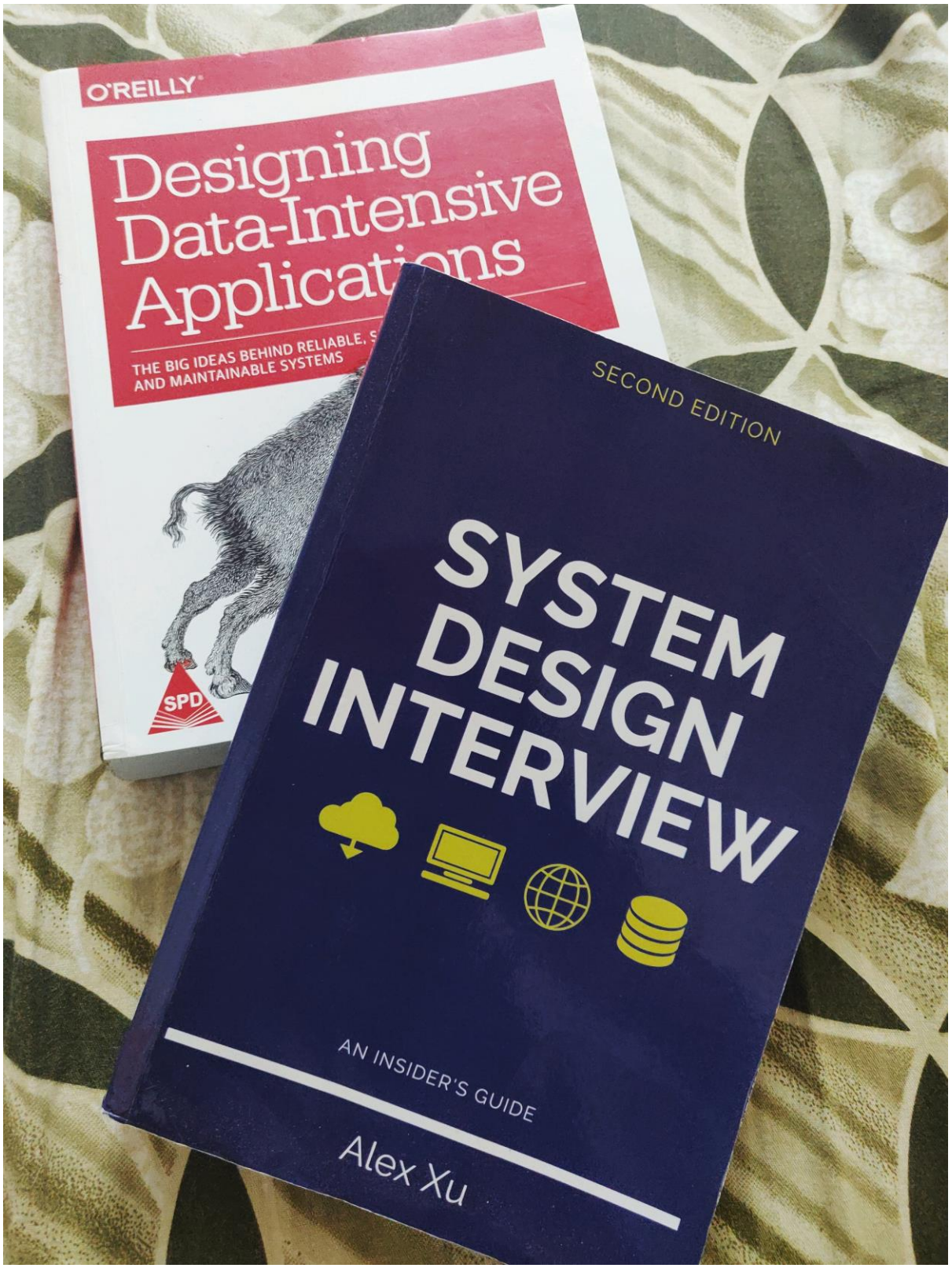

O'REILLY
Designing Data-Intensive Applications
THE BIG IDEAS BEHIND RELIABLE,
AND MAINTAINABLE SYSTEMS
SPD
SECOND EDITION
SYSTEM DESIGN INTERVIEW
AN INSIDER'S GUIDE
Alex Xu

Tips for System Design Interview

Friday, 31 December 2021 3:09 PM

Start with the Expectations, i.e. functional, nonfunctional requirements

1. Don't go into details prematurely
   - But, when it comes to DB design, ask if they want you to draw ER diagram right then (ideally, we also want to do that)

2. Don't have a said Architecture 100%
   i.e. not caring about requirements and designing as you've already read the question.

