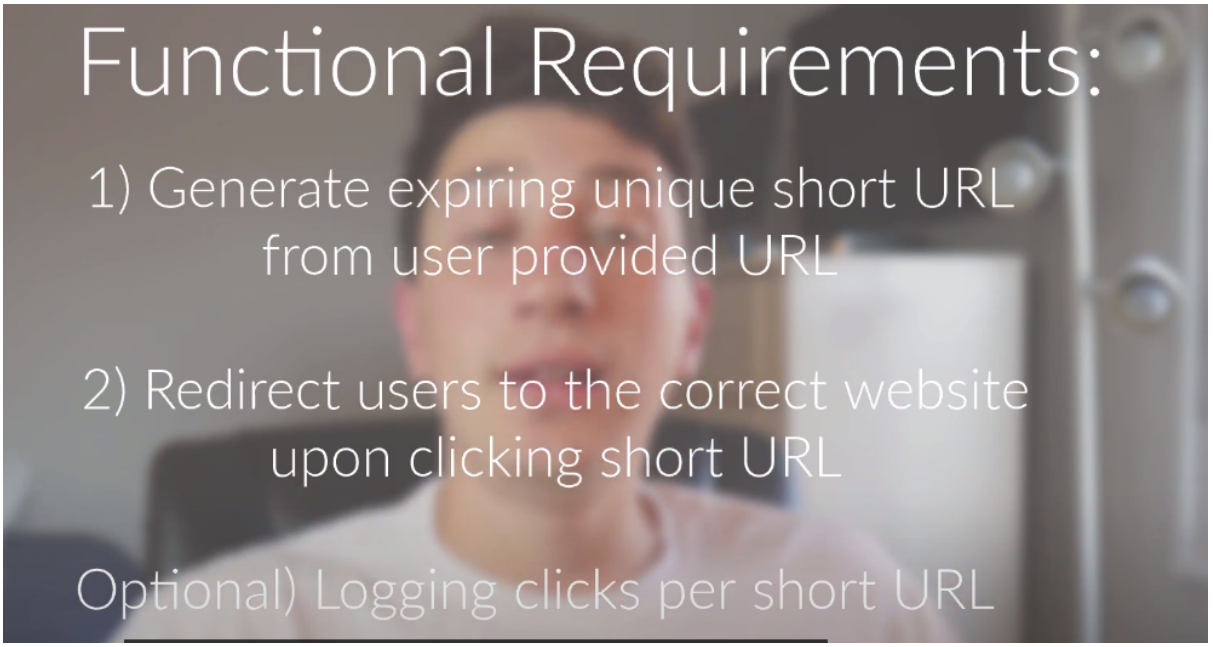
Tiny url



Capacity estimation

Functional Requirements for TinyURL:

> Generate a shorter unique URL for a given original URL.
> Ensure low latency and high availability for generating unique URLs.
> Redirect users to the proper place when they access short URLs.
> Implement link expiration for managing available short links.
> Incorporate analytics to track the number of clicks on short URLs.

Functional Requirements for Pastebin:

Allow users to upload and store text data.
Generate a shortened URL for accessing the uploaded text.
Implement link expiration for managing the lifespan of stored text.
Provide the ability to access the uploaded text via the URL.

Capacity Estimations for TinyURL: Assumptions:

- 500 million URL shortenings generated per month.
- 100 to 1 read to write ratio (more reads than writes).

Calculations:

- Approx. 200 writes per second (500 million / 30 days / 24 hours / 3600 seconds).
- Approx. 20,000 reads per second (200 writes * 100 read-to-write ratio).
- Storage requirement: 15 terabytes over five years (500 bytes per URL * 500 million * 12 months * 5 years).
- Caching: Assuming 20% of keys are responsible for 80% of traffic, caching 100 million URLs, which is approximately 50 gigabytes of cache per month.

Capacity Estimations for Pastebin:

- Similar calculations would apply, but the storage requirement depends on the average size of the pastes (uploaded text).
- It's important to consider variations in paste size, from a few lines of text to potentially gigabytes.

Segment 1: Introduction and Problem Statement

The problem is to create a system that generates unique short URLs for original URLs (or text for Pastebin) and redirects users to the original content.
Two main approaches are discussed for generating short URLs: hashing and key generation service.
The goal is to ensure that short URLs are unique and not conflicting.
Segment 2: Short URL Size and Combinations

The size of short URLs is determined by the characters used (a-z and 0-9) and the length of the URL (e.g., 6 or 8 characters).
Calculation of the total possible combinations based on the character set and length.

Let's imagine that our short URL keys, so like tinyurl.com short URL key, can contain let's say 'a' through 'z' all lowercase and '0' through '9'.

So that's going to be 26 letters and 10 numbers, so 36 characters in total ,say a length of 6. Then you could have 36 to the sixth total combinations

Segment 3: Hashing Approach

Hashing involves mapping one value to another.
Hash algorithms like SHA or MD5 can be used to generate short URLs.
Challenges include truncating the hash result to fit the desired length and potential collisions.
Collisions occur when two different URLs produce the same hash, leading to conflicts.
Segment 4: Key Generation Service Approach

Key generation service stores all possible keys in the database.
When a user requests a short URL, the service provides an unused key.
Challenges involve checking the database for unused keys and handling potential race conditions.
Locking mechanisms and atomic operations can prevent key allocation conflicts.atomic operation like a comparison set or a lightweight transaction where we are basically putting a lock on an individual key row.
two separate tables could be there, one of unused keys and one of used keys.

Segment 5: Replication and Consistency

Multi-leader replication (e.g., Cassandra) might cause conflicts and data loss due to last-write-wins strategy.
Single-leader replication is suggested to avoid conflicts,
why- it would be really useful to continue to use a NoSQL architecture because, again, we don't need all of these abstractions that SQL puts on top of our data,
which slows things down quite a bit in a distributed setting.
I'm thinking that any type of NoSQL architecture using an LSM tree-based storage engine but still using single leader replication
where you have the ability to do some sort of lightweight transaction on a single partition is probably the best way to actually make sure that we're avoiding conflicts on a single URL.
Possibility of using databases like MongoDB or Riak for single-leader replication(Riak, even though it is a leaderless replication type of database,
does have some automatic conflict resolution logic which might make it so that we don't just clobber conflicting short URLs.
)

Segment 6: Partitioning and Load Balancing

Partitioning keys is important for even data distribution. If we are just partitioning by the actual short key itself, those are already hashes, so there should be a relatively even distribution of keys per machine.

This can be done where we go ahead and split up the unused keys

over a bunch of tables and then use some sort of load balancer to just go ahead and map every single server request to a different chunk of the database. So we can get a key from one of those
Caching helps prevent overwhelming the database, with eviction policies like least recently used (LRU).

Load balancing strategies like round-robin or consistent hashing can distribute requests among servers.
in order to avoid having a single point of failure, we can have it in active-passive configuration more or less, meaning that we have one load balancer which communicates with a second load balancer sitting in standby, waiting to go ahead and become the main load balancer if the first one stops sending it heartbeats.

ZooKeeper can help manage partitioning information and track database status.
Segment 7: Design for Pastebin

A modification of the system for Pastebin is discussed.
Content can be stored in S3, and popular pastes can be cached using a content delivery network (CDN).

Diagram of the complete system architecture

Load balancers
Microservices for key creation and fetching(due to read write ratio: 100:1)
MongoDB for single-leader replication

ZooKeeper for coordination
Cache for hot links
S3 for content storage (Pastebin)
CDN for caching popular pastes (Pastebin)

So, the client, after making either a create URL or a fetch URL request, is going to hit our load balancer, which in order to avoid having a single point of failure, we can have it in active-passive configuration more or less, meaning that we have one load balancer which communicates with a second load balancer sitting in standby, waiting to go ahead and become the main load balancer if the first one stops sending it heartbeats.

