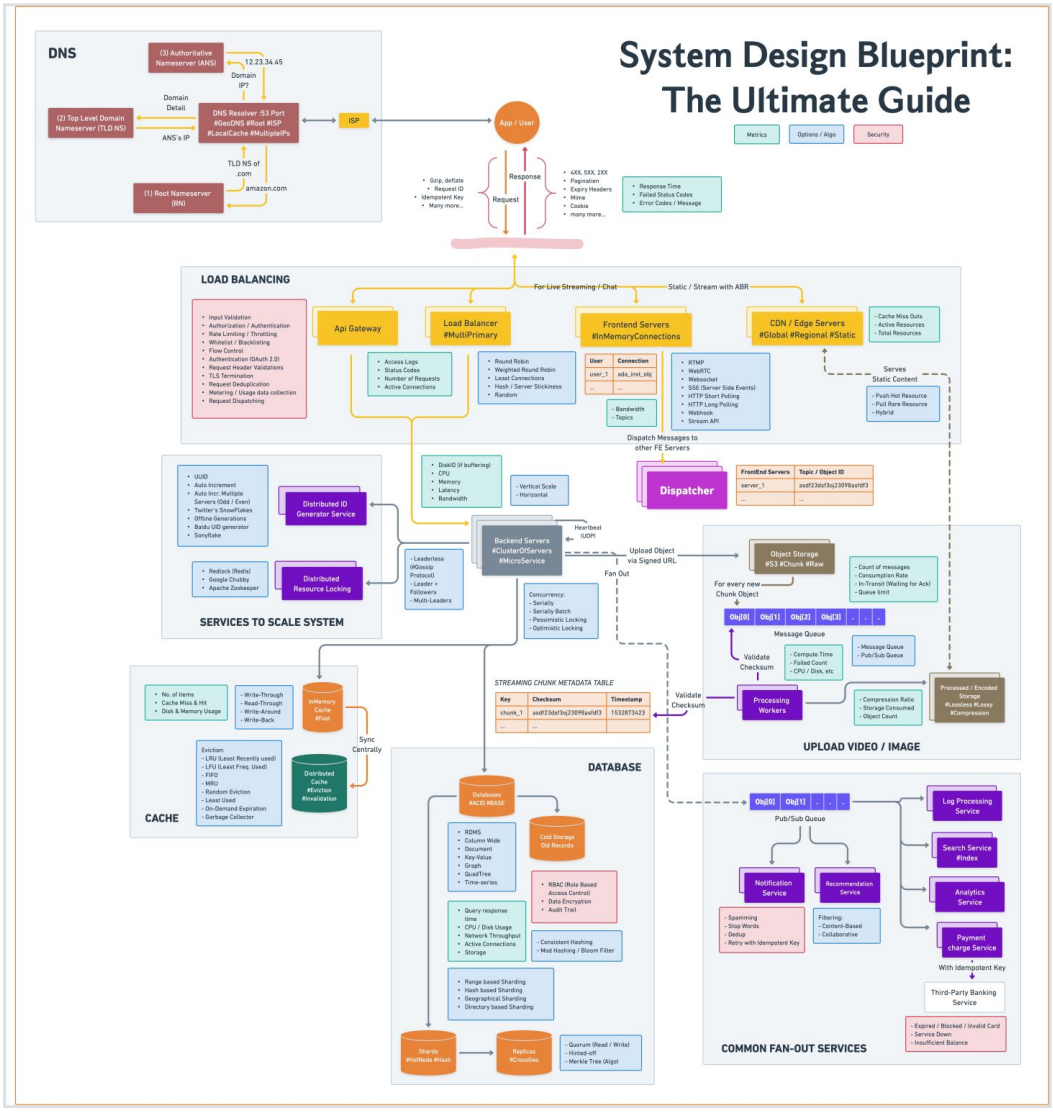
SYSTEM DESIGN

#largeScale

https://github.com/ashishps1/awesome-system-design-resources



Important Links to go through —>

https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-sticky-sessions.html

https://iorilan.medium.com/i-asked-this-system-design-question-to-3-guys-during-a-developer-interview-and-none-of-them-gave-9c23abe45687

1. For a Read-Heavy System - Consider using a Cache.
2. For a Write-Heavy System - Use Message Queues for async processing
3. For a Low Latency Requirement - Consider using a Cache and CDN.
4. Need **A**tomicity, **C**onsistency, **I**solation, **D**urability Compliant DB - Go for RDBMS/SQL Database.
5. Have unstructured data - Go for NoSQL Database.
6. Have Complex Data (Videos, Images, Files) - Go for Blob/Object storage.
7. Complex Pre-computation - Use Message Queue & Cache.
8. High-Volume Data Search - Consider search index, tries or search engine.
9. Scaling SQL Database - Implement Database Sharding.
10. High Availability, Performance, & Throughput - Use a Load Balancer.
11. Global Data Delivery - Consider using a CDN.
12. Graph Data (data with nodes, edges, and relationships) - Utilize Graph Database.
13. Scaling Various Components - Implement Horizontal Scaling.
14. High-Performing Database Queries - Use Database Indexes.
15. Bulk Job Processing - Consider Batch Processing & Message Queues.
16. Server Load Management & Preventing DOS Attacks- Use a Rate Limiter.
17. Microservices Architecture - Use an API Gateway.
18. For Single Point of Failure - Implement Redundancy.
19. For Fault-Tolerance and Durability - Implement Data Replication.
20. For User-to-User fast communication - Use Websockets.
21. Failure Detection in Distributed Systems - Implement a Heartbeat.
22. Data Integrity - Use Checksum Algorithm.
23. Efficient Server Scaling - Implement Consistent Hashing.
24. Decentralized Data Transfer - Consider Gossip Protocol.
25. Location-Based Functionality - Use Quadtree, Geohash, etc.
26. Avoid Specific Technology Names - Use generic terms.
27. High Availability and Consistency Trade-Off - Eventual Consistency.
28. For IP resolution & Domain Name Query - Mention DNS (Domain Name System).
29. Handling Large Data in Network Requests - Implement Pagination.
30. Cache Eviction Policy - Preferred is LRU (Least Recently Used) Cache.

- [x] 1. If we are dealing with a read-heavy system, it's good to consider using a Cache.
- [ ] 2. If we need low latency in the system, it's good to consider using a Cache & CDN.
- [ ] 3. If we are dealing with a write-heavy system, it's good to use a

- [ ] Message Queue for Async processing
- [ ] 4. If we need a system to be ACID complaint, we should go for RDBMS or SQL Database
- [ ] 5. If data is unstructured & doesn't require ACID properties, we should go for NO-SQL Database
- [ ] 6. If the system has complex data in the form of videos, images, files etc, we should go for Blob/Object storage
- [ ] 7. If the system requires complex pre-computation like a news feed, we should use a Message Queue & Cache
- [ ] 8. If the system requires searching data in high volume, we should consider using a search index, tries or a search engine like Elasticsearch
- [ ] 9. If the system requires to Scale SQL Database, we should consider using Database Sharding
- [ ] 10. If the system requires High Availability, Performance, & Throughput, we should consider using a Load Balancer
- [ ] 11. If the system requires faster data delivery globally, reliability, high availability, & performance, we should consider using a CDN
- [ ] 12. If the system has data with nodes, edges, and relationships like friend lists, & road connections, we should consider using a Graph Database
- [ ] 13. If the system needs scaling of various components like servers, databases, etc, we should consider using Horizontal Scaling
- [ ] 14. If the system requires high-performing database queries, we should use Database Indexes
- [ ] 15. If the system requires bulk job processing, we should consider using Batch Processing & Message Queues
- [ ] 16. If the system requires reducing server load and preventing DOS

attacks, we should use a Rate Limiter

- 17. If the system has micro-services, we should consider using an API Gateway (Authentication, SSL Termination, Routing etc)
- 18. If the system has a single point of failure, we should implement Redundancy in that component
- 19. If the system needs to be fault-tolerant, & durable, we should implement Data Replication (creating multiple copies of data on different servers)
- 20. If the system needs user-to-user communication (bi-directional) in a fast way, we should use Websockets
- 21. If the system needs the ability to detect failures in a distributed system, we should implement a Heartbeat
- 22. If the system needs to ensure data integrity, we should use Checksum Algorithm
- 23. If the system needs to transfer data between various servers in a decentralised way, we should go for the Gossip Protocol
- 24. If the system needs to scale servers with add/removal of nodes efficiently, with no hotspots, we should implement Consistent Hashing
- 25. If the system needs anything to deal with a location like maps, nearby resources, we should consider using Quadtree, Geohash, etc

1. For searching some text in millions of record: Use inverted index

2. For ACID property: use RDBMS.

3. For Unstructured data: use NoSQL

4. Database Scaling: Horizontal (Preferred for NoSQL) or Vertical Sharding(Preferred for RDBMS)

5. Read Heavy System: use Read through Cache

6. Low Latency Requirement: CDN + Load Balancer + Cache

7. Write-Heavy System: Use ASYNC Processing (kafka message queues)

8. Handle Complex Data like Videos, Images, Files etc: Go for Object storage (ex: Amazon S3)

9. For High Availability, Performance, & Throughput: Use Load Balancer

10. Global Data Delivery: Use CDN.

11. For Fast DB Queries: Database Indexing

12. Load Management on a Component: Use Rate Limiter.

13. Avoid Single Point of Failure: Set up Disaster Recovery Data Centre

14. Fault-Tolerance: Write though cache + Master Slave Architecture.

15. Peer to Peer communication: Use WebSockets.

16. For VideoCall : Use WebRTC

17. For Data Integrity between 2 system: Use Checksum Algorithm.

18. Efficient Managing of the Servers: Use Consistent Hashing.

19. High Availability and Consistency Trade-Off: Eventual Consistency.

20. Cache Eviction Policy: LRU cache (generally preferred, but there are more)

NOSQL vs MYSQL

NoSQL -

1. Can be sharded easily
2. Have its own gossip protocol
3. Have some kind of Qorum
4. Have some kind of consistency and availability trade off over sql particular kinds of data it will take very quickly

So use cases for NoSQL come in the form of things like social networks, video streaming and their comments stuff like that. For things that need transaction consistency like commerce you're better off with a SQL db.

Notes to self:
* Sharding is basically a hiearchical way to index databases.
* One problem is that you have to split the database somehow. What do you

split on?
* You only shard shards when the shard grow too big.
* When shard fails you use the master/slave architecture. Writes always go to master, reads are distributed across the slaves. When the master fails one of the slaves become master.