3. Keep it Simple, Stupid (KISS)

4. Don't make statements without JUSTification.
   - ex:- Saying you'll use a NoSQL DB without any relevant statements to back the design.

5. Be aware of current technologies
   - ex:- Say you'll use a CDN, it might be nice to mention a real CDN Tech. like Akamai (or) cloud front.

   (or)

   For a message queue, say you'll use maybe Rabbit MQ / Kafka etc..

⑥ Don't do capacity estimation without any reason

↳ Maybe speak out with interviewer when you notice a particular service needs to know the capacity when you're 100% sure.

⑦ Mention n/w protocols b/w communication of the systems in architecture !!! And mention pros & cons of using one over other (ex: HTTP vs FTP etc...)

⑧ Speak out internals of system (ex:- why a NOSQL DB like cassandra over MySQL. what does Cassandra bring to the architecture other DB's cannot)

⑨ TRADE OFFS !!!

Terms & Analogy

Saturday, 25 December 2021 3:07 PM

1. Scaling → Horizontal / Vertical
2. Load Balancing
3. Backups (Avoid Single Point of failure)
4. Microservices Architecture (Isolate Responsibilities)
5. Distributed System (Partitioning)
6. Decoupling
7. Logging & Metrics
8. Caching
9. API (Application programmable interface)
10. CAP Theorem (Consistency, Availability, Partition Tolerance)
11. CDN (Content Delivery network)
12. DNS (Domain name system)
13. Ports (ex. HTTP - 80, HTTPS - 443)
14. Network Protocols (IP, HTTP, TCP)

Snippets

Friday, 14 January 2022 5:23 PM

Read heavy ?

↳ Throw a cache

write heavy ?

↳ Push write reqs to a Message Queue.

Search Data ?

↳ put a search index

More users ? Need to scale Database?

↳ need ACID properties ?

↳ Shard the RDBMS Database

↳ ACID properties not mandatory ?

↳ use NOSQL Database

High throughput ?

↳ scale application servers. & put load balancer with Consistent hashing

Network Protocols

Wednesday, 12 January 2022 10:39 PM

↳ Just a set of rules to interact b/w 2 things.

↳ IP
↳ TCP
↳ HTTP

IP (Internet Protocol)

↳ 2^{16} bytes $\approx$ 0.06 MB

2 Main things in IP packet are

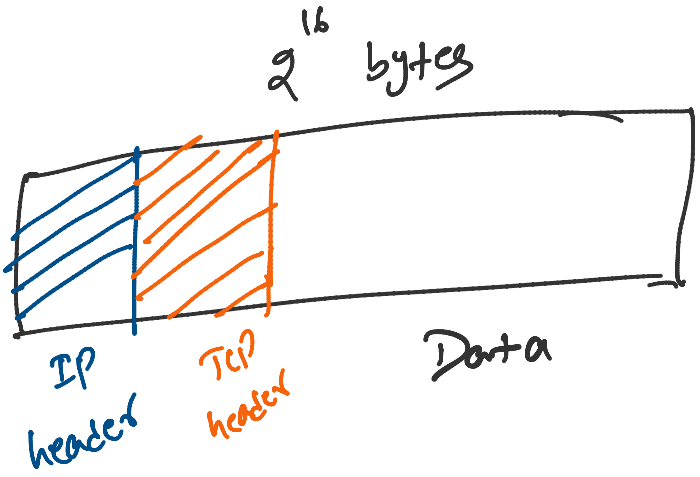
① Header (src, target, total size of packet, IP version (ex.. IPv4/IPv6)
↓
very small in general