Then I've split the service into two microservices, one for creating the key, which as you can see, we have our horizontally scaled key creation servers which, upon looking to create an actual key for a given original URL, we'll first go ahead and use our either key generation logic or our hashing logic to go ahead and hit our single leader replicated database. For this one, I've just chosen to use MongoDB because MongoDB supports single leader replication out of the box and it's pretty popular.

After going ahead and hitting our database, what's going to go ahead and happen is we're going to replicate the result of that key creation to our replicas. And then upon actually trying to have another user fetch the result of a tiny URL, we're going to first check if that link is in the key cache, aka if it's one of the hot links. And if not, we're going to go ahead and access it from the database or one of the replicas. If it is in the key cache, we'll just go ahead and return that instantly and we'll save a lot of load from our database.

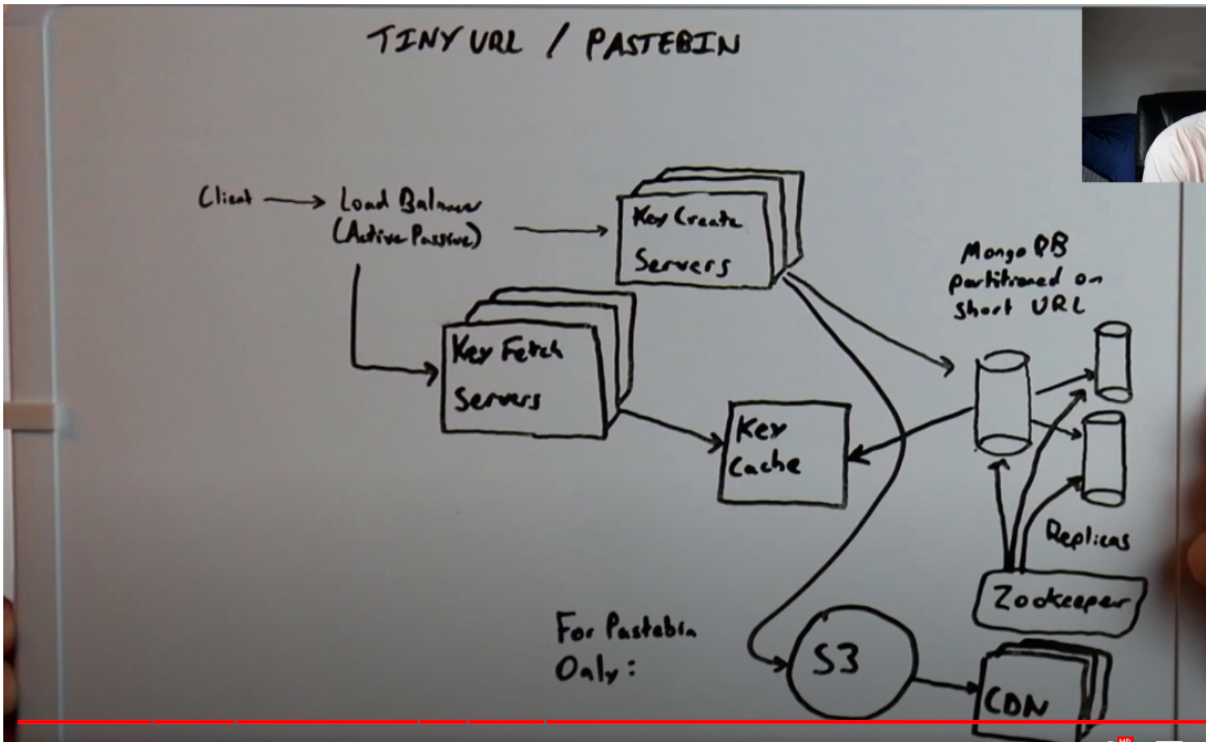
Additionally, in order to support the actual partitioning information of the replicas, we will have a ZooKeeper instance, which is going to periodically receive heartbeats from each of the replicas to make sure they're up and also all of the partition masters in the cluster in order to make sure they're up. So, the ZooKeeper is just going to be our coordination service in order to kind of deal with the status of all of our databases.

And then in addition to that, even though I didn't draw it out here, we should probably have a second set of load balancers between basically the fetching servers and the actual databases themselves, so that the basically that second load balancer is going to determine which replica to go ahead and read from in the event that we're going to actually make a read that is a cache miss and pulls from the database.

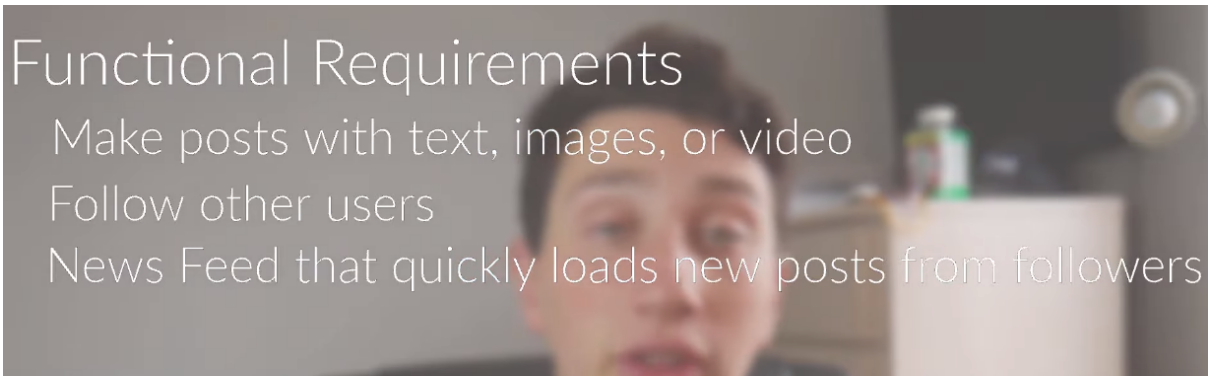
Additionally, on cache misses, we can choose to go ahead and load that database into our cache using our LRU eviction policy in order to go ahead and pull data from the database. Then finally, if we're talking about Pastebin, because like I said, Pastebin is just a small modification to TinyURL, in addition to actually having our database, when we create one of our short URLs corresponding to a paste, what we must first do before actually updating our database full of short URLs is putting the paste data in S3. And then for particularly popular pastes, we can go ahead and load those into a Content Delivery Network (CDN), because that is basically a cache that is distributed all over the globe, such that we can get content to our users really quickly.

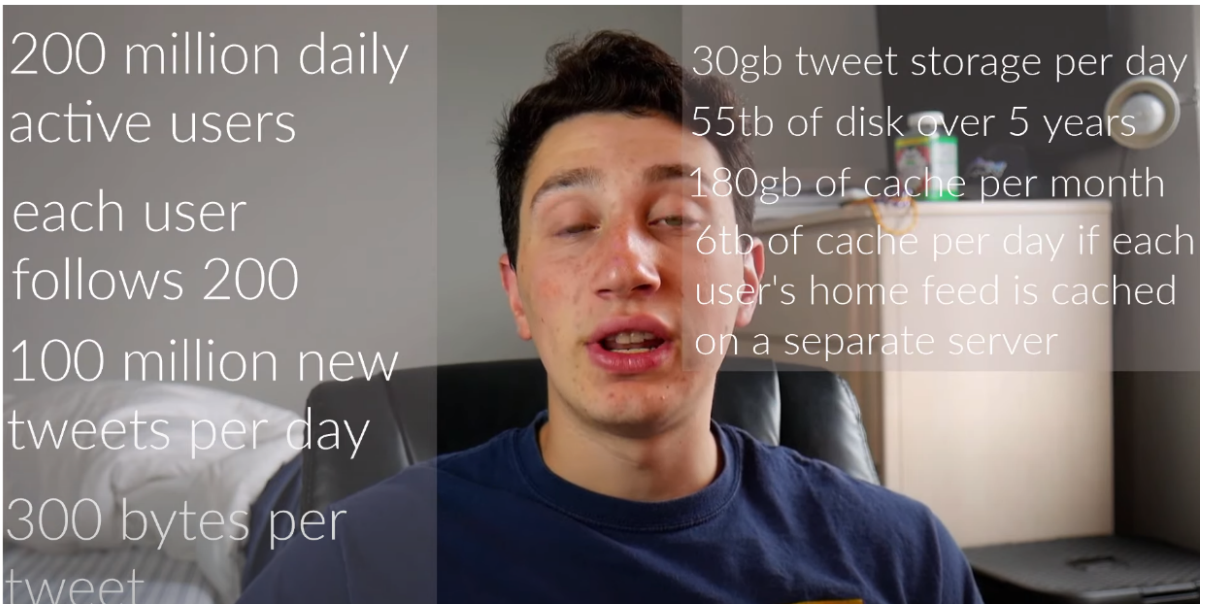
CDNs similarly to caches also have an expiration time for a bunch of the content in them and, furthermore, because they're distributed globally, we can do this kind of smart partitioning strategy, such that say only pastes in English will go to the American region. And then we can kind of distribute our pastes based on language in order to put them in one region or another.