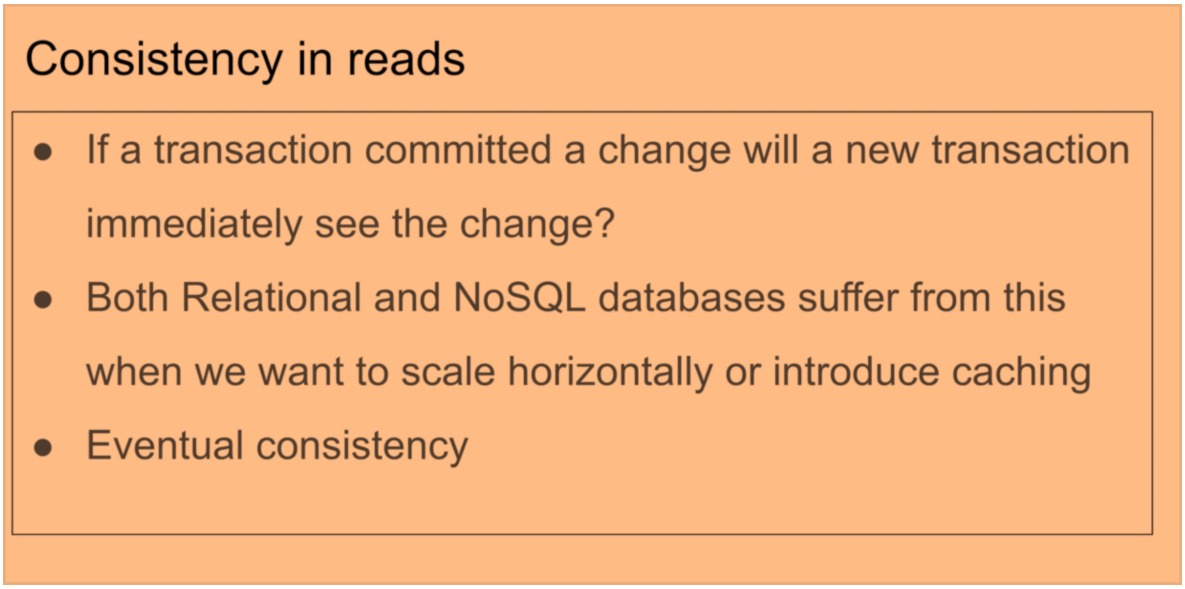
HOW TO TACKLE SYSTEM DESIGN INTERVIEWS?

EXTREMELY IMPORTANT:

- Bandwidth requirement
- Storage estimations
- Trade off

CONSISTENCY VS EVENTUAL CONSISTENCY

- CONSISTENCY : Lets take an example to understand consider a situation you go to bank and deposit the money the updated money should reflect in your account that is consistency which generally bank requires it.
- EVENTUAL CONSISTENCY : Let take an example to understand that consider a situation where Virat Kohli has posted a picture and got 100k likes but the actual was 10001031 something here in this case we can follow eventual consistent.



What happens when we index the table (behind the

scenes)

Consider a situation where table is not indexed

If tables as 1000 rows so what mysql does it store the data of 1-3 rows in page 1
3-6 in page 2 and so on ....

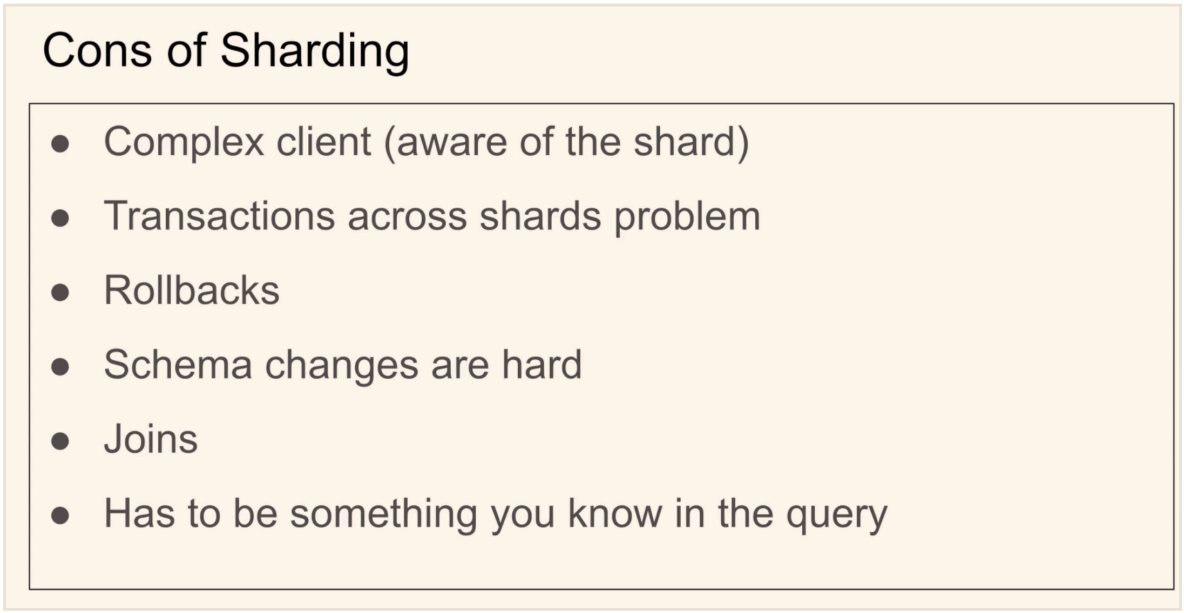
So now 1000/3 ~ 333 pages

Let say now we query like "SELECT * FROM TBL_EMPLOYEE where id = 10" now what it does it goes to every page and check if there is any where id = 10 this will take worst case complexity as 333 pages and it will explore every page till it find any of them

Now lets say we index the table so the data will be stored in the form of B-Tree format

So as B-Tree will take smaller time as compare the to the non-index table

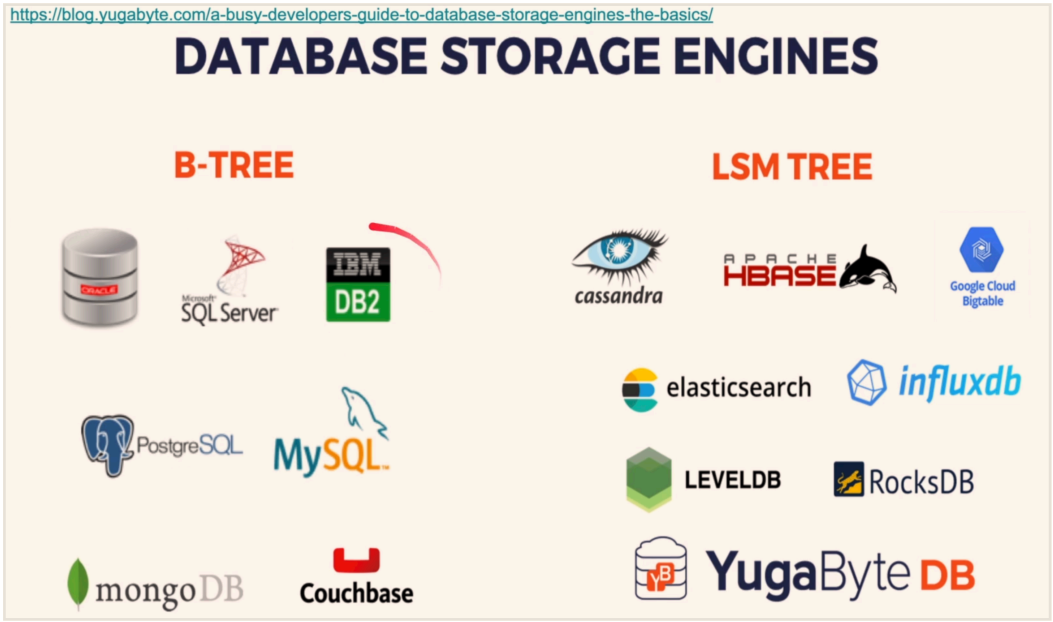
Don't put too many index in a table the greater the index in the table the greater willl be the time for a query to fetch



. When the number of tids are less then its fast to do index scan (random disk access).
2. When tids are moderate its fast to do bitmap scan since bitmap scan is more sequential as compared to random access incase of index scan.
3. If tids are very high its better to do sequential scan, as sequential scan on disk is faster then random disk access based on tids is what I understood (severalnines.com overview various scan methods postgresql)

Notes

- Sometimes the heap table can be organized around a single index. This is called a clustered index or an Index Organized Table.
- Primary key is usually a clustered index unless otherwise specified.
- MySQL InnoDB always have a primary key (clustered index) other indexes point to the primary key “value”
- Postgres only have secondary indexes and all indexes point directly to the row_id which lives in the heap.



https://iorilan.medium.com/a-5-years-tech-lead-said-they-shard-a-database-to-scale-but-then-he-failed-to-answer-this-question-8be39115dcb0

To this topic, there are much more difficult questions ahead in my mind:

- When you add a new machine, do you need re-shard? and how did you handle it?
- How do you handle schema changes?
- How do you join tables? Does your code have no sub-query? and what if you have to use it? Is there an alternative approach?
- How do you make sure data integrity? after you did sharding, you threw away the entire ACID (primary key only applies to local shard table) which came from RDBMS: data loss and inconsistency now become possible.
- And all the aggregating functions like count, sum, and max

become challenging, there must be a high-performance and mature sharding proxy or “merging engine” like vitess or shardingsphere in the middle. and you need to master this engine to save your ass.

- How is the transaction handling different from a single database?
- ...

Never talk about something you are not sure about during an interview. 🙂

Today let’s spend a few minutes talking about scaling the database.
First thing, you will not want to go sharding in the first place.
Unless you tried all the possible options, otherwise, never shard your database (even for NoSQL which naturally support distributed system, do not shard until you have to).
Why?
It makes things super complicated.
So let’s follow this interviewee’s problem, he has a database table, that keeps growing, and makes queries slow, how to solve it?

First, split tables

Yes, sharding, but vertically.
To mitigate the table growing issue, we need to split the columns into 2 categories:

- Frequently changes (red)
- No or fewer changes (blue)

So that we can eliminate tracking those unchanged rows.

❓

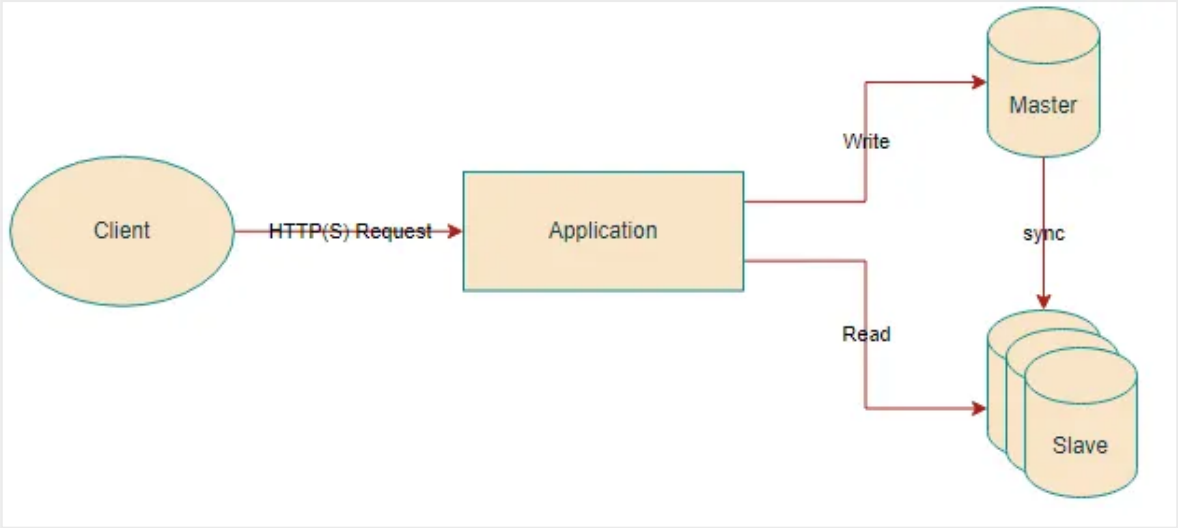
So let’s say it is an online shopping system. We have a product table with some columns rarely change after first creation, like *id, name photo*. and some columns that may need to be updated frequently like *supplier or warehouse_id and verified employee id*.
by splitting it into 2 tables. we reduced the writes into the product_basic table.
So indirectly we reduced the potential locks on the old big table, and improved the performance.
Okay you may ask “Wait. this is the problem: we have a table, that keeps growing, and using the above solution will still end up with a huge product_etx table”
Sure we can continue to try something else.