② Data

TCP (Transmission Control Protocol)

↳ Built on top of Internet Protocol

-> Sends IP packets in order, reliable, error free way



HTTP (HyperText Transfer Protocol)

HTTP (Hyper Text Transfer Protocol)

↳ Built on top of TCP

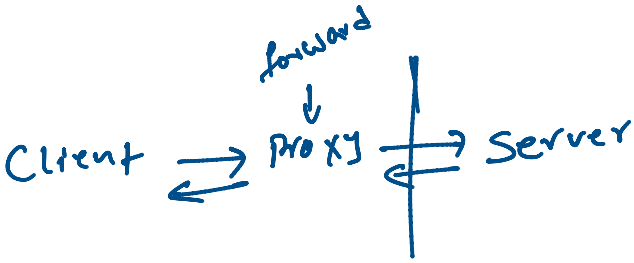
↳ This abstraction is called request-response paradigm.

Proxy (on Behalf)

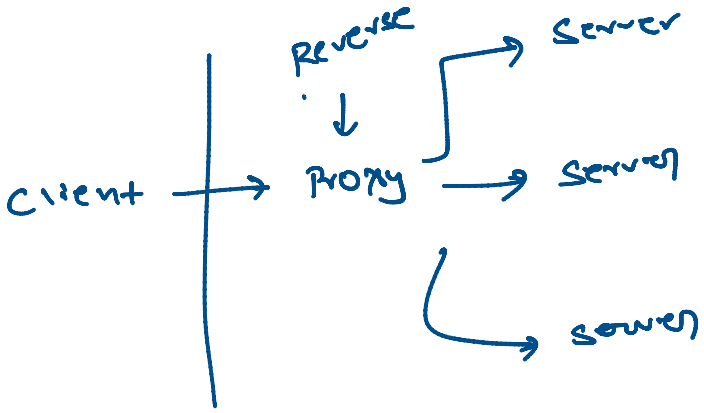
Friday, 31 December 2021 8:07 PM

Forward Proxy → Server doesn't know which client it is connecting from (VPN)

Reverse Proxy → Client doesn't know which server it is connecting to (balancer)



Forward Proxy :- Proxy sits b/w client & server and disguises client IP address, block malicious traffic from clients etc.



Reverse Proxy :- It's basically a server side proxy.

↳ can be used for traffic control, Load Balancing, caching response from servers etc...

CAP Theorem

Sunday, 2 January 2022 12:04 AM

Any N/w shared system (distributed system) cannot have all 3 properties (nearly impossible). You need to forfeit one of them

↳ consistency
↳ Availability
↳ Partition Tolerance

Consistency :- A read happening after a write should always return the latest value of the recently done write operation.

Availability :- Every available node in system should respond in a non error format to any read request, without the guarantee of returning the latest write.

Partition Tolerance:- System will be responding to all read & write requests even if the communication channel (or middleware) b/w nodes is broken (or partitioned)

-> Partition Tolerance mostly happens through N/w failures. In a distributed system, this is almost unavoidable. (so, we cannot give up on this property)

-> we can design a system with either

① Partition Tolerance + Availability (AP)

(or)

② Partition Tolerance + consistency (CP)

Degrees of consistency & Availability

-> So, we know that we cannot provide 100% consistency and 100% availability with partition tolerance in picture.

-> But, we CAN provide some extent of consistency, and availability!
(or)
degree

↳ This absolutely depends on usecase if we can achieve it or not!!

Eg:) In ATM machine, no way we can be inconsistent, it always has to be 100% consistent right. (or) Booking Tickets System.

Eg:- 1) In ATM machine, no way we can [illegible]
have to be 100% consistent right? (or) Booking Tickets System.

2) Let's say, this time we're talking about No. of views for a youtube video. Here, you can forfeit the 100% consistency (The nodes will eventually reconcile & update anyways).

API (It's only a contract !!!)

Tuesday, 28 December 2021 7:44 PM

↳ A documented way in which external consumers know how to interact with your code (They don't know how code works, they only know how to interact with it).

