Grokking the System Design Interview

2021 – 04 – 12

Junfan Zhu

(junfanz@gatech.edu; junfanzhu@uchicago.edu)

Course Links

https://www.educative.io/courses/grokking-the-system-design-interview/

Table of Contents

---

1. Back-of-the-envelope estimation

Scaling, partitioning, load balancing, and caching.

- What scale is expected from the system?

- How much storage will we need?
- What network bandwidth usage are we expecting?

2. Shortening URL

2.1. Encoding actual URL

Use MD5 algorithm as hash function ⇒ produce a 128-bit hash value.

2.2. Cache

Which cache eviction policy would best fit our needs?

When the cache is full, and we want to replace a link with a newer/hotter URL, how would we choose? . Under this policy, we discard the least recently used URL first. We can use a Linked Hash Map or a similar data structure to store our URLs and Hashes, which will also keep track of the URLs that have been accessed recently.

To further increase the efficiency, we can replicate our caching servers to distribute the load between them.

2.3. Load Balancer (LB)

We can add a Load balancing layer at three places in our system:

- Between Clients and Application servers
- Between Application Servers and database servers
- Between Application Servers and Cache servers

3. DropBox

3.1. Clarify Requirements and Goals of the System

What do we wish to achieve from a Cloud Storage system? Here are the top-level requirements for our system:

- Users should be able to upload and download their files/photos from any device. Users should be able to share files or folders with other users.

- Our service should support automatic synchronization between devices, i.e., after updating a file on one device, it should get synchronized on all devices.
- The system should support storing large files up to a GB.
- ACID-ity is required. Atomicity, Consistency, Isolation and Durability of all file operations should be guaranteed.
- Our system should support offline editing. Users should be able to add/delete/modify files while offline, and as soon as they come online, all their changes should be synced to the remote servers and other online devices.

4. Facebook Messenger

4.1. Message Handling

How does the messenger maintain the sequencing of the messages? We can store a timestamp with each message, which is the time the message is received by the server. This will still not ensure the correct ordering of messages for clients. The scenario where the server timestamp cannot determine the exact order of messages would look like this:

- User-1 sends a message M1 to the server for User-2.
- The server receives M1 at T1.
- Meanwhile, User-2 sends a message M2 to the server for User-1.
- The server receives the message M2 at T2, such that T2 > T1.
- The server sends message M1 to User-2 and M2 to User-1.
- So User-1 will see M1 first and then M2, whereas User-2 will see M2 first and then M1.

To resolve this, we need to keep a sequence number with every message for each client. This sequence number will determine the exact ordering of messages for EACH user. With this solution, both clients will see a different view of the message sequence, but this view will be consistent for them on all devices.

4.2. Storing and retrieving the messages from the database

Whenever the chat server receives a new message, it needs to store it in the database. To do so, we have two options:

- Start a separate thread, which will work with the database to store the message.
- Send an asynchronous request to the database to store the message.

We have to keep certain things in mind while designing our database:

- How to efficiently work with the database connection pool.
- How to retry failed requests.
- Where to log those requests that failed even after some retries.
- How to retry these logged requests (that failed after the retry) when all the issues have resolved.

Which storage system we should use?

We need to have a database that can support a very high rate of small updates and also fetch a range of records quickly. We cannot use RDBMS like MySQL or NoSQL like MongoDB because we cannot afford to read/write a row from the database every time a user receives/sends a message. This will not only make the basic operations of our service run with high latency but also create a huge load on databases.

Both of our requirements can be easily met with a wide-column database solution like HBase. HBase is a column-oriented key-value NoSQL database that can store multiple values against one key into multiple columns. HBase is modeled after Google's BigTable and runs on top of Hadoop Distributed File System (HDFS). HBase groups data together to store new data in a memory buffer and, once the buffer is full, it dumps the data to the disk. This way of storage not only helps to store a lot of small data quickly but also fetching rows by the key or scanning ranges of rows. HBase is also an efficient database to store variable-sized data, which is also required by our service.

Design Summary:

Clients will open a connection to the chat server to send a message; the server will then pass it to the requested user. All the active users will keep a connection open with the server to receive messages. Whenever a new message arrives, the chat server will push it to the receiving user on the long poll request. Messages can be stored in HBase, which supports quick small updates, and range based searches. The servers can broadcast the online status of a user to other relevant users. Clients can pull status updates for users who are visible in the client's viewport on a less frequent basis.

5. YouTube

5.1. Metadata Sharding

Since we have a huge number of new videos every day and our read load is extremely high, therefore, we need to distribute our data onto multiple machines so that we can perform read/write operations efficiently.