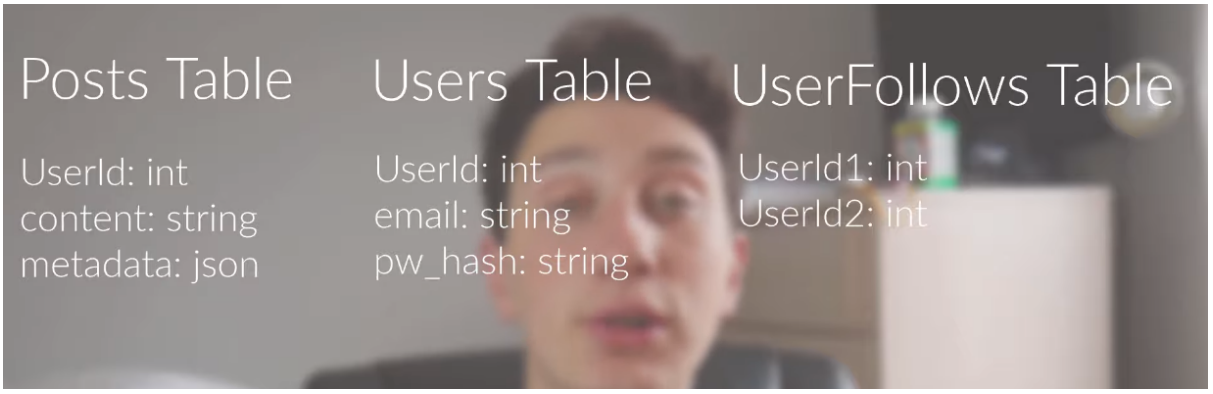
Okay, so that's more or less the overall design.



Twitter/facebook/Instagram news feed







Dropbox/Google Drive Design Deep Dive with Google SWE!

Estimation
Assuming an average file size of 100 kilobytes (KB).
Calculation: 100 KB * 100 billion files = 10 petabytes (PB) of storage needed.
Storage Solution:

Given the massive data storage requirements, the system would likely need to rely on an established storage solution provider like Amazon due to their vast resources and capabilities.
Dropbox initially used Amazon S3 but eventually built their own storage solution.
Real-time Changes and Connections:

The system anticipates around a million active connections per minute.
Implementing real-time change notifications could involve using techniques like web hooks.

Database Considerations:

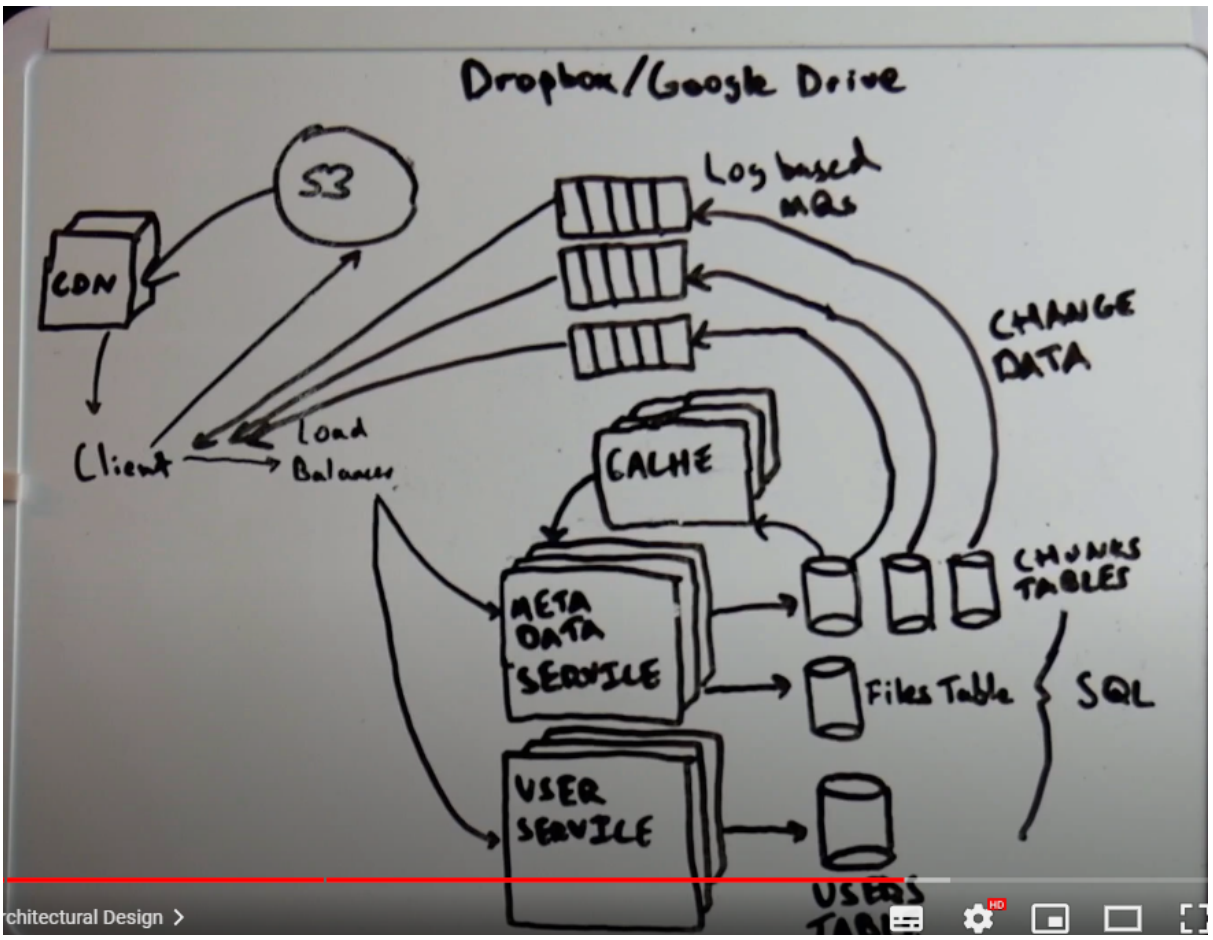
The primary purpose of the database is to store metadata for each file.
Due to the potential for large 1 GB files, it's logical to split files into chunks. This allows for uploading only modified chunks instead of the entire file.
Hashes of chunks can help identify whether a chunk is already in the system.
The database needs to store information about file versions and associated chunks.
Transaction and Consistency:

The need for transactions arises from the possibility of interleaved changes by multiple users leading to corrupted data.
To avoid interleaved changes and maintain data consistency, a relational SQL database with ACID compliance is recommended.
Database Choice:

A relational SQL database is well-suited due to its support for transactions and data consistency.
While parts of the system might benefit from a NoSQL database like Cassandra, the focus on avoiding write conflicts makes a single leader replication SQL configuration more suitable.
Database Schema:

Tables required include Users (user_id, email, password hash), Files (file_id, name), File Users (file_id with user_id references), File Version Chunks (file_id, version, chunk hash), and Chunks (chunk hash, S3 URL).
A Chunks table should include a file_id field for partitioning purposes.
Partitioning:

Given the massive amount of metadata, sharding and partitioning are crucial.
Partitioning can be done based on the file_id field to ensure data locality. Hashing file_id allows for distributing data across shards while keeping related metadata together.