Things to Remember when Designing API

1. Naming is important
2. Don't take additional parameters unless absolutely necessary. (Cost to optimise the call.)
3. Don't give more info. than caller needs
4. Error handling → Response codes (HTTP) → Respond with correct HTTP codes.
   (Don't over do it) Response messages → Don't give out sensitive info here!!!

Ex:- HTTP GET ← HTTP method

www.example.com / service / method

- where to? i.e. Route.
- Typically a micro service
- The functionality etc...

Content Delivery n/w

Wednesday, 29 December 2021 8:47 PM

↳ example Akamai, S3

CDN does: (primarily focuses insanely fast data delivery by having multiple data centers geographically distributed !!!)

1. Host the data close to users
2. Follow Regulations
3. Provides some interface to update/remove? data from the data centers ?
   ↳ The CDN basically ?
4. can configure how much time to store some data in this CDN.

Cloudfront

↳ super cheap

↳ Reliable & easy to use

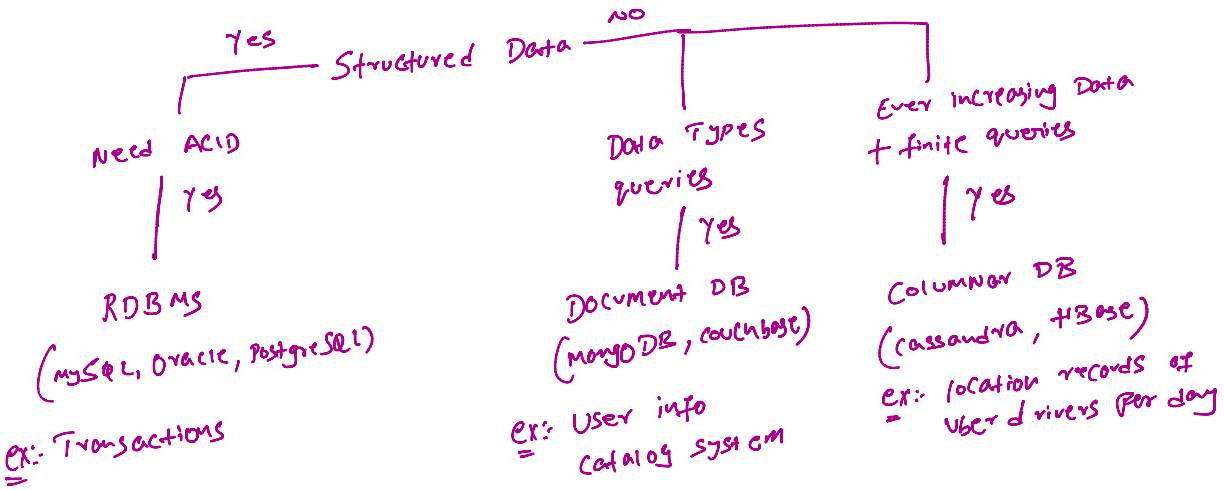
SQL vs NoSQL

Monday, 24 January 2022 1:35 AM

① Always think of caching to reduce latency.
- ↳ the server API call cache
- ↳ *Redis, MemCached (Key value stores) → non relational DBs

② Storing Images, Videos? These are blob storages & not queried upon.
- ↳ So use Amazon S3 to store
- ↳ use a CDN infront of S3 to distribute same data geographically! (faster way)

③ Text searching Feature (ex: Amazon product entire search / Netflix show title search etc..)
- ↳ ElasticSearch / Solr



-> Feel free to use a combination of database types based on the scenario!!!

① Relational Databases
- ↳ Tables & rows with a fixed schema
- ↳ ACID properties
- ↳ usecase → Entirely structured data / need acid properties
  - ex: Employee Table, Transactions etc..
- ↳ vertical scaling is easy (add more RAM, CPU etc)
- ↳ horizontal scaling is tough here (i.e. sharding)

Databases (Horizontal Partitioning inbuilt mostly)
↳ Built for scale

② Non relational Databases (Horizontal Partitioning inbuilt mostly)

↳ Key-value store (Redis, Memcache)
↳ fast & provide quick access & they are in memory
↳ used in caching solutions
↳ Built for scale
↳ Built for aggregations

↳ Document Based (mongoDB)
↳ used when not sure about structure of data & would evolve over time (i.e. No fixed schema, and can be changed easily)
↳ supports heavy reads & writes, provides sharding capabilities
↳ They have collections (like Tables) and documents (like rows)
↳ usecase → Product details of Amazon (since we want all data of one Product at one place, no joins etc..)