5.1.1. Sharding based on UserID

We can try storing all the data for a particular user on one server. While storing, we can pass the UserID to our hash function, which will map the user to a database server where we will store all the metadata for that user's videos. While querying for videos of a user, we can ask our hash function to find the server holding the user's data and then read it from there. To search videos by titles, we will have to query all servers, and each server will return a set of videos. A centralized server will then aggregate and rank these results before returning them to the user.

5.1.2. Sharding based on VideoID

Our hash function will map each VideoID to a random server where we will store that Video's metadata. To find videos of a user, we will query all servers, and each server will return a set of videos. A centralized server will aggregate and rank these results before returning them to the user. This approach solves our problem of popular users but shifts it to popular videos.

5.2. Load Balancing

We should use Consistent Hashing among our cache servers, which will also help in balancing the load between cache servers. Since we will be using a static hash-based scheme to map videos to hostnames, it can lead to an uneven load on the logical replicas due to each video's different popularity. For instance, if a video becomes popular, the logical replica corresponding to that video will experience more traffic than other servers. These uneven loads for logical replicas can then translate into uneven load distribution on corresponding physical servers. To resolve this issue, any busy server in one location can redirect a client to a less busy server in the same cache location. We can use dynamic HTTP redirections for this scenario. Consistent hashing will not only help in replacing a dead server but also help in distributing load among servers.

However, the use of redirections also has its drawbacks. First, since our service tries to load balance locally, it leads to multiple redirections if the host that receives the redirection can't serve the video. Also, each redirection requires a client to make an additional HTTP request; it also leads to higher delays before the video starts playing back. Moreover, inter-tier (or cross data-center) redirections lead a client to a distant cache location because the higher tier caches are only present at a small number of locations.

6. Designing Typeahead Suggestion

We can have a Map-Reduce (MR) set-up to process all the logging data periodically say every hour. These MR jobs will calculate frequencies of all searched terms in the past hour. We can then update our trie with this new data. We can take the current snapshot of the trie and update it with all the new terms and their frequencies. We should do this offline as we don't want our read queries to be blocked by update trie requests. We can have two options:

- We can make a copy of the trie on each server to update it offline. Once done we can switch to start using it and discard the old one.
- Another option is we can have a primary-secondary configuration for each trie server. We can update the secondary while the primary is serving traffic. Once the update is complete, we can make the secondary our new primary. We can later update our old primary, which can then start serving traffic, too.

7. API Rate Limiter

Rate Limiting helps to protect services against abusive behaviors targeting the application layer like Denial-of-service (DOS) attacks, brute-force password attempts, brute-force credit card transactions, etc. These attacks are usually a barrage of HTTP/S requests which may look like they are coming from real users, but are typically generated by machines (or bots). As a result, these attacks are often harder to detect and can more easily bring down a service, application, or an API.

Rate limiting is also used to prevent revenue loss, to reduce infrastructure costs, to stop spam, and to stop online harassment. Following is a list of scenarios that can benefit from Rate limiting by making a service (or API) more reliable:

- Misbehaving clients/scripts: Either intentionally or unintentionally, some entities can overwhelm a service by sending a large number of requests. Another scenario could be when a user is sending a lot of lower-priority requests and we want to make sure that it doesn't affect the high-priority traffic. For example, users sending a high volume of requests for analytics data should not be allowed to hamper critical transactions for other users.
- Security: By limiting the number of the second-factor attempts (in 2-factor auth) that the users are allowed to perform, for example, the number of times they're allowed to try with a wrong password.
- To prevent abusive behavior and bad design practices: Without API limits, developers of client applications would use sloppy development tactics, for example, requesting the same information over and over again.

- To keep costs and resource usage under control: Services are generally designed for normal input behavior, for example, a user writing a single post in a minute. Computers could easily push thousands/second through an API. Rate limiter enables controls on service APIs.
- Revenue: Certain services might want to limit operations based on the tier of their customer's service and thus create a revenue model based on rate limiting. There could be default limits for all the APIs a service offers. To go beyond that, the user has to buy higher limits
- To eliminate spikiness in traffic: Make sure the service stays up for everyone else.

8. Web Crawler

8.1. How to crawl?

Breadth-first or depth-first? Breadth First Search (BFS) is usually used. However, Depth First Search (DFS) is also utilized in some situations, such as, if your crawler has already established a connection with the website, it might just DFS all the URLs within this website to save some handshaking overhead.

Path-ascending crawling: Path-ascending crawling can help discover a lot of isolated resources or resources for which no inbound link would have been found in regular crawling of a particular Web site. In this scheme, a crawler would ascend to every path in each URL that it intends to crawl.