Kind of the central idea is all replication is going to be single leader because we want to maintain consistency. All partitioning is basically going to be partitioned by files so we get a bunch of data locality. We don't basically have to do cross partition joins, and if we want for a given file to find all the chunks of it, we can just look at one partition.

Files are split into small chunks to allow efficient changes. A few modified chunks are better than uploading all chunks (up to 250 potential) for revisions. Reduction in server load and network bandwidth is significant.

Aim is to minimize server load and maximize client-side logic. Clients serve as local caches for files or documents.

So, with this in mind, what I think should be done is more or less the following: The client can basically look at its old revision of a document and then look at the new changes that have been made. Then basically say to itself, "Alright, here are the basically hashes of all the chunks that we last downloaded from the database. Here are the new hashes of all the chunks. We can see that we only have to say upload these few chunks." Once it determines that the problem still isn't overly simple, I think simplifies the system is to just from the client directly upload those differing chunks to S3. Then go ahead and reach out to the actual metadata server and basically say, "If I have the newest version of the file, then go ahead and commit these changes."

Of course, it's possible that the client doesn't have the newest version of the file because someone else made a change while they were offline. In which case, what's probably going to have to happen is the client is going to have to download the newest version of the file, merge those changes somehow, and then try this all again. You may say to yourself, "Oh well, this is going to result in some redundant uploads to S3." And while that's technically true, I don't think this is a case that's going to be happening frequently enough that it's going to result in a ton of unused chunks in S3. But if it really got to that point and we were really bloating our data storage and causing ourselves a ton of money, in theory, we could write some sort of separate cleanup or background process that basically occasionally pulls the database, tries to check whether a given hash is ever in use in a document, and if not, purge it from S3.

The only thing that you're really doing is making a single database update to your metadata table, and it's not that complex. So, I personally would just keep this as a traditional HTTP POST request.

For Alerts:

What Grok does every single time a document is changed: Each client should have a response queue corresponding to it, and the server that changed the document is responsible for potentially sending messages to every single response queue where there's a response queue for every single client.

Jordan's approach: Instead of pushing every single change to response queues partitioned by user, where every single user has their own response queue and we have to do all these pushes, I think what we should do instead is go ahead and basically partition our response queues by file ID. The reasoning for this is that we know that based on the assumptions of our problem, pretty much all the users aren't going to have any more than 200 files. As a result of that, each user potentially only has to keep about 200-ish websockets open with those response queues or server-sent events or whatever it may be. But they only have to persist 200 connections, which is very doable.

==========

So, as far as actually going ahead and making any modification to a file goes, I'm assuming that a client is basically holding the files that they have on hand. In addition to holding those files, they have on hand, they're holding this kind of limited internal version of a metadata table where they're keeping track of all the chunks and hashes of the chunks that had previously made up the version of the file that they've since edited. So, anyways, basically what I'm assuming now is that when a client wants to go ahead and make a change, what it's going to do is first consult that kind of mini metadata table and say, "Okay, compared to this, what are the chunks of the way the file currently looks and what are the chunks now that I need to upload to S3?" It then goes ahead and uploads these files to S3, which as you can see, I have attached to a CDN. If certain files are very popular, a CDN can really speed up the access times of those chunks.

And so, anyways, after uploading kind of those chunks to our file storage, what we're going to go ahead and do is hit our load balancer, which we can, you know, have in a kind of redundant configuration to ensure it's not a single point of failure. The load balancer can basically either send you to a user service if you need to make changes to user information or the permissions of a file in a user, or more importantly, if we're actually updating a file, to the metadata service where the metadata service is basically going to say the following: It's going to take a look at the files table and the chunks table and say, "If you're trying to update this file and you are updating what was previously known as the most up-to-date version, then you're good. Let's go ahead and make a change to the chunks table to basically say we've constituted a new version of the file." Otherwise, we're going to basically respond back to the client saying, "Hey, you actually need to go ahead and redownload the newer version of this document and merge in those conflicts because you don't actually know what the newest version was." And as a result, we don't want to basically let write conflicts occur. So, those are the two possibilities there.

But assuming that the write was successful, you know, the chunks are uploaded into the metadata table, which, keep in mind, should certainly be an SQL transaction. That's the reason all these tables are in SQL. Not only does changing all these chunks for a version of the file have to be atomic, it's very important that it is serializable in the sense that we can't have two people trying to interleave the chunks for a given version of a file. That would corrupt the file and be very bad.

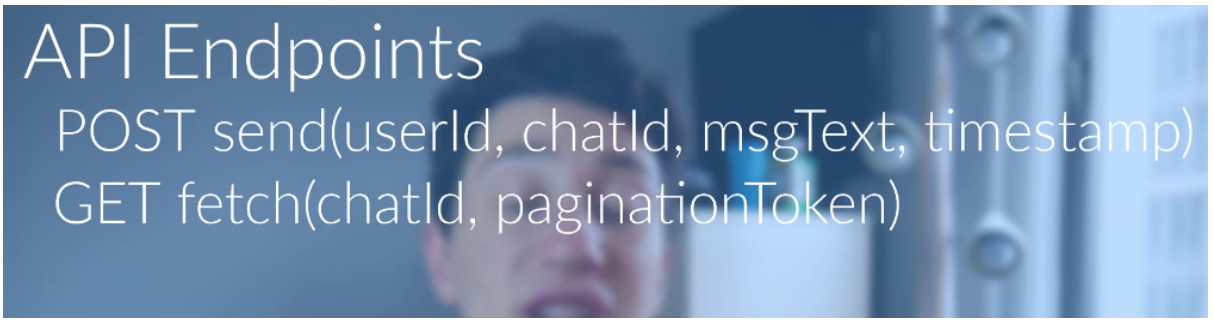
So, once those changes are in the chunks table, we're going to basically have a metadata cache to speed up reads for very commonly accessed files. And then additionally, to that, keep in mind that the chunks tables, like I mentioned earlier, are going to be partitioned on file ID. So, what that means is that when we pull this change data from every single chunk table, what is effectively going to be telling us are all the changes being made to a given set of files. And we can—you know, because we're partitioning on file ID or rather probably a hash of file ID—what we can go ahead and do from the beginning is have our client, since it's only probably has around 200 files in it, go ahead and subscribe to all of those message queues that are corresponding to the files that the client cares about. And so, by subscribing to basically probably up to 200 message queues, we can either use something like long polling or server-side events or webhooks—it doesn't really matter here—to go ahead and pull from these log-based message queues. And that way, we can ensure that a client connected to the internet is constantly receiving new updates for every single file in the system.

