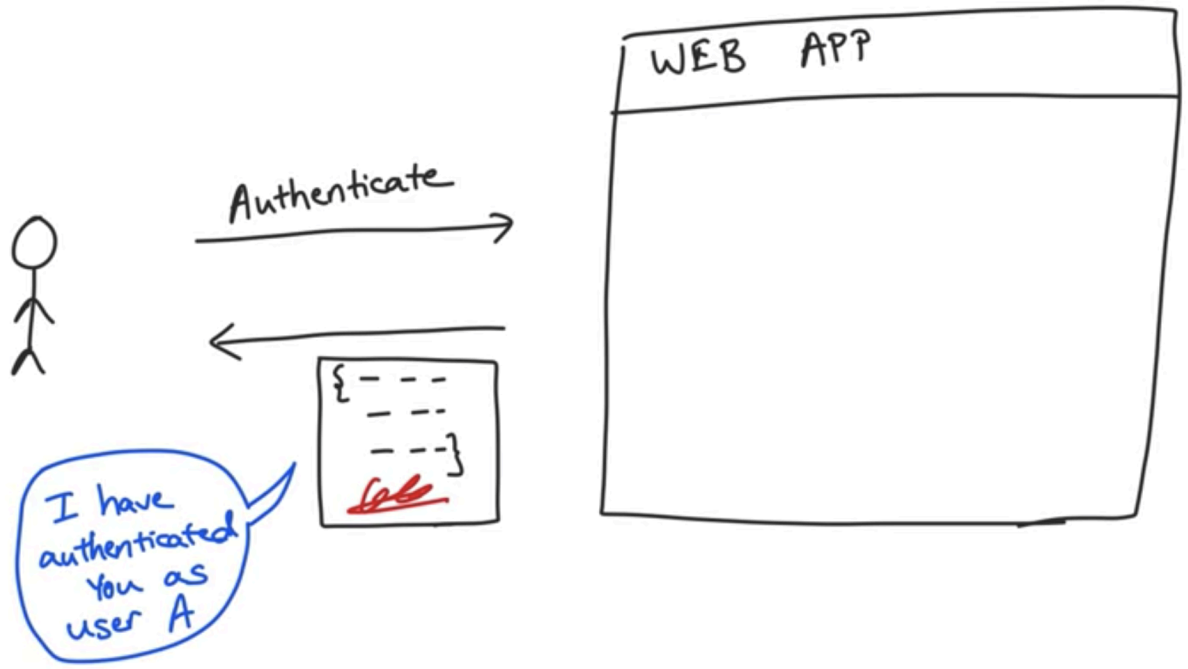


3. We can direct requests from a client to a particular server(consistent hashing) but this is tricky in microservice architecture.

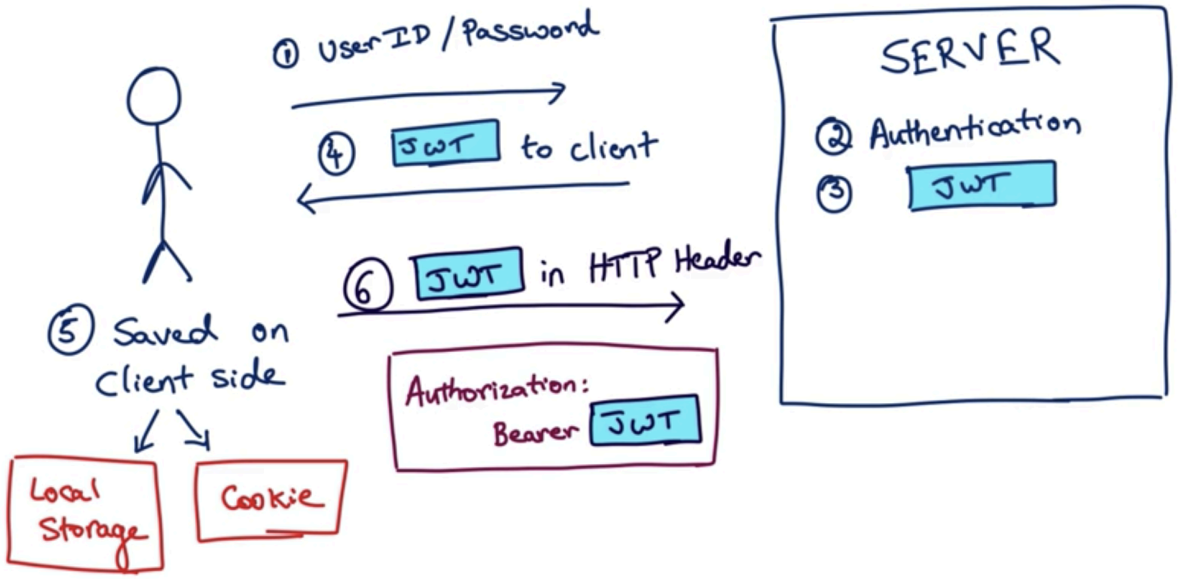
New!
Using JWT(used for Authorization only)

Instead of returning Session ID. The **server returns the entire user information**. This piece of information is signed to secure it. This cuts out dependency of the web server to manage the session details.