8.2. Component Design

1. The URL frontier: The URL frontier is the data structure that contains all the URLs that remain to be downloaded. We can crawl by performing a breadth-first traversal of the Web, starting from the pages in the seed set. Such traversals are easily implemented by using a FIFO queue.

2. The fetcher module: The purpose of a fetcher module is to download the document corresponding to a given URL using the appropriate network protocol like HTTP. As discussed above, webmasters create robot.txt to make certain parts of their websites off-limits for the crawler. To avoid downloading this file on every request, our crawler's HTTP protocol module can maintain a fixed-sized cache mapping host-names to their robot's exclusion rules.

3. Document input stream: Our crawler's design enables the same document to be processed by multiple processing modules. To avoid downloading a document multiple times, we cache the document locally using an abstraction called a Document Input Stream (DIS).

4. Document Dedupe test: Many documents on the Web are available under multiple, different URLs. There are also many cases in which documents are mirrored on various servers. Both of these effects will cause any Web crawler to download the same document multiple times. To prevent the processing of a document more than once, we perform a dedupe test on each document to remove duplication.

5. URL filters: The URL filtering mechanism provides a customizable way to control the set of URLs that are downloaded. This is used to blacklist websites so that our crawler can ignore them. Before adding each URL to the frontier, the worker thread consults the user-supplied URL filter. We can define filters to restrict URLs by domain, prefix, or protocol type.

6. Domain name resolution: Before contacting a Web server, a Web crawler must use the Domain Name Service (DNS) to map the Web server's hostname into an IP address. DNS name resolution will be a big bottleneck of our crawlers given the amount of URLs we will be working with. To avoid repeated requests, we can start caching DNS results by building our local DNS server.

7. URL dedupe test: While extracting links, any Web crawler will encounter multiple links to the same document. To avoid downloading and processing a document multiple times, a URL dedupe test must be performed on each extracted link before adding it to the URL frontier.

8. Checkpointing: A crawl of the entire Web takes weeks to complete. To guard against failures, our crawler can write regular snapshots of its state to the disk. An interrupted or aborted crawl can easily be restarted from the latest checkpoint.

9. Facebook's Newsfeed

Component Design

9.1. Feed generation

Offline generation for newsfeed: We can have dedicated servers that are continuously generating users' newsfeed and storing them in memory. So, whenever a user requests for the new posts for their feed, we can simply serve it from the pre-generated, stored location. Using this scheme, user's newsfeed is not compiled on load, but rather on a regular basis and returned to users whenever they request for it.

We can store FeedItemIDs in a data structure similar to Linked HashMap or TreeMap, which can allow us to not only jump to any feed item but also iterate

through the map easily. Whenever users want to fetch more feed items, they can send the last FeedItemID they currently see in their newsfeed, we can then jump to that FeedItemID in our hash-map and return next batch/page of feed items from there.

9.2. Feed publishing

The process of pushing a post to all the followers is called fanout. By analogy, the push approach is called fanout-on-write, while the pull approach is called fanout-on-load. Let's discuss different options for publishing feed data to users.

1. "Pull" model or Fan-out-on-load

This method involves keeping all the recent feed data in memory so that users can pull it from the server whenever they need it. Clients can pull the feed data on a regular basis or manually whenever they need it. Possible problems with this approach are a) New data might not be shown to the users until they issue a pull request, b) It's hard to find the right pull cadence, as most of the time pull requests will result in an empty response if there is no new data, causing waste of resources.

2. "Push" model or Fan-out-on-write.

For a push system, once a user has published a post, we can immediately push this post to all the followers. The advantage is that when fetching feed you don't need to go through your friend's list and get feeds for each of them. It significantly reduces read operations. To efficiently handle this, users have to maintain a Long Poll request with the server for receiving the updates. A possible problem with this approach is that when a user has millions of followers (a celebrity-user) the server has to push updates to a lot of people.

3. Hybrid

An alternate method to handle feed data could be to use a hybrid approach, i.e., to do a combination of fan-out-on-write and fan-out-on-load. Specifically, we can stop pushing posts from users with a high number of followers (a celebrity user) and only push data for those users who have a few hundred (or thousand) followers. For celebrity users, we can let the followers pull the updates. Since the push operation can be extremely costly for users who have a lot of friends or followers, by disabling fanout for them, we can save a huge number of resources. Another alternate approach could be that, once a user publishes a post, we can limit the fanout to only her online friends. Also, to get benefits from both the approaches, a combination of 'push to notify' and 'pull for serving' end-users is a great way to go. Purely a push or pull model is less versatile.

9.3. Data Partitioning

1. Sharding posts and metadata

Since we have a huge number of new posts every day and our read load is extremely high too, we need to distribute our data onto multiple machines such that we can read/write it efficiently. For sharding our databases that are storing posts and their metadata, we can have a similar design as discussed under Designing Twitter.