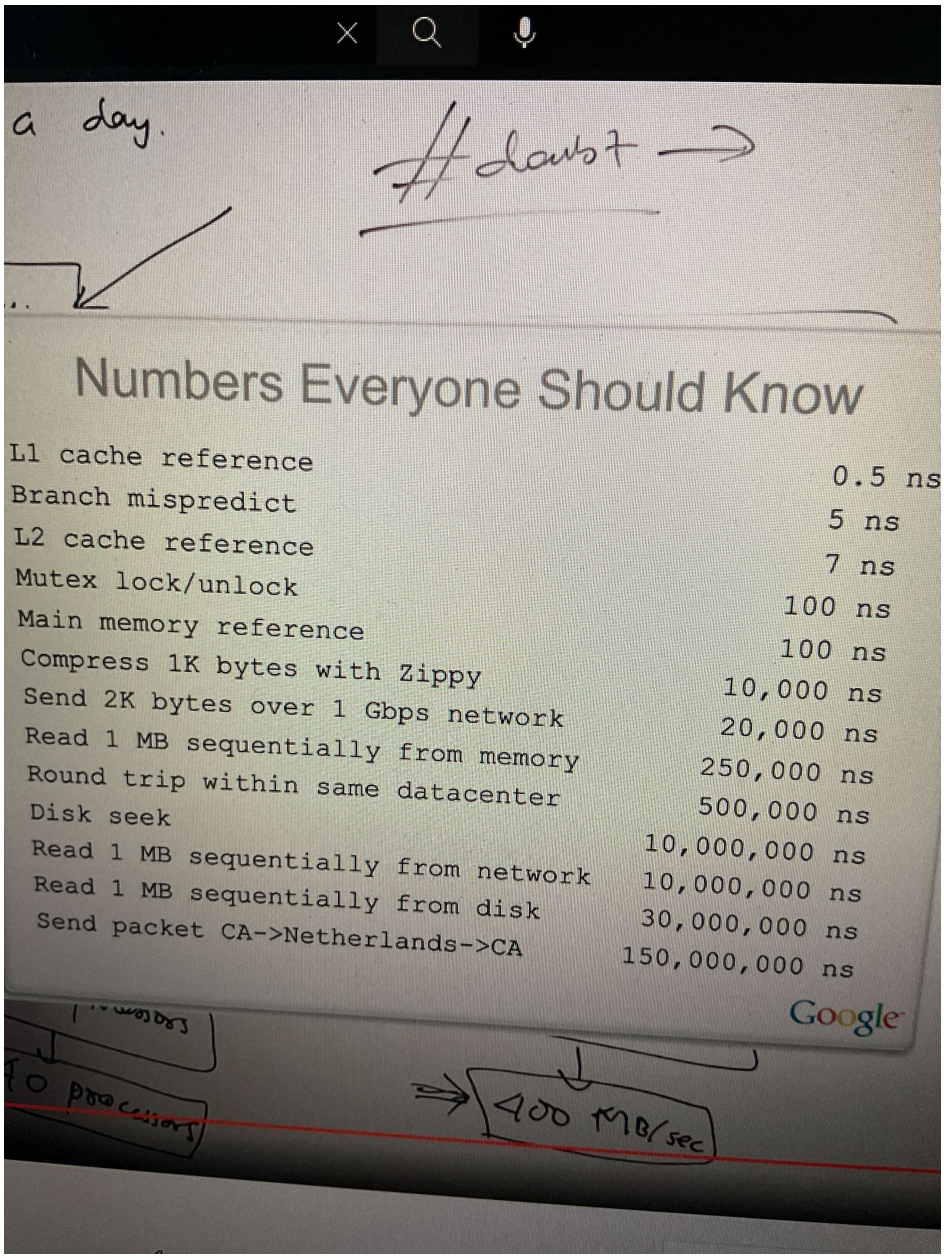
↳ Cons
↳ NO ACID Properties though we can handle it through application code, main loss is strong consistency.
↳ Joins are tough to achieve.