Then, try the simplest way first, to Archive

Keep the busy database small.
Archive the old records (split the cold and hot data). create an archiving database and move data over there every month or year.
Do not forget to compress the files if your database supports this feature.
So if you need to search historical data, go to this archive database, if you find the schema is not what you want, you can even create a reporting database and

flow the data there for your purpose (you can build an ETL in between if needed).
So you may ask “Wait, What if after archiving it is still slow, because we can only archive 5 years ago data”.
Sure. we try something, and no still not sharding.

Read-Write Splitting



- Master-Slave + replica
- Read goes to slaves, Write into master
- One way sync master to slaves (most database support this)

If archiving itself doesn’t solve the problem. on top of that, we split read/write.
You may want to check out CQRS + event sourcing which is a very mature proven working solution for years.
You may wonder What if this case “very high concurrency (10k+ RPS) happens, so race-conditioning the same rows causes a lot of deadlocks. it may even kill MySQL.”
Sure, you can add a cache for that high frequent accessed table data in front to protect the database.
But since you introduced cache then remember to write through or use invalidate date to control the expiration of the caching data.
And if needed, make a cache cluster as well.
“No, the cache only deal with the concurrency but didn’t do anything to the fast-growing table, archiving too much data is not an option, and read-write is not enough to get the performance.”
Well. okay. let’s try one more.

Table Partition

By time range. the data will be split into different partition physical files and all are managed by MySQL.
Yes the limitation, or the difference from sharding (range based) is, all

partitions are on the same machine.
You may ask "Wait, It didn't solve the issue, right? it is not scalable. I can not scale the solution by adding a new machine."
but you can add a new disk, right?
"In the end, the machine will be a bottleneck".
Well, true, you are right. let's continue.

Sharding, but range based

In our application logic layer, we can route the data to different databases based on logical ids.
It could be *customer_id or user_id, time range, location_id, or city_id, etc.* the load will be balanced at the application layer.
If one of the machines has a problem, only affects the particular location or city or range of customers.
This solution is scalable, balanced the load, and also splits the risk.
Okay you may ask "what if in my case, the single location or city_id data grows still very very huge? Yes, I mean the data hotspot case, so we need another solution."
Well, you should choose the sharding key very very carefully to avoid this happening...
Btw, this is the max level of complexity I could accept for a database solution. beyond which is a bit out of control.

Key-based sharding

Let's say 2 machines. the hashing function is as simple as row_id%2.
row_id%2==0 go to machine1 and another one go machine2

It only looks simple, but If possible, I never want to reach here.
The "cons" make me stop thinking about it.

Cons

- I can never use incremental Id as the primary key.
- Primary or foreign keys and unique keys do not make sense anymore. it is only applied to the shard database table. we need another solution to get this feature back.
- The hashing function needs to be chosen very very carefully. e.g. For the above sample, when I add a machine, need to change the function into row_id%3. which is very very bad — re-shard is the cost is very very high: migration is complicated, potentially may lose data, and the downtime is unknown.
- Need to rewrite many queries (into very simple ones). no subquery, no joins, and no aggregate functions can be used.
- The transaction is hard to control. distributed system transaction is already complicated, sharding makes it even harder.
- When a slow query (after a query from multiple machines then merges and sorts in memory) happens, my luck depends on

caching or the engine in the middle like vitess or shardingsphere.

- With above. horizontal sharding seems more suitable for NoSQL. but I don't try until I have to.

Pros (compared to range based)

- No data hotspot problem
- Nothing more -_-!

So, we may still want to know, "I know all the cons, what if we have to use it". so when this happens, don't build it yourself, even if you are confident to handle all the above "cons".

- Try with NoSQL first. it is naturally distributed and mostly has more built-in features to support sharding
- If you have to use SQL like MySQL. use some proven engine like vitess (used by youtube). here is an example of how slack did it.
- If doesn't work, try another engine, keep trying, and use the existing solution, never build it yourself.

Recap

when scaling a big table with growing data.
#1 Try the simplest way first
#2 It does not work then slowly adds complexity
#3 Keep balancing the benefits that the complexity brings.
#4 Never jump into any complicated solutions (even if it looks sexy).

- Scale up: Add RAM/CPU (CPU bound) or use SSD (I/O bound). whenever possible.
- Reuse existing mature proven solutions. e.g. handle growing logs searching, throwing them into ELK.
- Review table fields. categorize the fields by updating frequency. put those fields updated more frequently into one table and others into another table.
- Archive history data + Split read and write + replica + Cache (In many cases, we stopped here, all below solutions has risk)
- Table partition (by date range). — risk: be mindful of the physical file size and the usage of an information schema.
- Sharding, range based (choose the key carefully, avoid hotspot data) — risk: think carefully when choosing the key, avoid hotspot. and if this happens, changing back to key-based is almost impossible.
- Is the data structure could be fit into any NoSQL? migrate there — risk: mysqldump is not possible. select into outfile csv then import seems the only way, to benchmark more and test all languages, and special chars.
- Shard by key. check Vitess and other existing proven shard engines. setup and reuse. — risk: Sorry I don't know what could

happen to you, good luck. :)

SYNCHRONOUS VS ASCYSCHORNOUS

- SYNCHRONOUS REPLICATION

Here what happens is that the replication happens to master and then if there are 5-6 backup slaves then it client has to wait if any of the slaves have updated the data

This is how the process follows

Client -> master write -> slave -> write -> request is completed to client

- ASYCSYNCHRONOUS REPLICATION

In this case only the master write is done and client request is completed and rest replication onto the slaves happens into the background;

DATABASE CURSOR

Question for interview : Sort the database row which as billion rows

Ans : As this will take a lot of time to sort we can use database cursor fetch each row store and then sort accordingly this is a trick question into the interview

UUDI VS SEQUENTIAL KEYS

UUID - it contains 256 bit data to store into the disk(where the data is stored) so if we make this as a primary key what will happen we will have to load the data related to that into the memory to display or to fetch because the data is stored onto the disk and then when we query something with the primary key it fetches the data from the disk and load into the memory to display this loading will have a huge amount of byte to load into the memory which can be slow.

UUID is totally random string and database don't like this stuff consider that we query with 1 UUID and database kind of cache things but UUID are so random that the ODDs of the UUID which I cached is small that it will pick it from cache and this way it willl again go the the INDEX find the location using pointer and then fetch the data from the disk and load into the memory here also we go to scan sequential and then find that UUID and the return the data.

Sequential Keys : In this case what happens when we do auto increment it is simple process but behind the scene what it does ….

Consider an employee table where there are 4-5 queries have been queried to insert into this table so

QUERY 1 will take a lock for a split second into the table while other are waiting for the lock to release so that other queries can be executed

HOW UPDATES HAPPENS IN SSD DISK ?

Whenever we try to update anything into the SSD what happens is that it pulls the page number copy it to memory change the data and the flush it to the disks this take an extra time as well and also SSD do like an update query it is meant for the insert.

DO and DON’Ts in SYSTEM DESIGN INTERVIEW

To wrap up, we summarize a list of the Dos and Don’ts.
Dos
• Always ask for clarification. Do not assume your assumption is correct. • Understand the requirements of the problem.
• There is neither the right answer nor the best answer. A solution designed to solve the problems of a young startup is different from that of an established company with millions of users. Make sure you understand the requirements.
• Let the interviewer know what you are thinking. Communicate with your interview. • Suggest multiple approaches if possible.
• Once you agree with your interviewer on the blueprint, go into details on each component. Design the most critical components first.
• Bounce ideas off the interviewer. A good interviewer works with you as a teammate. • Never give up.
Don’ts
• Don't be unprepared for typical interview questions.
• Don’t jump into a solution without clarifying the requirements and assumptions.
• Don’t go into too much detail on a single component in the beginning. Give the high- level design first then drills down.
• If you get stuck, don't hesitate to ask for hints. • Again, communicate. Don't think in silence.
• Don’t think your interview is done once you give the design. You are not done until your interviewer says you are done. Ask for feedback early and often.
Time allocation on each step
System design interview questions are usually very broad, and 45 minutes or an hour is not enough to cover the entire design. Time management is essential. How much time should you spend on each step? The following is a very rough

guide on distributing your time in a 45- minute interview session. Please remember this is a rough estimate, and the actual time distribution depends on the scope of the problem and the requirements from the interviewer.
Step 1 Understand the problem and establish design scope: 3 - 10 minutes
Step 2 Propose high-level design and get buy-in: 10 - 15 minutes
Step 3 Design deep dive: 10 - 25 minutes
Step 4 Wrap: 3 - 5 minutes

SQL vs NOSQL

Non-relational databases might be the right choice if:
• Your application requires super-low latency.
• Your data are unstructured, or you do not have any relational data.
• You only need to serialize and deserialize data (JSON, XML, YAML, etc.). • You need to store a massive amount of data.

Vertical scaling vs horizontal scaling

Vertical scaling, referred to as “scale up”, means the process of adding more power (CPU, RAM, etc.) to your servers. Horizontal scaling, referred to as “scale-out”, allows you to scale by adding more servers into your pool of resources.
When traffic is low, vertical scaling is a great option, and the simplicity of vertical scaling is its main advantage. Unfortunately, it comes with serious limitations.
• Vertical scaling has a hard limit. It is impossible to add unlimited CPU and memory to a single server.
• Vertical scaling does not have failover and redundancy. If one server goes down, the website/app goes down with it completely.
Horizontal scaling is more desirable for large scale applications due to the limitations of vertical scaling.
In the previous design, users are connected to the web server directly. Users will unable to access the website if the web server is offline. In another scenario, if many users access the web server simultaneously and it reaches the web server’s load limit, users generally experience slower response or fail to connect to the server. A load balancer is the best technique to address these problems.

Considerations for using cache

https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/

Caching is one of the easiest ways to increase system performance. Databases can be slow (yes even the NoSQL ones) and as you already know, speed is the name of the game.

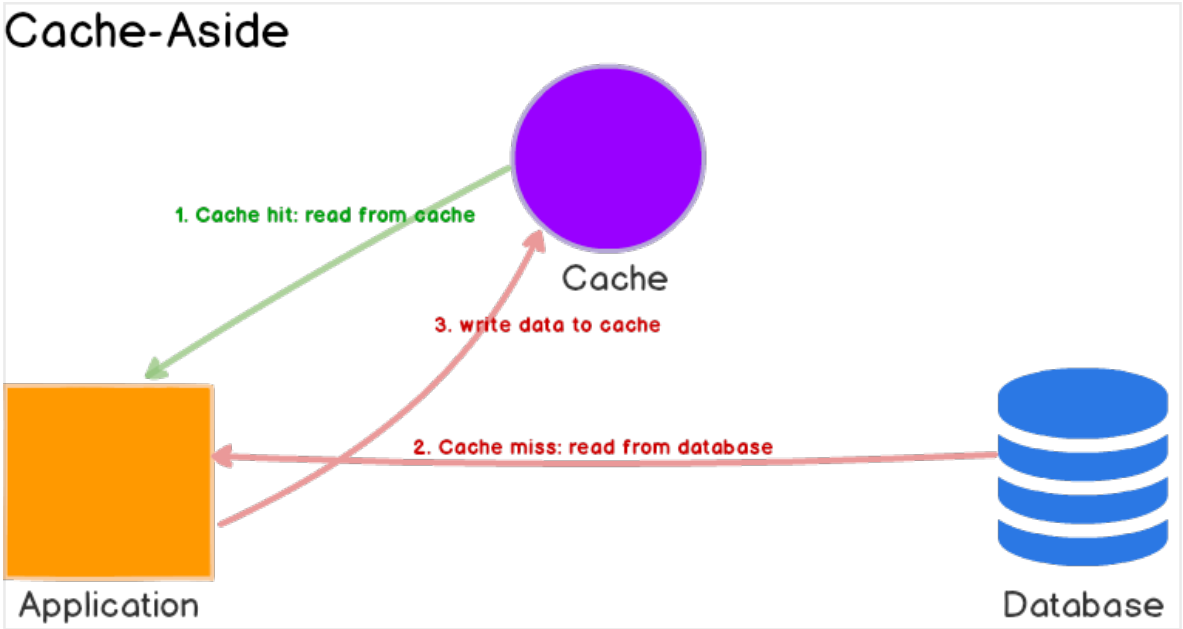
If done *right*, caches can reduce response times, decrease load on database, and save costs. There are several strategies and choosing the *right* one can make a big difference. Your caching strategy depends on the data and **data access patterns**. In other words, how the data is written and read. For example:

- is the system write heavy and reads less frequently? (e.g. time based logs)
- is data written once and read multiple times? (e.g. User Profile)
- is data returned always unique? (e.g. search queries)

A caching strategy for Top-10 leaderboard system for mobile games will be very different than a service which aggregates and returns user profiles. Choosing the right caching strategy is the key to improving performance. Let's take a quick look at various caching strategies.