2. Sharding feed data

For feed data, which is being stored in memory, we can partition it based on UserID. We can try storing all the data of a user on one server. When storing, we can pass the UserID to our hash function that will map the user to a cache server where we will store the user's feed objects. Also, for any given user, since we don't expect to store more than 500 FeedItemIDs, we will not run into a scenario where feed data for a user doesn't fit on a single server. To get the feed of a user, we would always have to query only one server. For future growth and replication, we must use Consistent Hashing.

10. Yelp

10.1. Dynamic size grids

Let's assume we don't want to have more than 500 places in a grid so that we can have a faster searching. So, whenever a grid reaches this limit, we break it down into four grids of equal size and distribute places among them. This means thickly populated areas like downtown San Francisco will have a lot of grids, and sparsely populated area like the Pacific Ocean will have large grids with places only around the coastal lines.

What data-structure can hold this information? A tree in which each node has four children can serve our purpose. Each node will represent a grid and will contain information about all the places in that grid. If a node reaches our limit of 500 places, we will break it down to create four child nodes under it and distribute places among them. In this way, all the leaf nodes will represent the grids that cannot be further broken down. So leaf nodes will keep a list of places with them. This tree structure in which each node can have four children is called a QuadTree.

11. Ticket Master

How would the server keep track of all the active reservations that haven't been booked yet? And how would the server keep track of all the waiting customers?

We need two daemon services, one to keep track of all active reservations and remove any expired reservation from the system; let's call it ActiveReservationService. The other service would be keeping track of all the waiting user requests and, as soon as the required number of seats become available, it will notify the (the longest waiting) user to choose the seats; let's call it WaitingUserService.

11.1. Active Reservations Service

We can keep all the reservations of a 'show' in memory in a data structure similar to Linked HashMap or a TreeMap in addition to keeping all the data in the database. We will need a linked HashMap kind of data structure that allows us to jump to any reservation to remove it when the booking is complete. Also, since we will have expiry time associated with each reservation, the head of the HashMap will always point to the oldest reservation record so that the reservation can be expired when the timeout is reached.

To store every reservation for every show, we can have a HashTable where the 'key' would be 'ShowID', and the 'value' would be the Linked HashMap containing 'BookingID' and creation 'Timestamp'.

In the database, we will store the reservation in the 'Booking' table and the expiry time will be in the Timestamp column. The 'Status' field will have a value of 'Reserved (1)' and, as soon as a booking is complete, the system will update the 'Status' to 'Booked (2)' and remove the reservation record from the Linked HashMap of the relevant show. When the reservation is expired, we can either remove it from the Booking table or mark it 'Expired (3)' in addition to removing it from memory.

ActiveReservationsService will also work with the external financial service to process user payments. Whenever a booking is completed, or a reservation gets expired, WaitingUsersService will get a signal so that any waiting customer can be served.

11.2. Waiting Users Service

Just like ActiveReservationsService, we can keep all the waiting users of a show in memory in a Linked HashMap or a TreeMap. We need a data structure similar to Linked HashMap so that we can jump to any user to remove them from the HashMap when the user cancels their request. Also, since we are serving in a first-come-first-serve manner, the head of the Linked HashMap would always be

pointing to the longest waiting user, so that whenever seats become available, we can serve users in a fair manner.

We will have a HashTable to store all the waiting users for every Show. The 'key' would be 'ShowID, and the 'value' would be a Linked HashMap containing 'UserIDs' and their wait-start-time.

Clients can use Long Polling for keeping themselves updated for their reservation status. Whenever seats become available, the server can use this request to notify the user.

Reservation Expiration On the server, ActiveReservationsService keeps track of expiry (based on reservation time) of active reservations. As the client will be shown a timer (for the expiration time), which could be a little out of sync with the server, we can add a buffer of five seconds on the server to safeguard from a broken experience, such that the client never times out after the server, preventing a successful purchase.

11.3. Concurrency

How to handle concurrency, such that no two users are able to book the same seat. We can use transactions in SQL databases to avoid any clashes. For example, if we are using an SQL server we can utilize Transaction Isolation Levels to lock the rows before we can update them. Here is the sample code:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

    -- Suppose we intend to reserve three seats (IDs: 54, 55, 56) for ShowID=99
    Select * From Show_Seat where ShowID=99 && ShowSeatID in (54, 55, 56) && Status=0 -- fr

    -- if the number of rows returned by the above statement is three, we can update to
    -- return success otherwise return failure to the user.
    update Show_Seat ...
    update Booking ...