↳ Column DB (Cassandra, HBase, Scylla)
↳ Sort of mid way of relational & Document DBs
↳ Doesn't support ACID
↳ usecase → Streaming data / event data for analytics.
↳ Distributed databases in general

↳ Search Databases (Elasticsearch, Solr)
↳ Full text Search
↳ Fuzzy Search
↳ Keep in Mind, this is NOT PRIMARY DATABASE!

For Reference

Tuesday, 28 December 2021 9:46 PM



How to avoid Cascading failures

Friday, 31 December 2021 1:47 PM

Problems

1. Cascading failures → Rate limiting
2. Going viral → Pre scale
3. Predictable load increase → Auto scale
4. Bulk job scheduling → Batch processing
5. Popular posts → Approximate statistics.

Database Sharding

Sunday, 26 December 2021 3:59 PM

Problem :- How to query huge data in DB?

① Optimise queries (Got lot of data, even with optimised queries, it's not good enough)

② Index on table (fine, but got lots of data!!!)

③ Sharding

-> Horizontal Partitioning

Database servers!!

Takes a key and partitions the data to different servers.
(This key is generally an attribute of data we've storing)

-> Practice of seperating one table's rows into multiple different tables, known as Partitioning → Sharding

Problems

① Joins across multiple shards.

② Fixed no. of shards

↳ use Hierarchial Sharding

i.e. Let's say even after sharding, the data in one shard is huge.

in this shard we can again ... to multiple

one shard

→ Now, in this shard we can again shard / partition the big shard into multiple shards.

② what happens if a shard goes down

↳ use master / slave architecture

↳ master is the most updated one & slaves keep reading from master for updates

↳ All writes go to master, Reads are distributed to slaves

↳ If master fails, one slave becomes master.

Distributed Caching

Tuesday, 28 December 2021 6:22 PM

-> Querying DB for commonly used data?
↳ use Cache! (In Memory Cache)

Helps in saving N/w calls

-> Avoid computations again & again
↳ Instead of computing it every time from querying DB, query once & store in Cache.

-> This also avoids load on DB.

Caches generally run on SSD (Solid State Drive) which are expensive!!!

① When to load Data into cache
② When to evict / remove data from Cache.
} Cache policy!

-> Cache policy determines how cache performance is.
↳ ① LRU (Least Recently used) → Mostly used!!!
② LFU (least frequently used) → not much used

Thrashing
↳ constantly loading & removing data from cache without ever using the results!

Problem

-> How can data be consistent in caches?

i.e. server 2 updates info in DB, which is cached in server 1 cache. now what???

① write Back Cache

↳ when updating data, hit cache (if this data exists, then update it here in cache, then update DB)

↓

This method will have some data inconsistency if multiple caches are there

? This update can be delayed if data updated in cache is not of high importance.

-> Then all updates on cache can at once be sent as a batch to DB for update.

② write Through Cache

↳ First hit the DB & then update in cache. (Synchronous)

↳ Reliable & better (But slow as update needs to happen in both places)

Consistent Hashing

Saturday, 25 December 2021 3:09 PM

Load Balancing

↳ let's say we have "N" servers.
and we need to balance load properly across those "N" servers. i.e. load Balancing.

→ consistent Hashing let's us do load Balancing efficiently.

Example: h(req id) / no. of servers (General Old fashioned hashing) Doesn't work.

↓

hash function is uniformly Random
i.e. takes request id & gives out some random no./string/something

AND every server will have uniform load.

X Requests → Load = X/N → load factor = 1/N
↓
on each server

Problem?

After adding new servers (scaled horizontally), the requests are routed again

→ i.e. In real world, a request id is generally uniform. And, when we hash this, we always get same output.

→ If this req is previously sent to server "A" because of addition of new servers, same req. would go to some other server "M".
(And this is a problem with simple hashing)

Why? → BCZ, there could be cached data we want to reuse.
And, we want to send a same req. with same id to same server.