Cache-Aside

This is perhaps the most commonly used caching approach, at least in the projects that I worked on. The cache sits on the *side* and the application **directly** talks to both the cache and the database. There is no connection between the cache and the primary database. All operations to cache and the database are handled by the application. This is shown in the figure below.



Here's what's happening:

1. The application first checks the cache.
2. If the data is found in cache, we've *cache hit*. The data is read and returned to the client.
3. If the data is **not found** in cache, we've *cache miss*. The application has to do some **extra work**. It queries the database to read the data,

returns it to the client and **stores** the data in cache so the subsequent reads for the same data results in a cache hit.

Use Cases, Pros and Cons

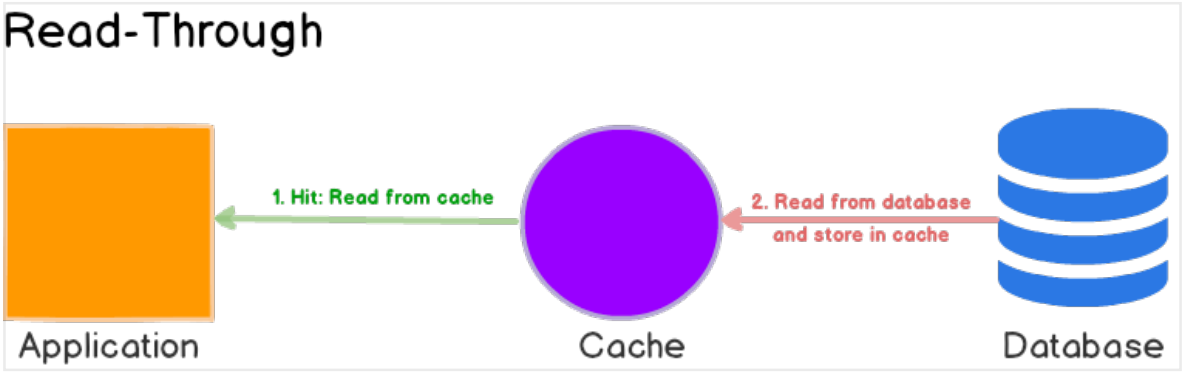
Cache-aside caches are usually general purpose and work best for **read-heavy workloads**. *Memcached* and *Redis* are widely used. Systems using cache-aside are **resilient to cache failures**. If the cache cluster goes down, the system can still operate by going directly to the database. (Although, it doesn't help much if cache goes down during peak load. Response times can become terrible and in worst case, the database can stop working.)

Another benefit is that the data model in cache can be different than the data model in database. E.g. the response generated as a result of multiple queries can be stored against some request id.

When cache-aside is used, the most common write strategy is to write data to the database directly. When this happens, cache may become inconsistent with the database. To deal with this, developers generally use time to live (TTL) and continue serving stale data until TTL expires. If data freshness must be guaranteed, developers either **invalidate the cache entry** or use an appropriate write strategy, as we'll explore later.

Read-Through Cache

Read-through cache sits in-line with the database. When there is a cache miss, it loads missing data from database, populates the cache and returns it to the application.



Both cache-aside and read-through strategies load data **lazily**, that is, only when it is first read.

Use Cases, Pros and Cons

While read-through and cache-aside are very similar, there are at least two key differences:

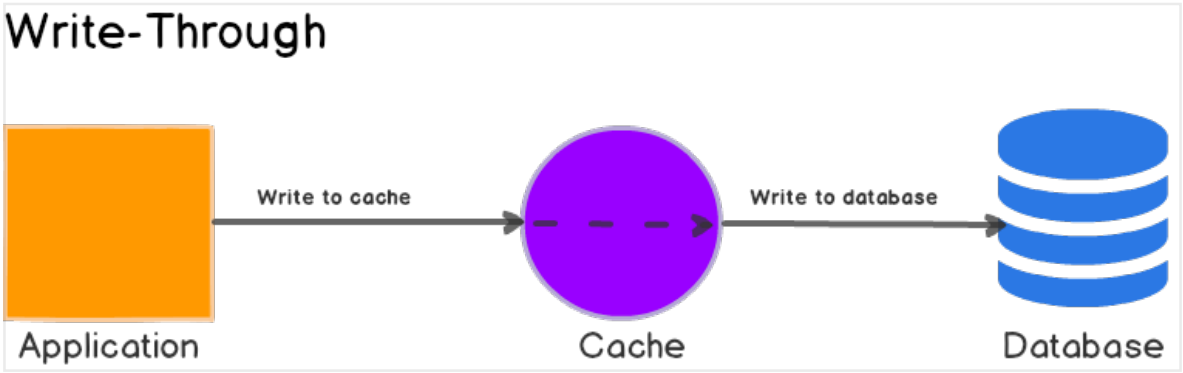
1. In cache-aside, the application is responsible for fetching data from the database and populating the cache. In read-through, this logic is usually supported by the library or stand-alone cache provider.
2. Unlike cache-aside, the data model in read-through cache cannot be different than that of the database.

Read-through caches work best for **read-heavy** workloads when the same data is requested many times. For example, a news story. The disadvantage is that when the data is requested the first time, it always results in cache miss and incurs the extra penalty of loading data to the cache. Developers deal with this

by '*warming*' or 'pre-heating' the cache by issuing queries manually. Just like cache-aside, it is also possible for data to become inconsistent between cache and the database, and solution lies in the write strategy, as we'll see next.

Write-Through Cache

In this write strategy, data is first written to the cache and then to the database. The cache sits in-line with the database and writes always go *through* the cache to the main database. This helps cache maintain consistency with the main database.



Here's what happens when an application wants to write data or update a value:

1. The application writes the data directly to the cache.
2. The cache updates the data in the main database. When the write is complete, both the cache and the database have the same value and the cache always remains consistent.

Use Cases, Pros and Cons

On its own, write-through caches don't seem to do much, in fact, they introduce extra write **latency** because data is written to the cache first and then to the main database (two write operations.) But when paired with read-through caches, we get all the benefits of read-through and we also get data **consistency** guarantee, freeing us from using cache invalidation (assuming ALL writes to the database go through the cache.)
DynamoDB Accelerator (DAX) is a good example of read-through / write-through cache. It sits inline with DynamoDB and your application. Reads and writes to DynamoDB can be done through DAX. (Side note: If you are planning to use DAX, please make sure you familiarize yourself with its data consistency model and how it interplays with DynamoDB.)

Write-Around

Here, data is written directly to the database and only the data that is read makes it way into the cache.

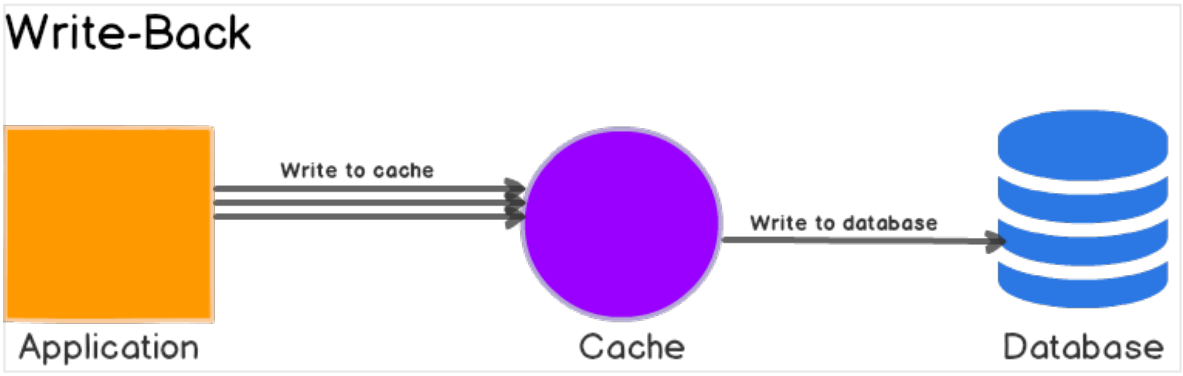
Use Cases, Pros and Cons

Write-around can be combine with read-through and provides good performance in situations where data is written once and read less frequently or never. For example, real-time logs or chatroom messages. Likewise, this pattern can be combined with cache-aside as well.

Write-Back or Write-Behind

Here, the application writes data to the cache which stores the data and acknowledges to the application *immediately*. Then later, the cache writes the

data *back* to the database.
This is very similar to to Write-Through but there's one crucial difference: In Write-Through, the data written to the cache is **synchronously** updated in the main database. In Write-Back, the data written to the cache is **asynchronously** updated in the main database. From the application perspective, writes to Write-Back caches are *faster* because only the cache needed to be updated before returning a response.



This is sometimes called write-behind as well.

Use Cases, Pros and Cons

Write back caches improve the write performance and are good for **write-heavy** workloads. When combined with read-through, it works good for mixed workloads, where the most recently updated and accessed data is always available in cache.
It's resilient to database failures and can tolerate some database downtime. If batching or coalescing is supported, it can reduce overall writes to the database, which decreases the load and **reduces costs**, if the database provider charges by number of requests e.g. DynamoDB. Keep in mind that **DAX is write-through** so you won't see any reductions in costs if your application is write heavy. (When I first heard of DAX, this was my first question - DynamoDB can be very expensive, but damn you Amazon.)
Some developers use Redis for both cache-aside and write-back to better absorb spikes during peak load. The main disadvantage is that if there's a cache failure, the data may be permanently lost.
Most relational databases storage engines (i.e. InnoDB) have write-back cache enabled by default in their internals. Queries are first written to memory and eventually flushed to the disk.

Summary

In this post, we explored different caching strategies and their pros and cons. In practice, carefully evaluate your goals, understand data access (read/write) patterns and choose the best strategy or a combination.
What happens if you choose wrong? One that doesn't match your goals or access patterns? You may introduce additional latency, or at the very least, not see the *full benefits*. For example, if you choose *write-through/read-through* when you actually should be using *write-around/read-through* (written data is accessed less frequently), you'll have useless junk in your cache. Arguably, if the cache is big enough, it may be fine. But in many real-world, high-throughput systems, when memory is never big enough and server costs

are a concern, the right strategy, matters.