COMMIT TRANSACTION;
```

'Serializable' is the highest isolation level and guarantees safety from Dirty, Nonrepeatable, and Phantoms reads. One thing to note here; within a transaction, if we read rows, we get a write lock on them so that they can't be updated by anyone else.

Once the above database transaction is successful, we can start tracking the reservation in ActiveReservationService.

12. Load Balancing

12.1. Benefits

- Users experience faster, uninterrupted service. Users won't have to wait for a single struggling server to finish its previous tasks. Instead, their requests are immediately passed on to a more readily available resource.
- Service providers experience less downtime and higher throughput. Even a full server failure won't affect the end user experience as the load balancer will simply route around it to a healthy server.
- Load balancing makes it easier for system administrators to handle incoming requests while decreasing wait time for users.
- Smart load balancers provide benefits like predictive analytics that determine traffic bottlenecks before they happen. As a result, the smart load balancer gives an organization actionable insights. These are key to automation and can help drive business decisions.
- System administrators experience fewer failed or stressed components. Instead of a single device performing a lot of work, load balancing has several devices perform a little bit of work.

12.2. Algorithms

How does the load balancer choose the backend server?

- Load balancers consider two factors before forwarding a request to a backend server. They will first ensure that the server they choose is actually responding appropriately to requests and then use a pre-configured algorithm to select one from the set of healthy servers. We will discuss these algorithms shortly.
- **Health Checks** - Load balancers should only forward traffic to "healthy" backend servers. To monitor the health of a backend server, "health checks" regularly attempt to connect to backend servers to ensure that servers are listening. If a server fails a health check, it is automatically removed from the pool, and traffic will not be forwarded to it until it responds to the health checks again.

Methods.

1. Least Connection Method — This method directs traffic to the server with the fewest active connections. This approach is quite useful when there are a large number of persistent client connections which are unevenly distributed between the servers.

2. Least Response Time Method — This algorithm directs traffic to the server with the fewest active connections and the lowest average response time.

3. Least Bandwidth Method — This method selects the server that is currently serving the least amount of traffic measured in megabits per second (Mbps).

4. Round Robin Method — This method cycles through a list of servers and sends each new request to the next server. When it reaches the end of the list, it starts over at the beginning. It is most useful when the servers are of equal specification and there are not many persistent connections.

5. Weighted Round Robin Method — The weighted round-robin scheduling is designed to better handle servers with different processing capacities. Each server is assigned a weight (an integer value that indicates the processing capacity). Servers with higher weights receive new connections before those with less weights and servers with higher weights get more connections than those with less weights.

6. IP Hash — Under this method, a hash of the IP address of the client is calculated to redirect the request to a server.

13. Caching

13.1. Cache Invalidation

While caching is fantastic, it requires some maintenance to keep the cache coherent with the source of truth (e.g., database). If the data is modified in the database, it should be invalidated in the cache; if not, this can cause inconsistent application behavior.

Solving this problem is known as cache invalidation; there are three main schemes that are used:

1. Write-through cache: Under this scheme, data is written into the cache and the corresponding database simultaneously. The cached data allows for fast retrieval and, since the same data gets written in the permanent storage, we will have complete data consistency between the cache and the storage. Also, this scheme ensures that nothing will get lost in case of a crash, power failure, or other system disruptions.

Although, write-through minimizes the risk of data loss, since every write operation must be done twice before returning success to the client, this scheme has the disadvantage of higher latency for write operations.

2. Write-around cache: This technique is similar to write-through cache, but data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write operations that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a "cache miss" and must be read from slower back-end storage and experience higher latency.

3. Write-back cache: Under this scheme, data is written to cache alone, and completion is immediately confirmed to the client. The write to the permanent storage is done after specified intervals or under certain conditions. This results in low-latency and high-throughput for write-intensive applications; however, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

13.2. Cache eviction policies

1. First In First Out (FIFO): The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.

2. Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.

3. Least Recently Used (LRU): Discards the least recently used items first.

4. Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first.

5. Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first.

6. Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.

14. Data Partitioning

14.1. Partitioning Criteria

1. Key or Hash-based partitioning

Under this scheme, we apply a hash function to some key attributes of the entity we are storing; that yields the partition number. For example, if we have 100 DB servers and our ID is a numeric value that gets incremented by one each time a new record is inserted. In this example, the hash function could

be 'ID % 100', which will give us the server number where we can store/read that record. This approach should ensure a uniform allocation of data among servers. The fundamental problem with this approach is that it effectively fixes the total number of DB servers, since adding new servers means changing the hash function which would require redistribution of data and downtime for the service. A workaround for this problem is to use Consistent Hashing.

2. List partitioning

In this scheme, each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there. For example, we can decide all users living in Iceland, Norway, Sweden, Finland, or Denmark will be stored in a partition for the Nordic countries.

3. Round-robin partitioning

This is a very simple strategy that ensures uniform data distribution. With 'n' partitions, the 'i' tuple is assigned to partition (i mod n).

4. Composite partitioning

Under this scheme, we combine any of the above partitioning schemes to devise a new scheme. For example, first applying a list partitioning scheme and then a hash based partitioning. Consistent hashing could be considered a composite of hash and list partitioning where the hash reduces the key space to a size that can be listed.

14.2. Common Problems of Data Partitioning

1. Joins and Denormalization

Performing joins on a database which is running on one server is straightforward, but once a database is partitioned and spread across multiple machines it is often not feasible to perform joins that span database partitions. Such joins will not be performance efficient since data has to be compiled from multiple servers. A common workaround for this problem is to denormalize the database so that queries that previously required joins can be performed from a single table. Of course, the service now has to deal with all the perils of denormalization such as data inconsistency.

2. Referential integrity

As we saw that performing a cross-partition query on a partitioned database is not feasible, similarly, trying to enforce data integrity constraints such as foreign keys in a partitioned database can be extremely difficult.

Most of RDBMS do not support foreign keys constraints across databases on different database servers. Which means that applications that require referential integrity on partitioned databases often have to enforce it in application code. Often in such cases, applications have to run regular SQL jobs to clean up dangling references.

3. Rebalancing

There could be many reasons we have to change our partitioning scheme:

- The data distribution is not uniform, e.g., there are a lot of places for a particular ZIP code that cannot fit into one database partition.
- There is a lot of load on a partition, e.g., there are too many requests being handled by the DB partition dedicated to user photos.

In such cases, either we have to create more DB partitions or have to rebalance existing partitions, which means the partitioning scheme changed and all existing data moved to new locations. Doing this without incurring downtime is extremely difficult. Using a scheme like directory based partitioning does make rebalancing a more palatable experience at the cost of increasing the complexity of the system and creating a new single point of failure (i.e. the lookup service/database).

15. Proxy Server

15.1. Open Proxy

An open proxy is a proxy server that is accessible by any Internet user. Generally, a proxy server only allows users within a network group (a closed proxy) to store and forward Internet services such as DNS or web pages to reduce and control the bandwidth used by the group. With an open proxy, however, any user on the Internet is able to use this forwarding service. There are 2 famous open proxy types:

1. Anonymous Proxy. Reveals its identity as a server but doesn't disclose the initial IP address. Though this proxy server can be discovered easily, it can be beneficial for some users as it hides their IP address.
2. Transparent Proxy: This proxy server again identifies itself, and with the support of HTTP headers, the first IP address can be viewed. The main benefit of using this sort of server is its ability to cache the websites.

15.2. Reverse Proxy

A reverse proxy retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client, appearing as if they originated from the proxy server itself.

16. SQL & NoSQL

16.1. SQL

Relational databases store data in rows and columns. Each row contains all the information about one entity and each column contains all the separate data points. Some of the most popular relational databases are MySQL, Oracle, MS SQL Server, SQLite, Postgres, and MariaDB.

16.2. NoSQL