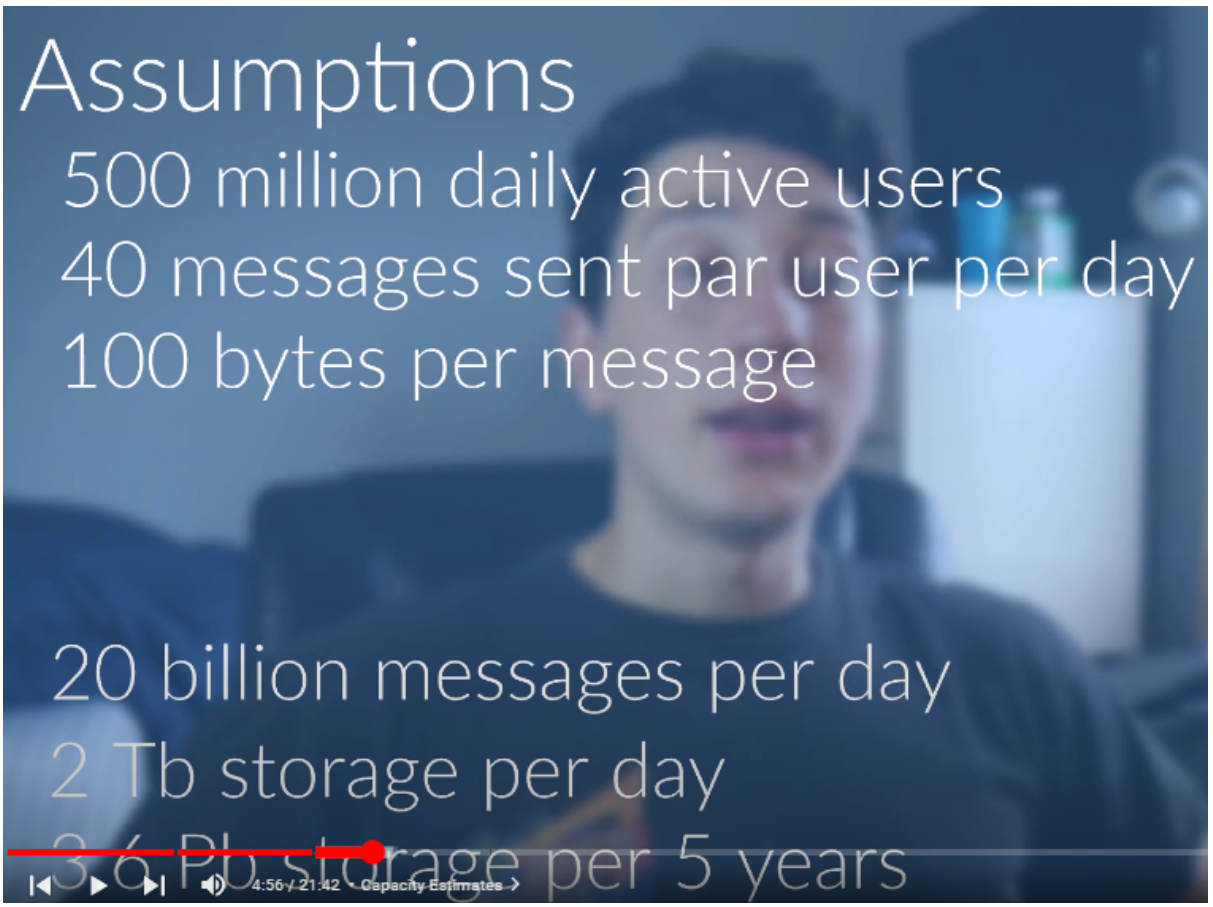
So, as you can see here, this is kind of one of the big differences that I have between my design and the design of others. They don't really specify the exact type of response queue that they're going to use, but at least as far as groking the system design interview goes, that response queue is partitioned by user ID, which I think is super, super dumb. Because for popular documents, you're going to have to potentially be sending that document to literally millions of queues at once, which is a massive bottleneck. So instead, I think it's better to just keep the change data from every single one of these partitions of the chunks tables in its own separate queue. The reason I would have them as a log-based message queue is that so many clients can potentially be pulling from each queue that we want the messages to be

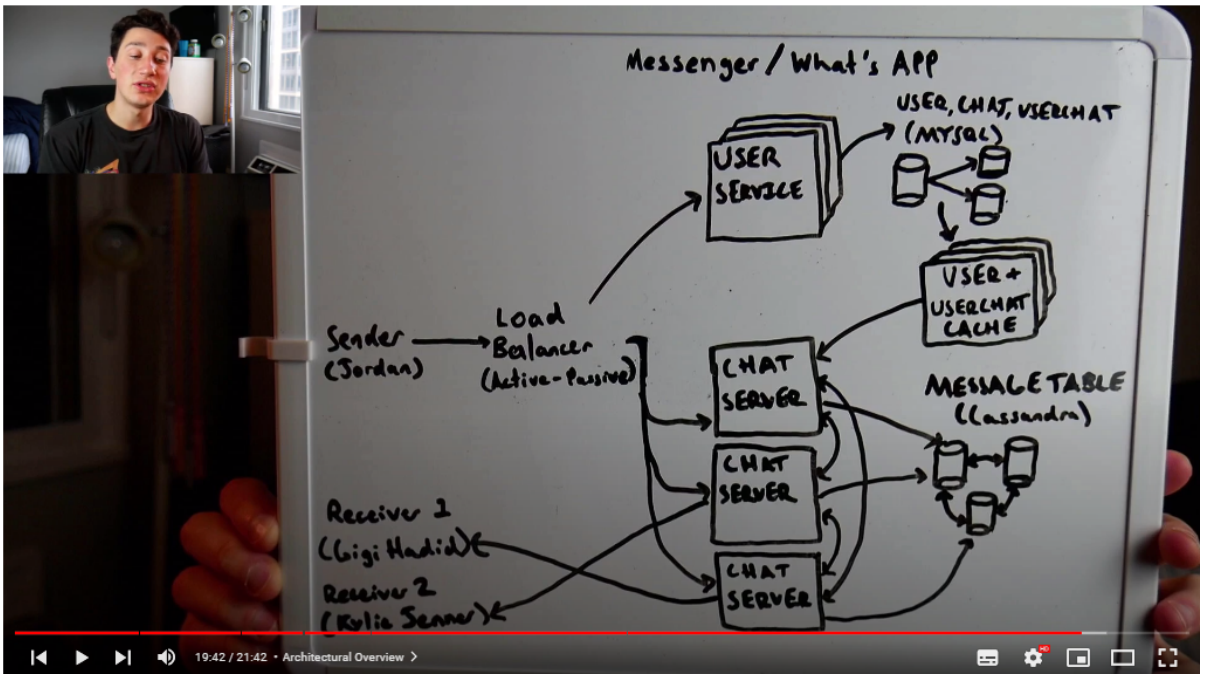
persistent. If one client were to get disconnected for some reason, you want it to be able to kind of pick up where it left off and start reading from those log-based message queues again. Obviously, these queues can be replicated as well to ensure fault tolerance. But yeah, overall, I think this is a pretty feasible system.

Like I've mentioned in the beginning of my video, kind of the point here is that this is one way to actually wire things together. But when you really want super high performance at scale like this, you have to start coming up with custom solutions. And I think the way that Dropbox ultimately did that was kind of coming up with their own file store to replace S3. But that's a little beyond our scope. We can talk about something like making file stores in a different video. But for now, this is the general high-level design that I hope to achieve, and I think it's actually pretty feasible. It's a little different than the grokking one in the sense that there's no request queue, and the response queues are partitioned quite differently. But ultimately, they share a lot of aspects in common, such as the chunking of files for a quicker upload to our file storage. Yeah, so anyways, guys, I hope that you enjoyed this video. I tried to actually differentiate myself a little bit here from the typical systems design interviews, and I made sure to watch a few before making this one of Dropbox and Google Drive so that I could try and, you know, cover what they didn't. For the most part, it seems that most of those videos are generally just clones of grokking the system's design interview, and so I purposely tend to try and go out of my way to differentiate myself and actually change my design up a little bit, and where I can, point out the flaws in those designs. Because I think a lot of them just don't go into enough depth, and if you really try and think about kind of the race conditions and some of the issues that they bring up, there are actually quite a few issues with them. And so, yeah, overall, this is kind of what I came up with, and I hope you guys like it. Have a good one.

Messanger/whatsup







For our fourth section of the video, we are going to be talking about database tables. The ones that I could come up with are:

- **Users Table:** This table will have fields like user id, email, and a hash of the user's password to be able to store there.
- **Chats Table:** This table will contain fields like chat id, chat name, and chat photo. It can also be relational

- **User Chats Table:** This table can have two fields: user id and chat id. Indexes can be applied for quick retrieval of chats for a given user and users for a given chat.