Client sends this whole JSON token on every subsequent request.



Format of JWT: <*header*>.<*payload*>.<*signature*>



Logging in Microservices

- logging library.
- zipkin i.e. distributed tracing system.

Redis

- **NoSQL in-memory database.** (key-value store). In SQL database adding a field(column) to the table causes an outage.
- **Single-threaded.**
- **Optional Durability:**
  Journaling (append only log): Whenever I write a key, go ahead and write in-memory and also write to commit log.
  Snapshotting: Every 2s flush to disk.

  Both of them happen asynchronously in the background.

- **Transport protocol:** TCP
- **Pub/Sub:** Redis supports publish subscribe model. Redis uses the **PUSH model instead of long-polling**.

How does redis expire keys when we set an expiry? (for e.g. set nametemp “Edmond” EX 10)

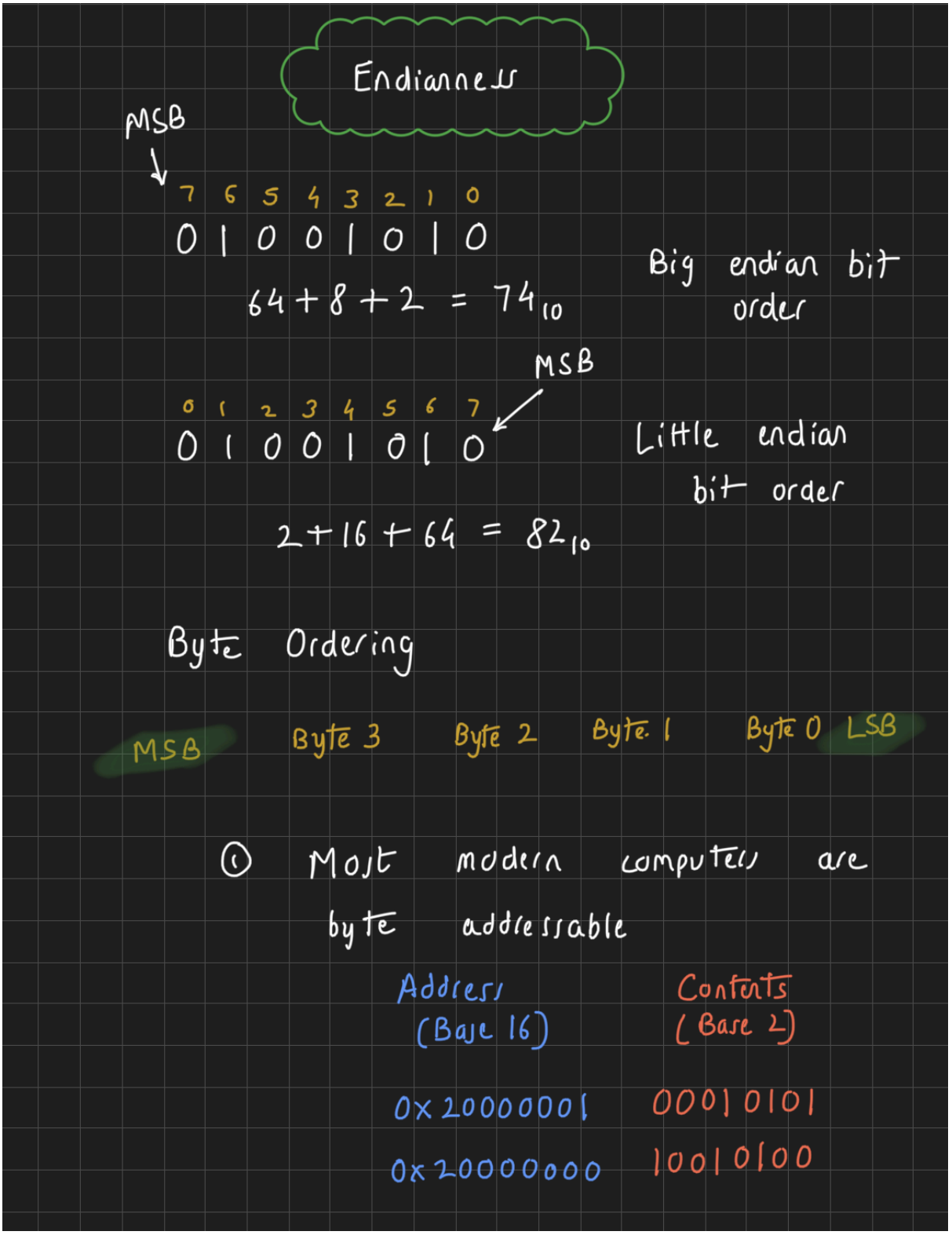
- Now, the expiration mechanism itself is semi-lazy. **Lazy expiration** means that you don't actually expire the objects until they are read. When reading an object, we check its expiration timestamp, and if it's in the past, we return nothing, and delete the object while we're at it. But the problem is that if a key is never touched, it just takes up memory for no reason.
  So Redis adds a second layer of **random active expiration**. It just reads random keys all the time, and when an expired key is touched it is deleted based on the lazy mechanism. This does not affect the expire behavior, it just adds "garbage collection" of expired keys.
- **NEW!!** The expiration cycle has been rewritten in Redis 6.0 to allow for much faster expirations that more closely match the time-to-live (TTL) property. **Redis 6** expiration will no longer be based on random sampling but will take **keys sorted by expiration time in a radix tree**.