1. **Key-Value Stores**: Data is stored in an array of key-value pairs. The 'key' is an attribute name which is linked to a 'value'. Well-known key-value stores include Redis, Voldemort, and Dynamo.
2. **Document Databases**: In these databases, data is stored in documents (instead of rows and columns in a table) and these documents are grouped together in collections. Each document can have an entirely different structure. Document databases include the CouchDB and MongoDB.
3. **Wide-Column Databases**: Instead of 'tables,' in columnar databases we have column families, which are containers for rows. Unlike relational databases, we don't need to know all the columns up front and each row doesn't have to have the same number of columns. Columnar databases are best suited for analyzing large datasets - big names include Cassandra and HBase.
4. **Graph Databases**: These databases are used to store data whose relations are best represented in a graph. Data is saved in graph structures with nodes (entities), properties (information about the entities), and lines (connections between the entities). Examples of graph database include Neo4J and InfiniteGraph.

16.3. Differences: SQL vs. NoSQL

1. Storage: SQL stores data in tables where each row represents an entity and each column represents a data point about that entity; for example, if

we are storing a car entity in a table, different columns could be 'Color', 'Make', 'Model', and so on.

NoSQL databases have different data storage models. The main ones are key-value, document, graph, and columnar. We will discuss differences between these databases below.

2. Schema: In SQL, each record conforms to a fixed schema, meaning the columns must be decided and chosen before data entry and each row must have data for each column. The schema can be altered later, but it involves modifying the whole database and going offline.

In NoSQL, schemas are dynamic. Columns can be added on the fly and each 'row' (or equivalent) doesn't have to contain data for each 'column.'

3. Querying: SQL databases use SQL (structured query language) for defining and manipulating the data, which is very powerful. In a NoSQL database, queries are focused on a collection of documents. Sometimes it is also called UnQL (Unstructured Query Language). Different databases have different syntax for using UnQL.

4. Scalability: In most common situations, SQL databases are vertically scalable, i.e., by increasing the horsepower (higher Memory, CPU, etc.) of the hardware, which can get very expensive. It is possible to scale a relational database across multiple servers, but this is a challenging and time-consuming process.

On the other hand, NoSQL databases are horizontally scalable, meaning we can add more servers easily in our NoSQL database infrastructure to handle a lot of traffic. Any cheap commodity hardware or cloud instances can host NoSQL databases, thus making it a lot more cost-effective than vertical scaling. A lot of NoSQL technologies also distribute data across servers automatically.

5. Reliability or ACID Compliancy (Atomicity, Consistency, Isolation, Durability): The vast majority of relational databases are ACID compliant. So, when it comes to data reliability and safe guarantee of performing transactions, SQL databases are still the better bet.

6. Most of the NoSQL solutions sacrifice ACID compliance for performance and scalability.

16.4. Choose which?

16.4.1. SQL

1. We need to ensure ACID compliance. ACID compliance reduces anomalies and protects the integrity of your database by prescribing exactly how transactions interact with the database. Generally, NoSQL databases sacrifice ACID compliance for scalability and processing speed, but for many e-commerce and financial applications, an ACID-compliant database remains the preferred option.

2. Your data is structured and unchanging. If your business is not experiencing massive growth that would require more servers and if you're only working with data that is consistent, then there may be no reason to use a system designed to support a variety of data types and high traffic volume.

16.4.2. NoSQL

When all the other components of our application are fast and seamless, NoSQL databases prevent data from being the bottleneck. Big data is contributing to a large success for NoSQL databases, mainly because it handles data differently than the traditional relational databases. A few popular examples of NoSQL databases are MongoDB, CouchDB, Cassandra, and HBase.

1. Storing large volumes of data that often have little to no structure. A NoSQL database sets no limits on the types of data we can store together and allows us to add new types as the need changes. With document-based databases, you can store data in one place without having to define what "types" of data those are in advance.

2. Making the most of cloud computing and storage. Cloud-based storage is an excellent cost-saving solution but requires data to be easily spread across multiple servers to scale up. Using commodity (affordable, smaller) hardware on-site or in the cloud saves you the hassle of additional software and NoSQL databases like Cassandra are designed to be scaled across multiple data centers out of the box, without a lot of headaches.

3. Rapid development. NoSQL is extremely useful for rapid development as it doesn't need to be prepped ahead of time. If you're working on quick iterations of your system which require making frequent updates to the data structure without a lot of downtime between versions, a relational database will slow you down.