- **Messages Table:** I believe we should deviate from a traditional SQL database for the messages table. Instead, I suggest using a NoSQL database like Cassandra or HBase due to the write-heavy nature of sending messages. The LSM tree architecture of NoSQL databases suits this purpose. I think Cassandra could be more suitable. It uses leaderless replication, which works well for geographically distributed users. The messages will be written to an in-memory buffer first, and this architecture should handle the high write volume efficiently. For the partition key, chat id could be used. Fields like message timestamp and content of the message should also be included.

Regarding the choice between HBase and Cassandra, Grok recommends HBase due to its schema-less nature, but Cassandra also offers this advantage. Moreover, HBase being column-oriented might not provide significant benefits if we frequently need to fetch entire rows with metadata, content, and timestamps. Cassandra's leaderless replication scheme and the need for pushing messages to other group chat members further solidify its suitability for this use case.

As for tracking which clients or devices are receiving messages and mapping them to chat servers, a dedicated database instance like Redis might not be necessary. Instead, **we could use a load balancer with a consistent hashing algorithm to quickly direct messages to the appropriate servers.**

In the next section, we'll delve into the design details of the messaging system and explore how all these components fit together.

So, basically, the first consideration I want to get to is this: we know that we want to be pushing our messages in real time to clients as they come in, after, for example, I send my message to the server, and then I'm in a chat with someone else who wants to receive that

message in real time. So, what technology should we actually be using for pushing the messages? Well, I'm going to propose a few considerations.

The first thing is that, as we know, we have 500 million daily active users, and we can estimate that a server is capable of handling around 50,000 to 65,000 open connections at once. It is impossible for one server to handle all 500 million websockets at once. Instead, all those websocket connections will have to be distributed horizontally over a bunch of application servers. To do that, we should use a load balancer with a consistent hashing mechanism to route user ids to application servers.

Regarding web hooks, server-side events (SSE), or long polling, there's not a conclusive answer. Short polling (hitting the server every few seconds) is not suitable. Long polling has the issue of resending headers every time, increasing payload size. Websockets and SSE are persistent connections, which is good. Websockets are bidirectional, which might not be necessary. SSE will automatically re-establish a connection if it fails, unlike websockets, which might not.

When sending messages, one client wants to send a message. They make a POST request to a chat server, via the load balancer, which directs them to a chat server to post the message. Once the message reaches the chat server, it has to push the message to the database and to all the clients in the group chat.

These processes might be concurrent, so the question is whether to use a distributed transaction. However, for this case, it's not critical. Uploading the message to the database first and then sending it to clients might be a good choice to ensure messages are persisted before being broadcast.

Regarding message ordering among chat members, Grok suggests it might not be perfect, but it can be achieved using front-end sorting based on timestamps. Timestamps may not be perfectly aligned due to distributed settings, but the order among chat members will be consistent.

In terms of the diagram, it aligns with what's been discussed. The sender sends a message, which reaches the load balancer, and then a chat server. The chat server queries user chat information to identify recipients. The message is stored in the message table, then forwarded to receivers via consistent hashing and SSE.

Finally, I'd like to wrap up this section by pointing out that we've covered the key distributed systems concepts that are relevant to this problem. The diagram provides an overview of how the messaging process works in this architecture.

So, basically, the first consideration I want to get to is this: we know that we want to be pushing our messages in real time to clients as they come in after, you know, say I send my message to the server and then I'm in a chat with someone else, and they want to be able to receive that message in real time. So, what technology should we actually be using for pushing the messages? Well, I'm going to propose a few considerations.

The first thing is that, as we know, we have 500 million daily active users, and we can basically go ahead and estimate that a server is capable of handling around 50,000 to 65,000 open connections at once, right? Which means that if that's the case, it is impossible for one server to be handling, you know, 500 million web sockets at once. Instead, all of those websocket connections or whatever type of connection we end up using will have to be distributed horizontally over a bunch of application servers. So, in order to do that, we should probably be using a load balancer with some sort of consistent hashing mechanism in order to go ahead and route a given user id to a given application server.

For webhooks, server-side events (SSE), or long polling, there's not really a great way to do this right, such that all the users of a given chat are together on the same server. That would minimize the network latency that we would have to use. But that's why I made that remark earlier about potentially using a graph database for users. Because if we could somehow partition up our users database in a way that all related users are going to be next to one another on the same partition, then maybe we could load balance those websocket connections as well, such that all related users, all users who might be in chats together, are going to have their websockets on the same servers. But obviously, that's pretty complex and I think it's probably beyond the scope of this video. Generally speaking, I think we can just assume that our load balancer is going to use round-robin or something, and that for a given user, it could be connected to basically any application server to get its messages received from.

Going on from there, sure, we understand that basically every single user has to be load balanced to some sort of chat server to receive messages. However, the question is now, should we use websockets to do this, should we use long polling, or should we basically use server-sent events?

I wouldn't say there's a conclusive great answer. Obviously, you don't want to use short polling, which is basically hitting the server every five seconds and saying "give me something new." Long polling could work, but it comes at the issue of having to resend headers every single time, which is annoying because it means that every single message that is basically going to be sent to that device, it's going to have a bigger payload, which means that there's going to be more network latency. On the contrary, websockets and server-sent events are both kind of persistent connections, which is good. However, it means that if that chat server were to go down, we would have to go ahead and reload balance all of those clients to a different chat server for them to instantly re-establish that handshake. And what that may result in is a thundering herd problem, where the increased load of all these clients trying to create a handshake with one of our application servers may lead to a cascading failure and actually take it down.

So, there is an argument to be made here for using long polling as opposed to one of these slightly more complex technologies like websockets or server-sent events. As far as websockets or SSE goes, websockets are a bi-directional connection, which we don't necessarily need here, because even though clients are going to be sending messages, they can just do that over a typical POST endpoint. And additionally, the one cool thing about server-sent events is that they will actually re-establish a connection automatically if they fail, and websockets don't do that. So perhaps, with all these things in mind, I would stick maybe

between server-sent events or long polling as opposed to websockets, because I don't want to deal with the thundering herd problem. And even though that may happen with SSE, at least we know that if an SSE connection were to fail in the short term, it's going to be re-established on its own. And, again, it's just a one-directional communication, but you can make an argument here either way.

Okay, additionally, now that we've spoken about receiving messages a little bit, let's actually go through the process of how we might send them. So, obviously, one client such as myself is going to want to send a message. And in order to do so, I'm going to make some sort of POST request to one of the chat servers, which I'll just go ahead and hit the load balancer, and then it'll send me to a chat server to go ahead and post that message.

So, once the message reaches the chat server, it's going to have to go ahead and do two things. One of those two things is push the message to the database, and the other is that it's going to have to push the message to all of the clients in the other group chat. The way it would push the message to the other clients in the group chat is by just going and hitting the load balancer, saying, "Please send all of these messages basically to the proper chat server, so that it can send them via the websocket or the server-sent event or the long polling." And then, additionally, it also just has to go ahead and make a typical HTTP request to the database server to go ahead and add that message in there.

The one thing here is these are technically going to be concurrent processes, and what happens if one of them works and, say, the other one fails? Do we want a distributed transaction here? Probably not, because it's not that important. It's not like we're dealing with financial data or anything that both of them go through. But perhaps, you know, if you care about the case where all the clients receive the message but then the message isn't persisted in the database because, say, I don't know, there's an error with the database connection, it's kind of problematic that all the clients might receive this message, have it cached locally, and then the message isn't in the database. So maybe what you could do is upload the message to the database first, and then go ahead and, upon receiving the success message, send that out to all these different clients. But I think there is a consideration basically between whether you want to do the database and the client uploading concurrently or whether you want to basically wait for the message to be in the database before sending them out to the clients.