Solution (use consistent Hashing)

Total (M)
[0, M-1] range

Solution (use consistent Hashing)

① Let's say our hash function outputs in $[0, M-1]$ range (Total)

② Servers themselves have Id's also, we hash these id's also and take remainder with M.

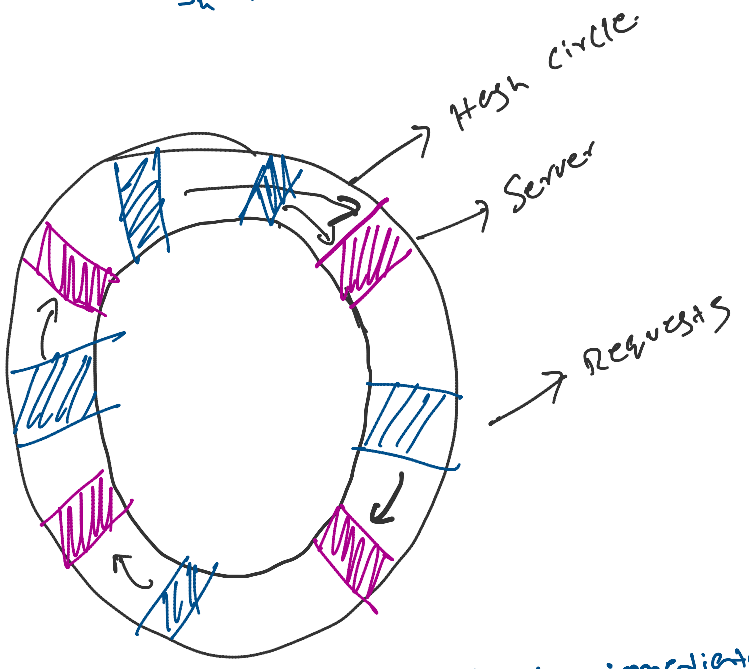
↳ i.e. $S_0 \rightarrow h(0) \% M$

$S_1 \rightarrow h(1) \% M$

$S_2 \rightarrow h(2) \% M$

$S_3 \rightarrow h(3) \% M$

$S_4 \rightarrow h(4) \% M$



- The request is served by the immediate server on the clockwise direction.
- If a newServer is added, it goes into the hash ring.
- If a server goes down, all requests that would go to this server will now be served by the next server on clockwise direction.

Problem

- If there are 5 requests that have Server "A" on clockwise direction. If this server "A" goes down,

⇒ If there ...
on clockwise direction. If this server "A" goes down
then all these 5 requests go to next server on
clockwise direction, let's say server "B".
⇒ Due to this server "B" load increased a lot!!!

Solution:-

use multiple hash functions for servers

i.e. $S_0 \rightarrow h_1(0)$ $h_2(0) \rightarrow ..$

$S_1 \rightarrow h_1(1)$ $h_2(0) \rightarrow ..$

⇒ If we have "K" hash functions, we can have K points for each server.

⇒ Likelihood of one server getting lot of load is now very less

⇒ If a server is removed, then remove the K points of that server.

Message Queue

Sunday, 26 December 2021 3:34 PM

> Takes tasks, persists the tasks, assign tasks to right servers, waits for task completion, if taking long time/server dead assigns to another server

All of these are done By a "Message Queue"

Ex:- RabbitMQ, JMS (Java Messaging Service)

Horizontal vs Vertical Scaling

Saturday, 25 December 2021 2:46 PM

→ Host services on cloud (AWS/GCP/Azure)

↓

Can be your product/service/anything.

Problem → How to handle more requests

i.e. Scalability

1. Buy bigger machine → vertical scaling
2. Buy more machines → Horizontal scaling

Trade offs

Horizontal	vertical
1. need load balancing	1. N/A
2. Resilient	2. Single point failure
3. n/w calls (RPC) between servers	3. Inter process communication
4. Data consistency issue	4. consistent
5. Scales well	5. Hardware limit

Solution

Horizontal scaling but each server is powerful enough