Here are a few considerations for using a cache system:
• Decide when to use cache. Consider using cache when data is read frequently but modified infrequently. Since cached data is stored in volatile memory, a cache server is not ideal for persisting data. For instance, if a cache server restarts, all the data in memory is lost. Thus, important data should be saved in persistent data stores.
• Expiration policy. It is a good practice to implement an expiration policy. Once cached data is expired, it is removed from the cache. When there is no expiration policy, cached data will be stored in the memory permanently. It is advisable not to make the expiration date too short as this will cause the system to reload data from the database too frequently. Meanwhile, it is advisable not to make the expiration date too long as the data can become stale.
• Consistency: This involves keeping the data store and the cache in sync. Inconsistency can happen because data-modifying operations on the data store and cache are not in a single transaction. When scaling across multiple regions, maintaining consistency between the data store and cache is challenging. For further details, refer to the paper titled "Scaling Memcache at Facebook" published by Facebook [7].
• Mitigating failures: A single cache server represents a potential single point of failure (SPOF), defined in Wikipedia as follows: "A single point of failure (SPOF) is a part of a system that, if it fails, will stop the entire system from working" [8]. As a result, multiple cache servers across different data centers are recommended to avoid SPOF. Another recommended approach is to overprovision the required memory by certain percentages. This provides a buffer as the memory usage increases.
• Eviction Policy: Once the cache is full, any requests to add items to the cache might cause existing items to be removed. This is called cache eviction. Least-recently-used (LRU) is the most popular cache eviction policy. Other eviction policies, such as the Least Frequently Used (LFU) or First in First Out (FIFO), can be adopted to satisfy different use cases.

Considerations of using a CDN

• Cost: CDNs are run by third-party providers, and you are charged for data transfers in and out of the CDN. Caching infrequently used assets provides no significant benefits so you should consider moving them out of the CDN.
• Setting an appropriate cache expiry: For time-sensitive content, setting a cache expiry time is important. The cache expiry time should neither be too long nor too short. If it is too long, the content might no longer be fresh. If it is too short, it can cause repeat reloading of content from origin servers to the CDN.
• CDN fallback: You should consider how your website/application copes with CDN failure. If there is a temporary CDN outage, clients should be able to detect the problem and request resources from the origin.

• Invalidating files: You can remove a file from the CDN before it expires by performing one of the following operations:
• Invalidate the CDN object using APIs provided by CDN vendors.
• Use object versioning to serve a different version of the object. To version an object, you can add a parameter to the URL, such as a version number. For example, version number 2 is added to the query string: image.png?v=2.

Stateless web tier

Now it is time to consider scaling the web tier horizontally. For this, we need to move state (for instance user session data) out of the web tier. A good practice is to store session data in the persistent storage such as relational database or NoSQL. Each web server in the cluster can access state data from databases. This is called stateless web tier.

Stateful architecture

A stateful server and stateless server has some key differences. A stateful server remembers client data (state) from one request to the next. A stateless server keeps no state information.

To store data in stateful architecture there are lot of databases but The NoSQL data store is chosen as it is easy to scale.

Sharding

Sharding is a great technique to scale the database but it is far from a perfect solution. It introduces complexities and new challenges to the system:
Resharding data: Resharding data is needed when 1) a single shard could no longer hold more data due to rapid growth. 2) Certain shards might experience shard exhaustion faster than others due to uneven data distribution. When shard exhaustion happens, it requires updating the sharding function and moving data around. Consistent hashing, which will be discussed in Chapter 5, is a commonly used technique to solve this problem.
Celebrity problem: This is also called a hotspot key 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 will be overwhelmed with read operations. To solve this problem, we may need to allocate a shard for each celebrity. Each shard might even require further partition.
Join and de-normalization: Once a database has been sharded across multiple servers, it is hard to perform join operations across database shards. A common workaround is to de- normalize the database so that queries can be performed in a single table.

CHAPTER 4: DESIGN A RATE LIMITER

While designing a rate limiter, an important question to ask ourselves is: where should the rater limiter be implemented, on the server-side or in a gateway? There is no absolute answer. It depends on your company's current technology stack, engineering resources, priorities, goals, etc. Here are a few general guidelines:
• Evaluate your current technology stack, such as programming language, cache service, etc. Make sure your current programming language is efficient to implement rate limiting on the server-side.
• Identify the rate limiting algorithm that fits your business needs. When you implement everything on the server-side, you have full control of the algorithm. However, your choice might be limited if you use a third-party gateway.
• If you have already used microservice architecture and included an API gateway in the design to perform authentication, IP whitelisting, etc., you may add a rate limiter to the API gateway.
• Building your own rate limiting service takes time. If you do not have enough engineering resources to implement a rate limiter, a commercial API gateway is a better option.
Algorithms for rate limiting
Rate limiting can be implemented using different algorithms, and each of them has distinct pros and cons. Even though this chapter does not focus on algorithms, understanding them at high-level helps to choose the right algorithm or combination of algorithms to fit our use cases. Here is a list of popular algorithms:
• Token bucket
• Leaking bucket
• Fixed window counter • Sliding window log
• Sliding window counter

Token bucket -
A token bucket is a container that has pre-defined capacity. Tokens are put in the bucket at preset rates periodically. Once the bucket is full, no more tokens are added.

The token bucket algorithm takes two parameters:
• Bucket size: the maximum number of tokens allowed in the bucket • Refill rate: number of tokens put into the bucket every second
How many buckets do we need? This varies, and it depends on the rate-limiting rules. Here are a few examples.
• It is usually necessary to have different buckets for different API endpoints. For instance, if a user is allowed to make 1 post per second, add 150 friends per day, and like 5 posts per second, 3 buckets are required for each user.
• If we need to throttle requests based on IP addresses, each IP address requires a bucket.
• If the system allows a maximum of 10,000 requests per second, it makes sense to have a global bucket shared by all requests.

Pros:
• The algorithm is easy to implement.
• Memory efficient.
• Token bucket allows a burst of traffic for short periods. A request can go through as long as there are tokens left.
Cons:
• Two parameters in the algorithm are bucket size and token refill rate. However, it might be challenging to tune them properly.

Leaking bucket algorithm

The leaking bucket algorithm is similar to the token bucket except that requests are processed at a fixed rate. It is usually implemented with a first-in-first-out (FIFO) queue. The algorithm works as follows:
• When a request arrives, the system checks if the queue is full. If it is not full, the request is added to the queue.
• Otherwise, the request is dropped.
• Requests are pulled from the queue and processed at regular intervals.

Leaking bucket algorithm takes the following two parameters:
• Bucket size: it is equal to the queue size. The queue holds the requests to be processed at a fixed rate.
• Outflow rate: it defines how many requests can be processed at a fixed rate, usually in seconds.

Pros:
• Memory efficient given the limited queue size.
• Requests are processed at a fixed rate therefore it is suitable for use cases that a stable outflow rate is needed.
Cons:
• A burst of traffic fills up the queue with old requests, and if they are not processed in time, recent requests will be rate limited.
• There are two parameters in the algorithm. It might not be easy to tune them properly.

Fixed window counter algorithm

Fixed window counter algorithm works as follows:
• The algorithm divides the timeline into fix-sized time windows and assign a counter for each window.
• Each request increments the counter by one.
• Once the counter reaches the pre-defined threshold, new requests are dropped until a new time window starts.

A major problem with this algorithm is that a burst of traffic at the edges of

time windows could cause more requests than allowed quota to go through

Pros:
• Memory efficient.
• Easy to understand.
• Resetting available quota at the end of a unit time window fits certain use cases.
Cons:
• Spike in traffic at the edges of a window could cause more requests than the allowed quota to go through.

Sliding window log algorithm

As discussed previously, the fixed window counter algorithm has a major issue: it allows more requests to go through at the edges of a window. The sliding window log algorithm fixes the issue. It works as follows:
• The algorithm keeps track of request timestamps. Timestamp data is usually kept in cache, such as sorted sets of Redis [8].
• When a new request comes in, remove all the outdated timestamps. Outdated timestamps are defined as those older than the start of the current time window.
• Add timestamp of the new request to the log.
• If the log size is the same or lower than the allowed count, a request is accepted. Otherwise, it is rejected.
We explain the algorithm with an example as revealed in Figure 4-10.
In this example, the rate limiter allows 2 requests per minute. Usually, Linux timestamps are stored in the log. However, human-readable representation of time is used in our example for better readability.
• The log is empty when a new request arrives at 1:00:01. Thus, the request is allowed.

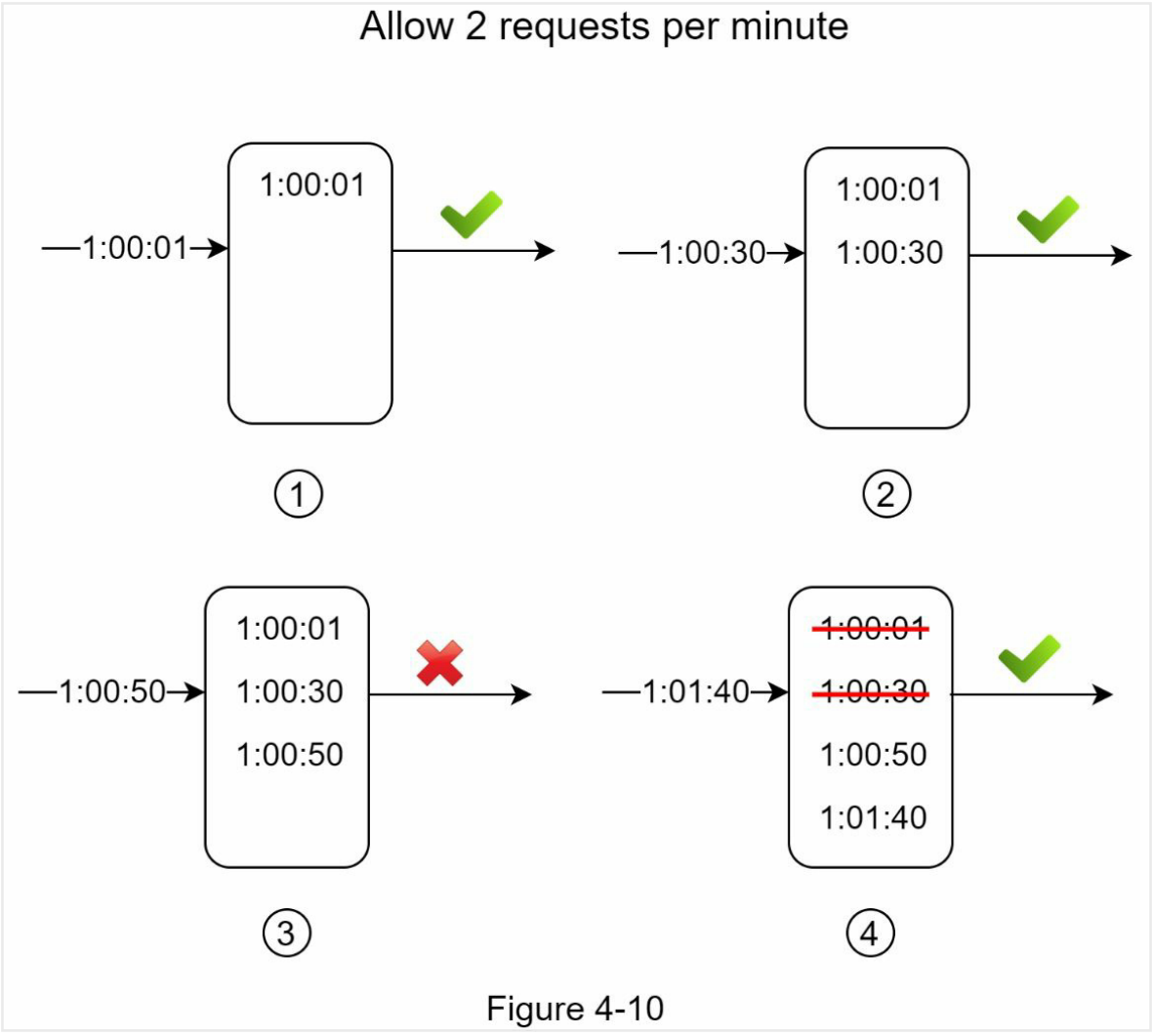


Figure 4-10

• A new request arrives at 1:00:30, the timestamp 1:00:30 is inserted into the log. After the insertion, the log size is 2, not larger than the allowed count. Thus, the request is allowed.
• A new request arrives at 1:00:50, and the timestamp is inserted into the log. After the insertion, the log size is 3, larger than the allowed size 2. Therefore, this request is rejected even though the timestamp remains in the log.
• A new request arrives at 1:01:40. Requests in the range [1:00:40,1:01:40) are within the latest time frame, but requests sent before 1:00:40 are outdated. Two outdated timestamps, 1:00:01 and 1:00:30, are removed from the log. After the remove operation, the log size becomes 2; therefore, the request is accepted.