17. CAP Theorem

CAP theorem states that it is impossible for a distributed software system to simultaneously provide more than two out of three of the following guarantees (CAP): Consistency, Availability, and Partition tolerance. When we design a distributed system, trading off among CAP is almost the first thing we want to consider. CAP theorem says while designing a distributed system, we can pick only two of the following three options:

- Consistency: All nodes see the same data at the same time. Consistency is achieved by updating several nodes before allowing further reads.
- Availability: Every request gets a response on success/failure. Availability is achieved by replicating the data across different servers.
- Partition tolerance: The system continues to work despite message loss or partial failure. A partition-tolerant system can sustain any amount of network failure that doesn't result in a failure of the entire network. Data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.

We cannot build a general data store that is continually available, sequentially consistent, and tolerant to any partition failures. We can only build a system that has any two of these three properties. Because, to be consistent, all nodes should see the same set of updates in the same order. But if the network loses a partition, updates in one partition might not make it to the other partitions before a client reads from the out-of-date partition after having read from the up-to-date one. The only thing that can be done to cope with this possibility is to stop serving requests from the out-of-date partition, but then the service is no longer 100% available.

18. Consistent Hashing

18.1. Improve caching system

Distributed Hash Table (DHT) is one of the fundamental components used in distributed scalable systems. Hash Tables need a key, a value, and a hash function where hash function maps the key to a location where the value is stored.

index = hash_function(key)

Suppose we are designing a distributed caching system. Given 'n' cache servers, an intuitive hash function would be 'key % n'. It is simple and commonly used. But it has two major drawbacks:

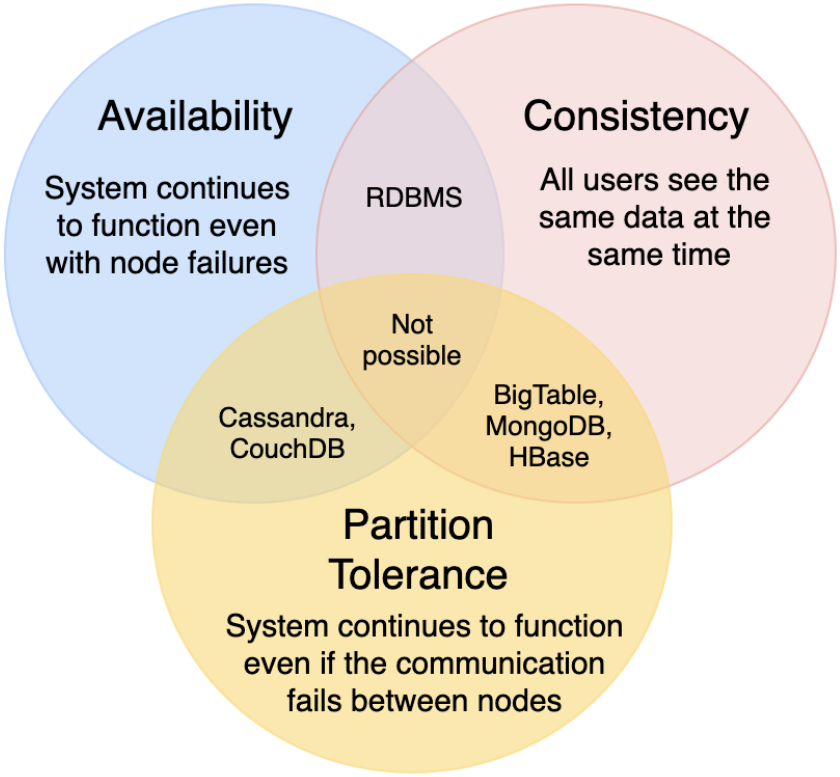
Availability
System continues to function even with node failures
Consistency
All users see the same data at the same time
RDBMS
Not possible
Cassandra, CouchDB
BigTable, MongoDB, HBase
Partition Tolerance
System continues to function even if the communication fails between nodes

Figure 1: CAP

- It is NOT horizontally scalable. Whenever a new cache host is added to the system, all existing mappings are broken. It will be a pain point in maintenance if the caching system contains lots of data. Practically, it becomes difficult to schedule a downtime to update all caching mappings.
- It may NOT be load balanced, especially for non-uniformly distributed data. In practice, it can be easily assumed that the data will not be distributed uniformly. For the caching system, it translates into some caches becoming hot and saturated while the others idle and are almost empty.

18.2. Consistent Hashing

Consistent hashing is a very useful strategy for distributed caching systems and DHTs. It allows us to distribute data across a cluster in such a way that will minimize reorganization when nodes are added or removed. Hence, the caching system will be easier to scale up or scale down.

In Consistent Hashing, when the hash table is resized (e.g. a new cache host is added to the system), only 'k/n' keys need to be remapped where 'k' is the total number of keys and 'n' is the total number of servers. Recall that in a caching system using the 'mod' as the hash function, all keys need to be remapped.

In Consistent Hashing, objects are mapped to the same host if possible. When a host is removed from the system, the objects on that host are shared by other hosts; when a new host is added, it takes its share from a few hosts without touching other's shares.

18.3. Algorithm

As a typical hash function, consistent hashing maps a key to an integer. Suppose the output of the hash function is in the range of [0, 256]. Imagine that the integers in the range are placed on a ring such that the values are wrapped around.

1. Given a list of cache servers, hash them to integers in the range.
2. To map a key to a server,

- Hash it to a single integer.
- Move clockwise on the ring until finding the first cache it encounters.
- That cache is the one that contains the key. See animation below as an example: key1 maps to cache A; key2 maps to cache C.

To add a new server, say D, keys that were originally residing at C will be split. Some of them will be shifted to D, while other keys will not be touched.

To remove a cache or, if a cache fails, say A, all keys that were originally mapped to A will fall into B, and only those keys need to be moved to B; other keys will not be affected.

For load balancing, as we discussed in the beginning, the real data is essentially randomly distributed and thus may not be uniform. It may make the keys on caches unbalanced.

To handle this issue, we add "virtual replicas" for caches. Instead of mapping each cache to a single point on the ring, we map it to multiple points on the ring, i.e. replicas. This way, each cache is associated with multiple portions of the ring.

If the hash function "mixes well," as the number of replicas increases, the keys will be more balanced.