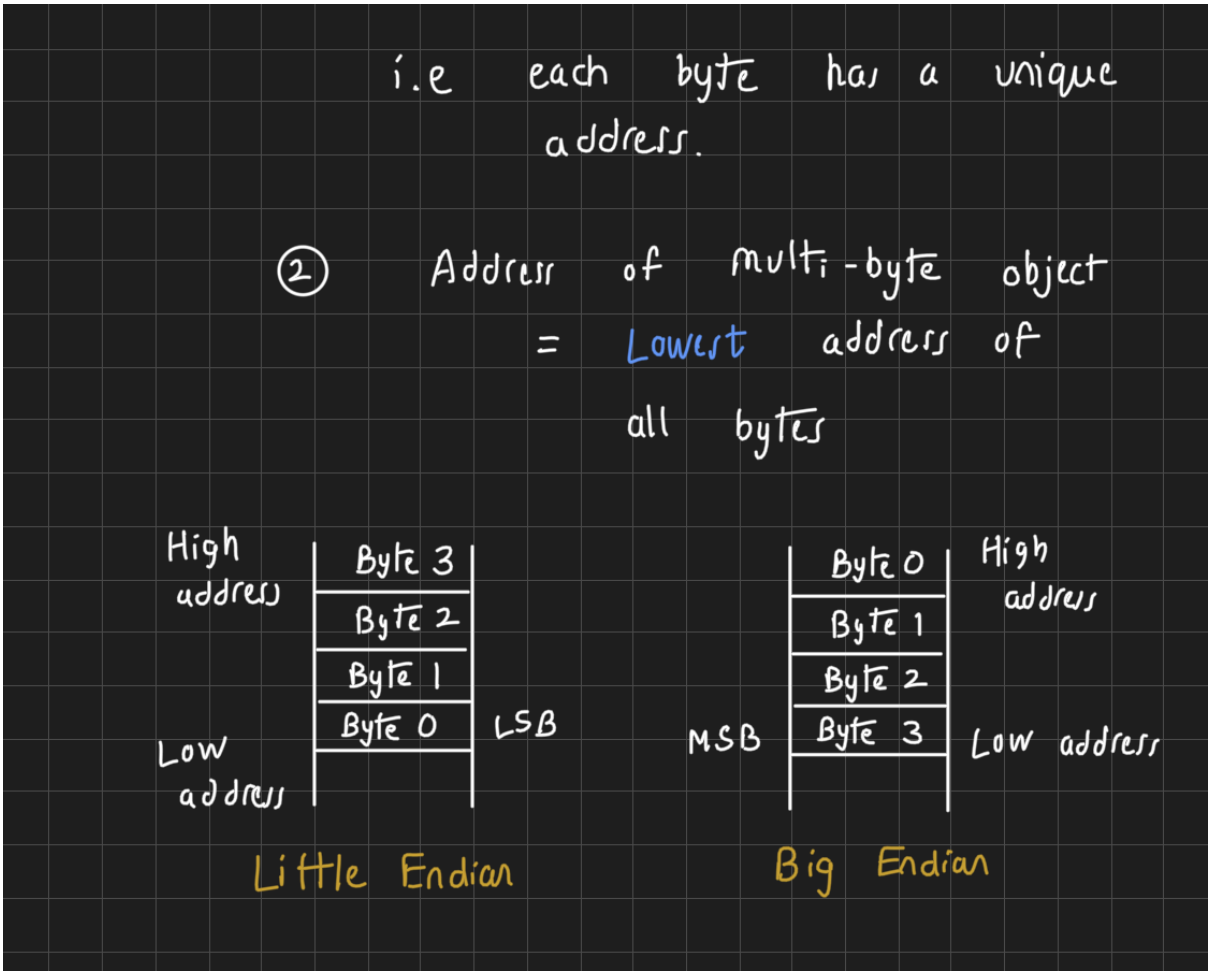
Implementations

1. Implement a web server: Build Your Own Web Server

Endianness

- endianness is the order or sequence of bytes of a word of digital data in computer memory.





Deadlock

Necessary conditions for deadlock:
i.e. a deadlock can arise if there exists:

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

Low Level Design

Step by Step designing details with UML diagrams:
https://github.com/tssovi/grokking-the-object-oriented-design-interview/blob/master/object-oriented-design-case-studies/design-a-movie-ticket-booking-system.md#code