Pros:

• Rate limiting implemented by this algorithm is very accurate. In any rolling window, requests will not exceed the rate limit.

Cons:

• The algorithm consumes a lot of memory because even if a request is rejected, its timestamp might still be stored in memory.

Sliding window counter algorithm (watch online video for this)

The sliding window counter algorithm is a hybrid approach that combines the

fixed window counter and sliding window log. The algorithm can be implemented by two different approaches. We will explain one implementation in this section and provide reference for the other implementation at the end of the section. Figure 4-11 illustrates how this algorithm works.
Assume the rate limiter allows a maximum of 7 requests per minute, and there are 5 requests in the previous minute and 3 in the current minute. For a new request that arrives at a 30% position in the current minute, the number of requests in the rolling window is calculated using the following formula:
• Requests in current window + requests in the previous window * overlap percentage of the rolling window and previous window

The tough part is this:

1. Api crashes at 1000 request per second. How do I determine the rate at which the requests are prevented. 1000? 800? and why.
2. Api has 1:3 requests for POST and GET. How to determine rate limit with this information.
3. Where to deploy the rate limiter? How to reduce network calls?

1. As mentioned in the post below, run experiments as well as load tests to have a deterministic rate at which requests are dropped. We should also use that information to apply global rate limit on the application. So, if the rate determined after our experiments and load test is 1000 requests / second, we can cap global rate limit at 800.
2. This means that if the rate limit for POST requests is 200 requests / second, GET should be 600 requests / second.
3. We can deploy the rate limiter as part of the API gateway.
4. To reduce network calls, we can cache the GET requests on the client. This however, introduces some complexities. We need to figure out how to ensure that if there is a change in the data, clients will discard their caches and go back to the server to fetch the updated records. If we know the rate at which data changes occur, then we can build it into our implementation. One way to tackle this is to invalidate client cache say every 3 seconds. Each time client has need to serve data, it first checks if the cache is still valid. If the cache is still valid and the data it's looking for is in its cache, it will serve from the cache, otherwise, it will do a network call.
   This approach will generally reduce the number of calls the service receives but also introduces an eventual consistency that is within 3 seconds. Our interviewer should be happy with this. What if we don't have control over the client?
   We could resort to adding more workers to help with serving the api requests. We should also apply caching on the server side. While this will generally improve the health of our servers, it does not reduce network calls.

CHAPTER 5: DESIGN CONSISTENT HASHING

The rehashing problem

This approach works well when the size of the server pool is fixed, and the data distribution is even. However, problems arise when new servers are added, or existing servers are removed. For example, if server 1 goes offline, the size of the server pool becomes 3. Using the same hash function, we get the same hash value for a key. But applying modular operation gives us different server indexes because the number of servers is reduced by 1

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 remapped 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

Two issues in the basic approach
The consistent hashing algorithm was introduced by Karger et al. at MIT [1]. The basic steps are:
• Map servers and keys on to the ring using a uniformly distributed hash function.
• To find out which server a key is mapped to, go clockwise from the key position until the first server on the ring is found.
Two problems are identified with this approach. First, it is impossible to keep the same size of partitions on the ring for all servers considering a server can be added or removed. A partition is the hash space between adjacent servers. It is possible that the size of the partitions on the ring assigned to each server is very small or fairly large.

As the number of virtual nodes increases, the distribution of keys becomes more balanced. This is because the standard deviation gets smaller with more virtual nodes, leading to balanced data distribution. Standard deviation measures how data are spread out. The outcome of an experiment carried out by online research [2] shows that with one or two hundred virtual nodes, the standard deviation is between 5% (200 virtual nodes) and 10% (100 virtual nodes) of the mean. The standard deviation will be smaller when we increase the number of virtual nodes. However, more spaces are needed to store data about virtual nodes. This is a tradeoff, and we can tune the number of virtual nodes to fit our system requirements.

Wrap up
In this chapter, we had an in-depth discussion about consistent hashing, including why it is needed and how it works. The benefits of consistent hashing include:

• Minimized keys are redistributed when servers are added or removed.
• It is easy to scale horizontally because data are more evenly distributed.
• Mitigate hotspot key 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. Consistent hashing helps to mitigate the problem

CHAPTER 6: DESIGN A KEY-VALUE STORE

CAP theorem
CAP theorem states it is impossible for a distributed system to simultaneously provide more than two of these three guarantees: consistency, availability, and partition tolerance. Let us establish a few definitions.
Consistency: consistency means all clients see the same data at the same time no matter which node they connect to.
Availability: availability means any client which requests data gets a response even if some of the nodes are down.
Partition Tolerance: a partition indicates a communication break between two nodes. Partition tolerance means the system continues to operate despite network partitions.
CAP theorem states that one of the three properties must be sacrificed to support 2 of the 3 properties as shown in Figure 6-1.
Nowadays, key-value stores are classified based on the two CAP characteristics they support:
CP (consistency and partition tolerance) systems: a CP key-value store supports consistency and partition tolerance while sacrificing availability.
AP (availability and partition tolerance) systems: an AP key-value store supports availability and partition tolerance while sacrificing consistency.
CA (consistency and availability) systems: a CA key-value store supports consistency and

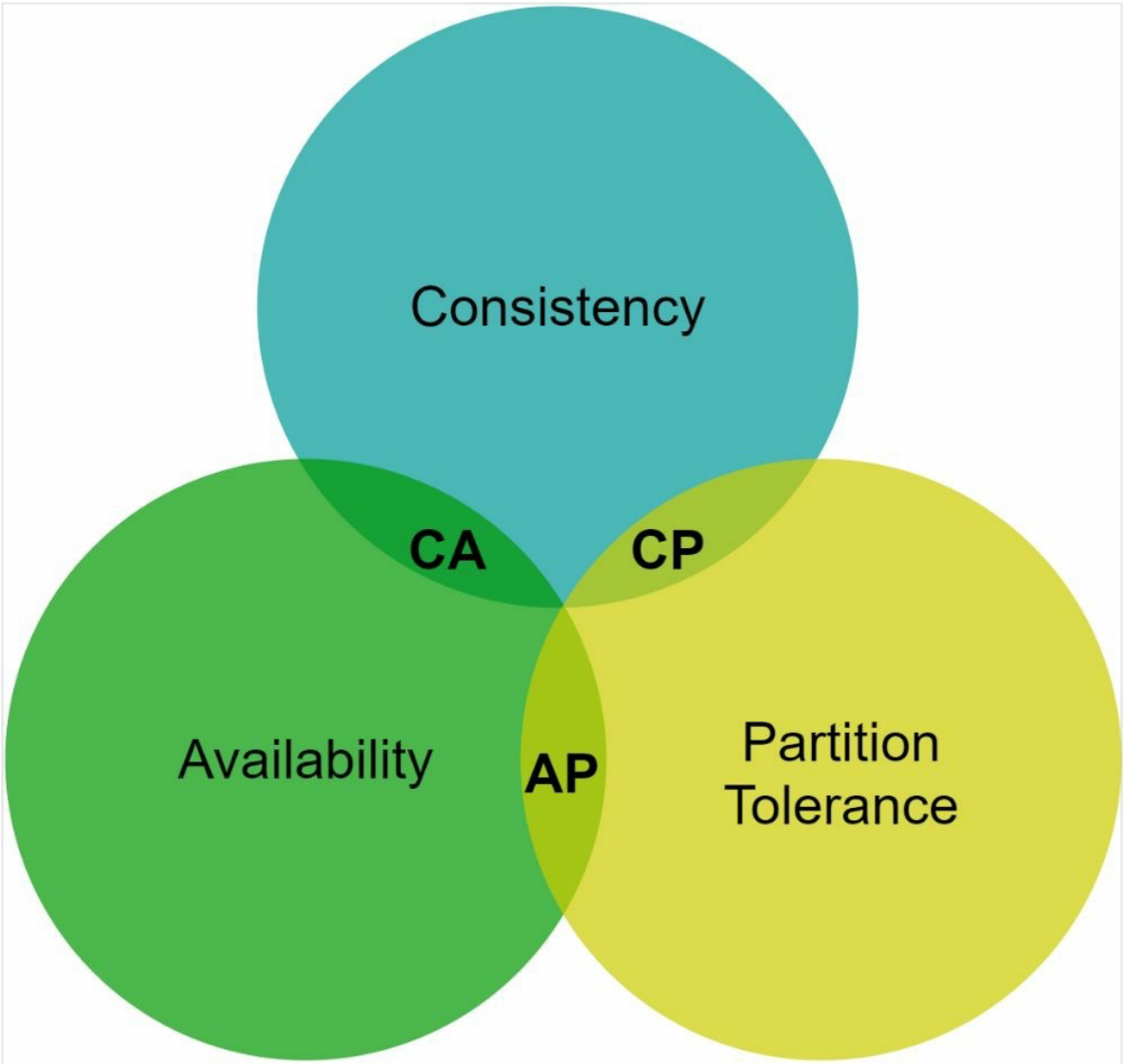


Figure 6-1

availability while sacrificing partition tolerance. Since network failure is unavoidable, a distributed system must tolerate network partition. Thus, a CA system cannot exist in real- world applications.

Bank systems usually have extremely high consistent requirements. For example, it is crucial for a bank system to display the most up-to-date balance info. If inconsistency occurs due to a network partition, the bank system returns an error before the inconsistency is resolved.
However, if we choose availability over consistency (AP system), the system keeps accepting reads, even though it might return stale data. For writes, *n1* and *n2* will keep accepting writes, and data will be synced to *n3* when the network partition is resolved.
Choosing the right CAP guarantees that fit your use case is an important step in building a distributed key-value store. You can discuss this with your interviewer and design the system accordingly.

System components

In this section, we will discuss the following core components and techniques used to build a key-value store:

- Data partition
- Data replication
- Consistency
- Inconsistency resolution
- Handling failures
- System architecture diagram • Write path
- Read path

Data partition

For large applications, it is infeasible to fit the complete data set in a single server. The simplest way to accomplish this is to split the data into smaller partitions and store them in multiple servers. There are two challenges while partitioning the data:

- Distribute data across multiple servers evenly.
- Minimize data movement when nodes are added or removed.

Data replication

To achieve high availability and reliability, data must be replicated asynchronously over *N* servers, where *N* is a configurable parameter. These *N* servers are chosen using the following logic: 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 6-5 (*N = 3*), *key0* is replicated at *s1, s2,* and *s3*.

With virtual nodes, the first *N* nodes on the ring may be owned by fewer than *N* physical servers. To avoid this issue, we only choose unique servers while performing the clockwise walk logic.

Nodes in the same data center often fail at the same time due to power outages, network issues, natural disasters, etc. For better reliability, replicas are placed in distinct data centers, and data centers are connected through high-speed networks.

How to configure *N, W*, and *R* to fit our use cases? Here are some of the possible setups:

If *R = 1* and *W = N*, the system is optimized for a fast read.