Additionally, another concern I saw that Grok and the systems design interview brought up, which I don't really think is a valid one, is ordering messages amongst all the members in the chat. They basically said that even if you were using timestamps for the messaging, not all of the participants of the chat would see the same order of the messages. And this isn't true. It just kind of depends on how you use your front end, because they were basically just saying, if I were to send a message and then receive a message from someone else, even if that received message had basically an earlier timestamp than the one I sent, I would still see the received message later, and I would just respond to that. Well, why not just use a front end that maps out the messages in the order of their timestamps, and then that way everyone sees all the messages in the same order? Obviously, though, timestamps aren't perfect. What timestamp do we actually use? Do we use the timestamp on the device that sent the message? Do we use the timestamp of the server that received the message? As we know, timestamps aren't perfect, and in any distributed setting, timestamps aren't going

to be perfectly aligned with one another. And so, we can't ensure that the ordering of the messages is correct with the actual time that I clicked the send button. But we can ensure that using timestamps, it's at least consistent amongst all the members of the chat. Like I said, it just kind of depends on how you map them out with your front end.

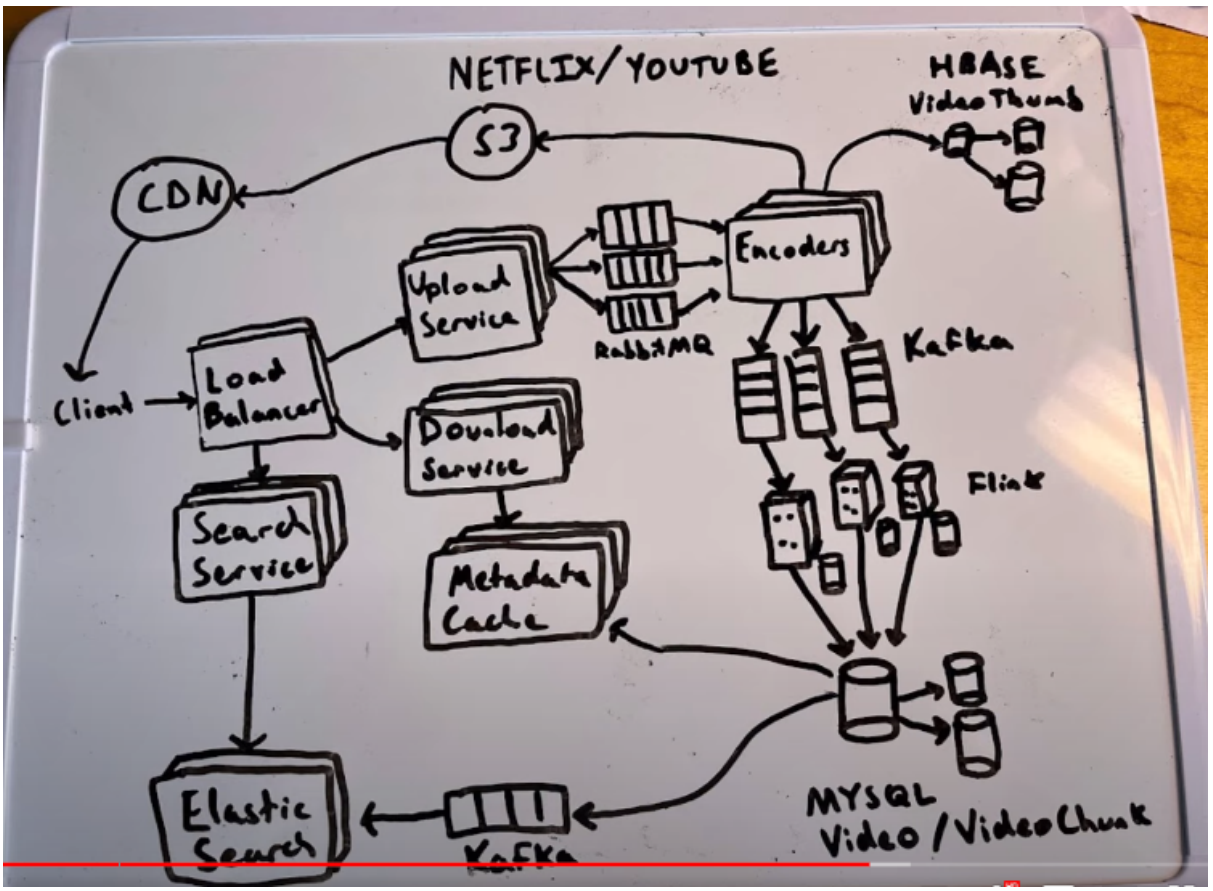
With that in mind, I think we generally speaking covered all of the deep distributed systems concepts that can come up in a problem like this. So, as a result, I would like to go into the diagram and walk us through that. Okay, let's actually go ahead and start talking about this diagram that I came up with. It's pretty in line with what I've been saying this whole video, and we can examine it.

So, let's imagine we have a sender, aka me, as far as this diagram goes. And the first thing I'm going to do upon sending a message is it's going to go ahead and reach the load balancer. As per usual, the load balancer is a single point of failure, and what I've gone ahead and done is said that we could put it in an active-passive configuration for redundancy. If we need to share some state between the two, we could, you know, pull some sort of high-availability stunt and use a shared coordination service between them, such as ZooKeeper. And the first thing that's going to do is it's going to go ahead and hit our chat server.

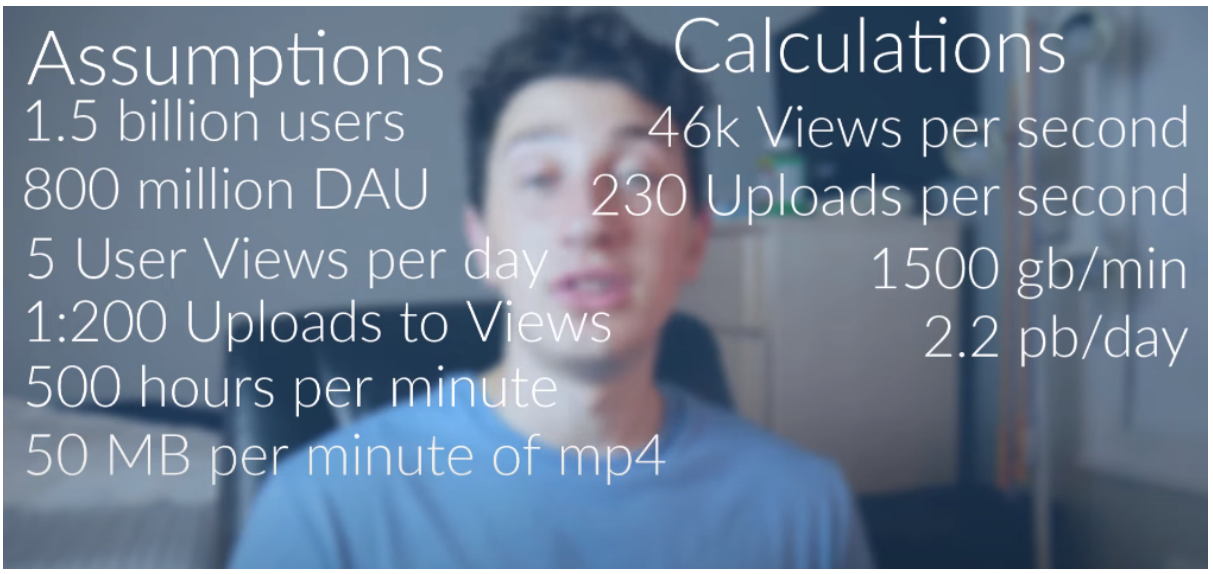
Now, the chat server upon receiving this message is going to see the chat ID for it and then go ahead and query our user chat table in order to figure out all of the user IDs that it's going to need to go ahead and forward this to. Hopefully, that data is going to be in the user chat cache, which is just going to be a replicated partition cache based on chat ID. But if it's not, we can go ahead and hit the table itself. Once the chat server has this information, it can go ahead using the load balancer to forward all of these messages to the various receivers. Right after it first puts the message in our message table, which is just going to be some sort of Cassandra instance, as you can see, it is leaderless and also going to be partitioned by the chat ID with a sort key of the timestamp, so that we can properly sort those messages on our client-facing front ends.