If *W = 1 and R = N*, the system is optimized for fast write.

If *W + R > N*, strong consistency is guaranteed (Usually *N = 3, W = R = 2*).

If *W + R <= N*, strong consistency is not guaranteed.

Consistency models

Consistency model is other important factor to consider when designing a key-value store. A consistency model defines the degree of data consistency, and a

wide spectrum of possible consistency models exist:
• Strong consistency: any read operation returns a value corresponding to the result of the most updated write data item. A client never sees out-of-date data.
• Weak consistency: subsequent read operations may not see the most updated value.
• Eventual consistency: this is a specific form of weak consistency. Given enough time, all updates are propagated, and all replicas are consistent.
Strong consistency is usually achieved by forcing a replica not to accept new reads/writes until every replica has agreed on current write. This approach is not ideal for highly available systems because it could block new operations. Dynamo and Cassandra adopt eventual consistency, which is our recommended consistency model for our key-value store. From concurrent writes, eventual consistency allows inconsistent values to enter the system and force the client to read the values to reconcile. The next section explains how reconciliation works with versioning.

Handling failures
As with any large system at scale, failures are not only inevitable but common. Handling failure scenarios is very important. In this section, we first introduce techniques to detect failures. Then, we go over common failure resolution strategies.
Failure detection

A better solution is to use decentralized failure detection methods like gossip protocol. Gossip protocol works as follows:
• Each node maintains a node membership list, which contains member IDs and heartbeat counters.
• Each node periodically increments its heartbeat counter.
• Each node periodically sends heartbeats to a set of random nodes, which in turn propagate to another set of nodes.
• Once nodes receive heartbeats, membership list is updated to the latest info.
• If the heartbeat has not increased for more than predefined periods, the member is considered as offline.

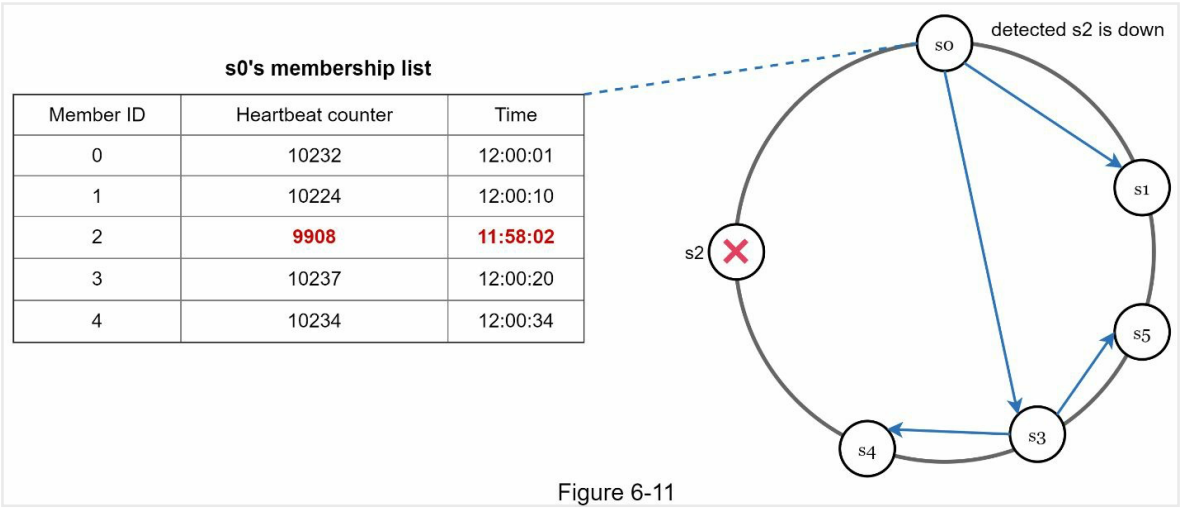


Figure 6-11

As shown in Figure 6-11:
• Node *s0* maintains a node membership list shown on the left side.
• Node *s0* notices that node s2's (member ID = 2) heartbeat counter has not increased for a
long time.
• Node *s0* sends heartbeats that include *s2*'s info to a set of random nodes. Once other nodes confirm that *s2*'s heartbeat counter has not been updated for a long time, node *s2* is marked down, and this information is propagated to other nodes.

Handling temporary failures

After failures have been detected through the gossip protocol, the system needs to deploy certain mechanisms to ensure availability. In the strict quorum approach, read and write operations could be blocked as illustrated in the quorum consensus section.

A technique called "sloppy quorum" [4] is used to improve availability. Instead of enforcing the quorum requirement, the system chooses the first *W* healthy servers for writes and first *R* healthy servers for reads on the hash ring. Offline servers are ignored.

If a server is unavailable due to network or server failures, another server will process requests temporarily. When the down server is up, changes will be pushed back to achieve data consistency. This process is called hinted handoff. Since *s2* is unavailable in Figure 6- 12, reads and writes will be handled by *s3* temporarily. When *s2* comes back online, *s3* will hand the data back to *s2*.

Handling permanent failures

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

https://www.youtube.com/watch?v=qHMLy5JjbjQ **(Merkel tree uses cases)**

Merkel trees are used in Dynamo category of databases and especially in Cassandra to handle something called Anti-Entropy(Basically to repair inconsistency during replication). Actually, even those handle anti-entropy using three strategies 1. Hinted Handoff 2.Read Repair and finally in background Merkel Tree kicks in for a wholistic repair.

Step 1: Divide key space into buckets (4 in our example) as shown in Figure 6-13. A bucket is used as the root level node to maintain a limited depth of the tree.
Step 2: Once the buckets are created, hash each key in a bucket using a uniform hashing method (Figure 6-14).
Step 3: Create a single hash node per bucket (Figure 6-15).
Step 4: Build the tree upwards till root by calculating hashes of children (Figure 6-16).

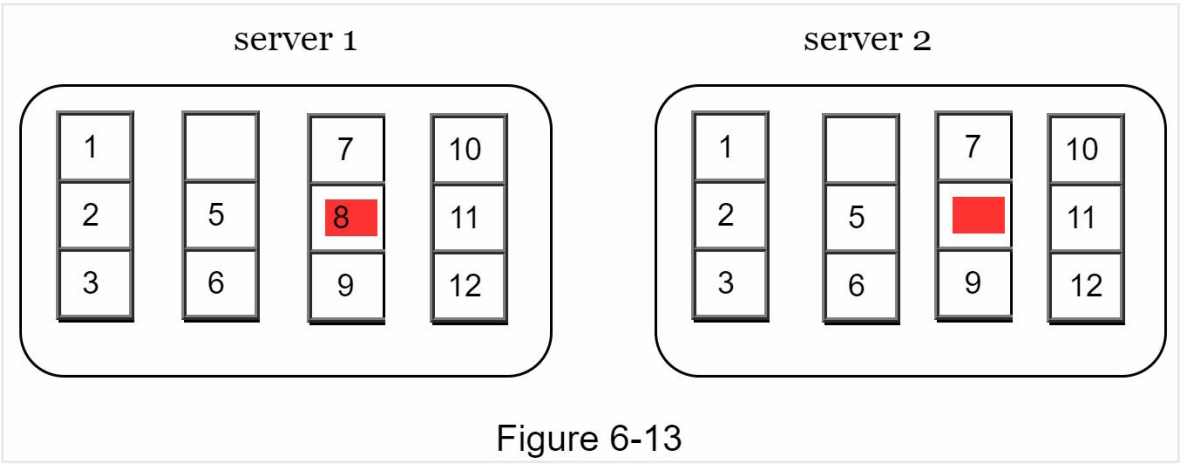


Figure 6-13

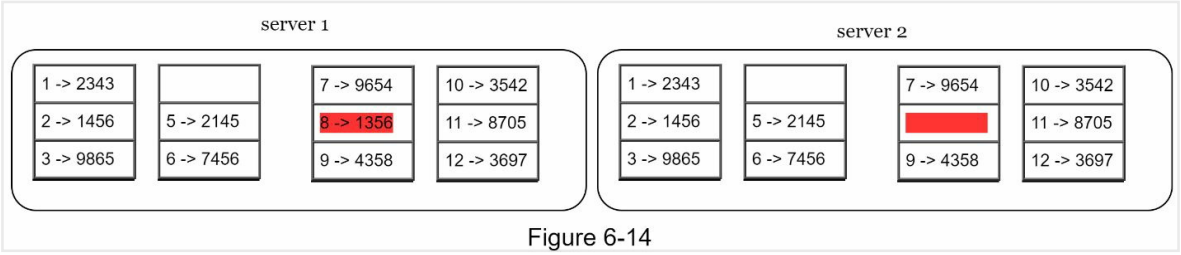


Figure 6-14

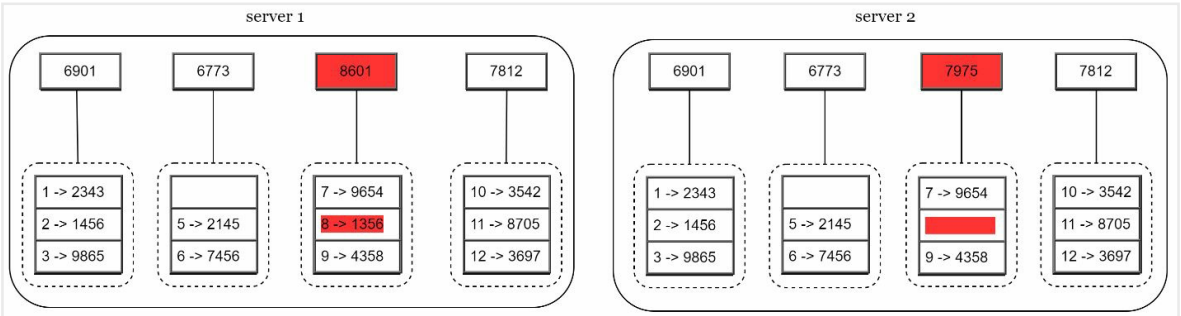


Figure 6-15

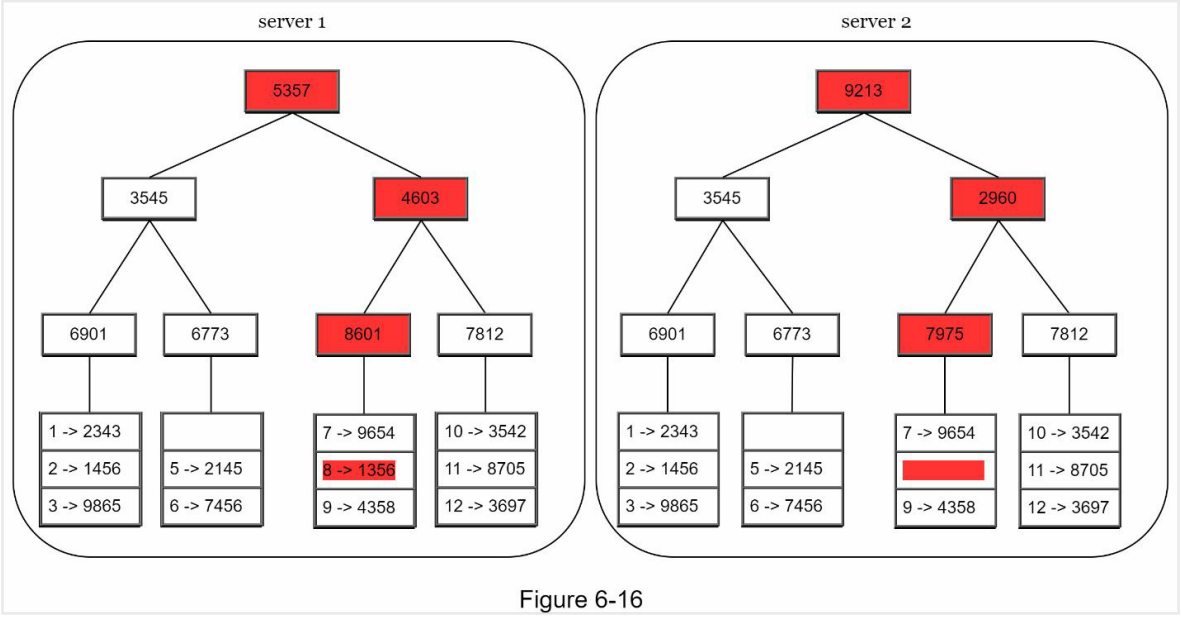


Figure 6-16

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

Using Merkle trees, the amount of data needed to be synchronized is proportional to the differences between the two replicas, and not the amount of data they contain. In real-world systems, the bucket size is quite big. For instance, a possible configuration is one million buckets per one billion keys, so each bucket only contains 1000 keys.

This chapter covers many concepts and techniques. To refresh your memory, the following table summarizes features and corresponding techniques used for a distributed key-value store.

Goal/Problems	Technique
Ability to store big data	Use consistent hashing to spread the load across servers
High availability reads	Data replication Multi-data center setup
Highly available writes	Versioning and conflict resolution with vector clocks
Dataset partition	Consistent Hashing
Incremental scalability	Consistent Hashing
Heterogeneity	Consistent Hashing
Tunable consistency	Quorum consensus
Handling temporary failures	Sloppy quorum and hinted handoff
Handling permanent failures	Merkle tree
Handling data center outage	Cross-data center replication

Table 6-2

CHAPTER 7: DESIGN A UNIQUE ID GENERATOR IN DISTRIBUTED SYSTEMS

Step 1 - Understand the problem and establish design scope
Asking clarification questions is the first step to tackle any system design interview question. Here is an example of candidate-interviewer interaction:
Candidate: What are the characteristics of unique IDs? **Interviewer**: IDs must be unique and sortable.
Candidate: For each new record, does ID increment by 1?
Interviewer: The ID increments by time but not necessarily only increments by 1. IDs created in the evening are larger than those created in the morning on the same day.
Candidate: Do IDs only contain numerical values? **Interviewer**: Yes, that is correct.
Candidate: What is the ID length requirement? **Interviewer**: IDs should fit into 64-bit.
Candidate: What is the scale of the system?
Interviewer: The system should be able to generate 10,000 IDs per second.

Multiple options can be used to generate unique IDs in distributed systems. The options we considered are:

• Multi-master replication
• Universally unique identifier (UUID) • Ticket server
• Twitter snowflake approach

Multi-master replication

This approach uses the databases' *auto_increment* feature. Instead of increasing the next ID by 1, we increase it by *k,* where *k* is the number of database servers in use.

UUID

A UUID is another easy way to obtain unique IDs. UUID is a 128-bit number used to identify information in computer systems. UUID has a very low probability of getting collusion.

In this design, each web server contains an ID generator, and a web server is responsible for generating IDs independently.
Pros:
• Generating UUID is simple. No coordination between servers is needed so there will not be any synchronization issues.
• The system is easy to scale because each web server is responsible for generating IDs they consume. ID generator can easily scale with web servers.
Cons:
• IDs are 128 bits long, but our requirement is 64 bits. • IDs do not go up with time.
• IDs could be non-numeric.

Ticket Server

The idea is to use a centralized *auto_increment* feature in a single database server (Ticket Server). To learn more about this, refer to flicker's engineering blog article [2].
Pros:
• Numeric IDs.
• It is easy to implement, and it works for small to medium-scale applications.
Cons:
• Single point of failure. Single ticket server means if the ticket server goes down, all systems that depend on it will face issues. To avoid a single point of failure, we can set up multiple ticket servers. However, this will introduce new challenges such as data synchronization.

Twitter snowflake approach

Approaches mentioned above give us some ideas about how different ID

generation systems work. However, none of them meet our specific requirements; thus, we need another approach. Twitter's unique ID generation system called "snowflake" [3] is inspiring and can satisfy our requirements.
Divide and conquer is our friend. Instead of generating an ID directly, we divide an ID into different sections. Figure 7-5 shows the layout of a 64-bit ID.
Each section is explained below.
• Sign bit: 1 bit. It will always be 0. This is reserved for future uses. It can potentially be used to distinguish between signed and unsigned numbers.
• Timestamp: 41 bits. Milliseconds since the epoch or custom epoch. We use Twitter snowflake default epoch 1288834974657, equivalent to Nov 04, 2010, 01:42:54 UTC.
• Datacenter ID: 5 bits, which gives us *2 ^ 5 = 32* datacenters.
• Machine ID: 5 bits, which gives us *2 ^ 5 = 32* machines per datacenter.
• Sequence number: 12 bits. For every ID generated on that machine/process, the sequence number is incremented by 1. The number is reset to 0 every millisecond.

In the high-level design, we discussed various options to design a unique ID generator in distributed systems. We settle on an approach that is based on the Twitter snowflake ID generator. Let us dive deep into the design. To refresh our memory, the design diagram is relisted below.
Datacenter IDs and machine IDs are chosen at the startup time, generally fixed once the system is up running. Any changes in datacenter IDs and machine IDs require careful review since an accidental change in those values can lead to ID conflicts. Timestamp and sequence numbers are generated when the ID generator is running.

Timestamp

The most important 41 bits make up the timestamp section. As timestamps grow with time, IDs are sortable by time. Figure 7-7 shows an example of how binary representation is converted to UTC. You can also convert UTC back to binary representation using a similar method.

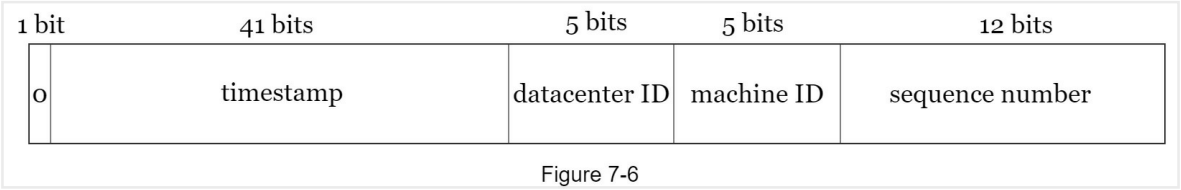


Figure 7-6

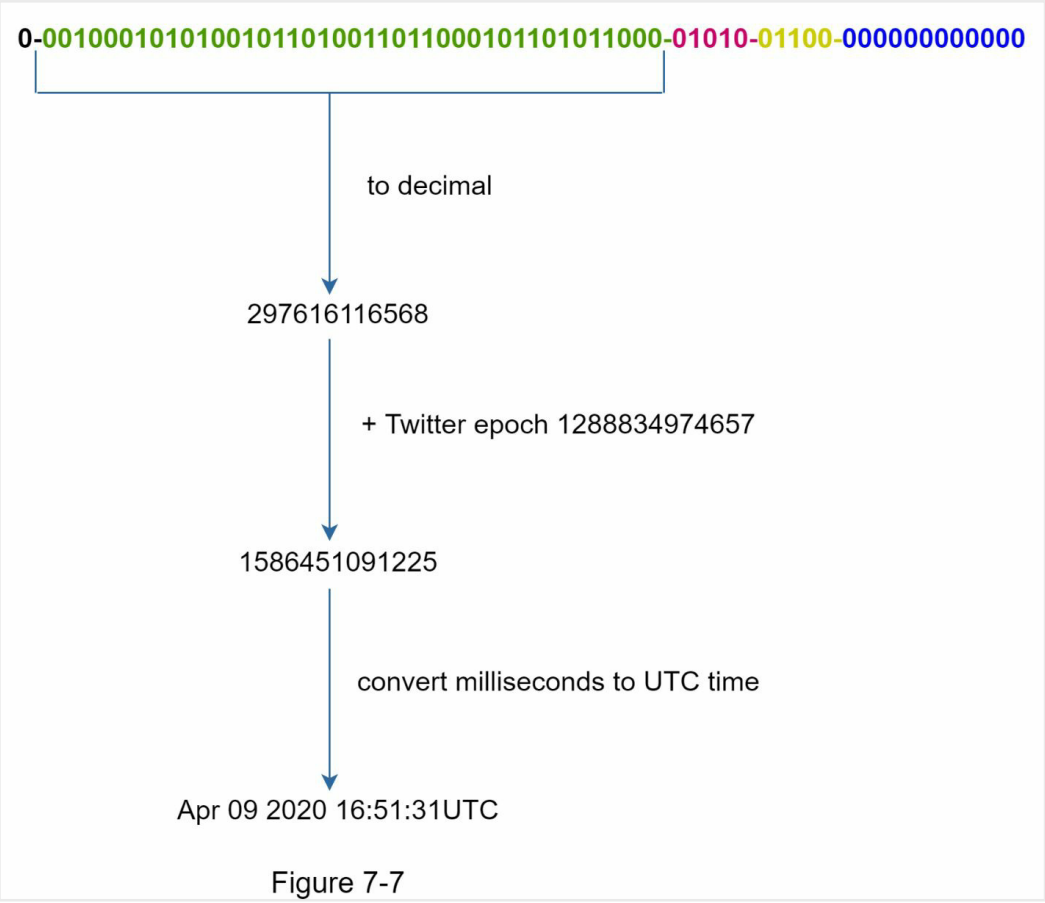


Figure 7-7

The maximum timestamp that can be represented in 41 bits is *2 ^ 41 - 1 = 2199023255551* milliseconds (ms), which gives us: ~ 69 years = *2199023255551 ms / 1000 seconds / 365 days / 24 hours/ 3600 seconds*. This means the ID generator will work for 69 years and having a custom epoch time close to today's date delays the overflow time. After 69 years, we will need a new epoch time or adopt other techniques to migrate IDs.

Sequence number

Sequence number is 12 bits, which give us 2 ^ 12 = 4096 combinations. This field is 0 unless more than one ID is generated in a millisecond on the same server. In theory, a machine can support a maximum of 4096 new IDs per millisecond.

CHAPTER 8: DESIGN A URL SHORTENER

System design interview questions are intentionally left open-ended. To design a well-crafted system, it is critical to ask clarification questions.
Candidate: Can you give an example of how a URL shortener work?
Interviewer: Assume URL https://www.systeminterview.com/q=chatsystem&c=loggedin&v=v3&l=long is the original URL. Your service

creates an alias with shorter length: https://tinyurl.com/ y7keocwj. If you click the alias, it redirects you to the original URL.
Candidate: What is the traffic volume?
Interviewer: 100 million URLs are generated per day.
Candidate: How long is the shortened URL? **Interviewer**: As short as possible.
Candidate: What characters are allowed in the shortened URL?
Interviewer: Shortened URL can be a combination of numbers (0-9) and characters (a-z, A- Z).
Candidate: Can shortened URLs be deleted or updated?
Interviewer: For simplicity, let us assume shortened URLs cannot be deleted or updated.
Here are the basic use cases:
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
Back of the envelope estimation
• Write operation: 100 million URLs are generated per day. • Write operation per second: 100 million / 24 /3600 = 1160
• Read operation: Assuming ratio of read operation to write operation is 10:1, read operation per second: 1160 * 10 = 11,600
• Assuming the URL shortener service will run for 10 years, this means we must support 100 million * 365 * 10 = 365 billion records.
• Assume average URL length is 100.
• Storage requirement over 10 years: 365 billion * 100 bytes * 10 years = 365 TB
It is important for you to walk through the assumptions and calculations with your interviewer so that both of you are on the same page.