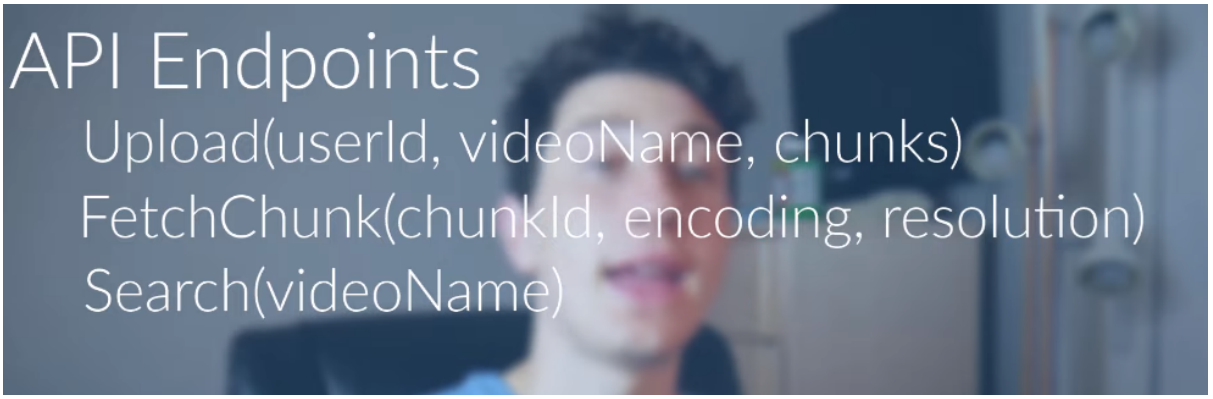
Then, finally, once that happens, those chat servers are going to communicate with each other using the consistent hashing pattern delegated by the load balancer. And then basically, via server-side events, deliver the messages properly.

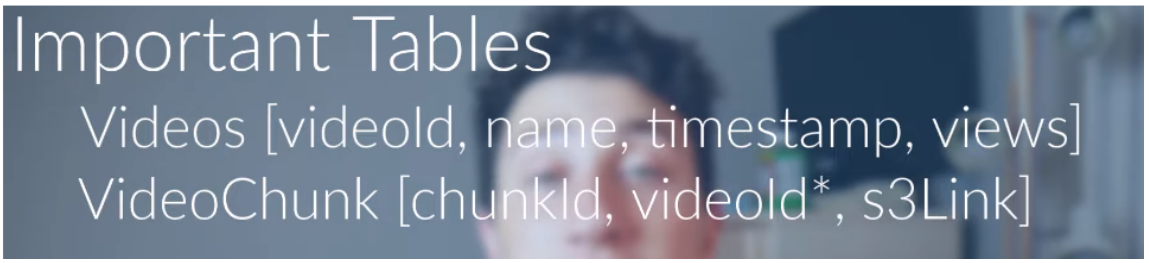
Netflix/youtube


NETFLIX/YOUTUBE
HBASE
Video Thumb
S3
CDN
Encoders
Upload Service
RabbitMQ
Kafka
Client
Load Balancer
Download Service
Flink
Search Service
Metadata Cache
Elastic Search
Kafka
MYSQL
Video / VideoChunk

Functional Requireme
Upload Videos
View Videos
Search Videos by Name







The first thing is that every single file that we're uploading, obviously we want to do the majority of our work on the write path, meaning that because there are going to be so many more video views than there are video uploads, the more work that we can do of pre-processing every single upload, obviously the less work that every single video view is going to have to do in order to basically render a certain type of encoding of the video, maybe get its thumbnails or anything like that. So, we have to do all these processing steps of every single video, and since we have all these processing steps, it would probably be easiest if we were to basically split that video file into chunks. Let's choose an arbitrary size, you know, in the few megabyte range, again, and we're going to basically do this so that we can parallelize all of that computation.

Every single one of these chunks is going to have to be encoded in a bunch of different formats and resolutions, and as a result, it would be better that, as opposed to having our

client upload the entire MP4 file at once to the server, that it could, in parallel, upload all these small MP4 chunks to a bunch of different servers, and then they could perform the processing. However, it's probably not sufficient to have this just be a traditional REST endpoint because that would insinuate that, you know, after, say, a few seconds, the client would expect to hear back from the server saying, "Hey, we uploaded everything, you're good to go." Obviously, these processes take a while because some of these videos are super long, and in addition to that, you know, when you're transferring a lot of data over the Internet like you would with an MP4 file, it's a slow process. So probably we're best off actually doing here is using something like a message queue.

Now, you might say, "Okay, well, should I be using a log-based message queue or an in-memory message queue?" And I personally would lean towards an in-memory message queue for at least this part of the process. The reasoning for that being, well, we don't really care about the order that the file chunks are uploaded. It just needs to be the case that they all eventually get uploaded. So, it'd be better to have something like a load-balanced in-memory message queue, which basically takes all the chunks of the video and gradually passes them to a bunch of different consumer nodes, which then do all of the necessary encoding or possible thumbnail extraction or anything like that.

Now, obviously, kind of the next step from here is, well, okay, fine, we have this original in-memory partitioned replicated queue. You know, that could be RabbitMQ, it could be Amazon SQS, whatever. The point is, let's say now that all these consumers have received the chunks, they've done the processing that needs to be done, how do they go ahead and basically get back to the client and say, "Hey, you know, we have done everything and now this video is entirely uploaded?" Well, personally, I think that it would be bad if they basically all individually put like a row in a database saying, "Oh yeah, my step is done." There are a couple of reasons for this. I think that the biggest one is that it's not atomic, right? So, it's possible that some of the chunks might succeed in being processed and some of the chunks might fail in being processed, and now these chunks are held in our database as video chunks, and the other ones are not. Now we have this incomplete picture of a YouTube video in our database.

So, what I personally think would be a cool way of doing this, or not necessarily a cool way, but a functional way, is you create basically another queue, and this time it would be log-based, so that we make sure we have replayability and durability in the actual messages themselves. And every single one of those processing nodes that processes a chunk and encodes it or does something like that, we'll then put an entry into a log-based queue, which can be partitioned by video ID. And that way, you know that all the entries for a given video are going to be in the same queue and then they can, in turn, be consumed by the same stateful consumer, and basically, so now we have all this data saying like "Chunk one processed," "Chunk two processed," "Chunk three processed," and it's all going through this one log-based queue to the same stateful consumer node, right? And that could be running software like Flink in order to keep track of that state.

So, now what's going to happen is that Flink consumer is going to keep track of this local state and say, "Okay, so for video ID, I don't know, 69, we know that, you know, I got one entry in the queue basically saying there are going to be, I don't know, 10 total chunks for this." And it can keep track of all the chunks that it's received. And so, once it receives word

or messages from that log-based queue of all 10 chunks, it can go ahead and upload all of the data to our metadata table, which is basically all the video chunks at once. I think personally it's important that this update is transactional, right? Because, like I said, you don't want it to be the case that only part of the chunks are uploaded and the other part are not. So, that's why personally I think that for basically all the tables in the scenario, SQL is actually a very acceptable solution. And if you're using SQL, it probably means that you're going to be doing something like single-leader replication. And in terms of partitioning, because obviously we have a ton of video data, I think that it's very sufficient to partition the video chunks table by just the video ID itself. You know, just make sure that all the chunks for a given video are on the same machine. And then for the video table, which is just like a video ID and a name, perhaps you partition that by something like a user ID of the person who posted it. So, you could quickly go to their wall and find all the videos they've posted in the past.

Anyways, my point is by having this stateful consumer node which is reading from this log-based queue and then ultimately using a transaction to update our metadata, we can ensure that we're never going to have basically a partial view of one of the videos and everything is going to be uploaded at once. In terms of the actual video chunk files themselves, those can obviously go into a flexible storage system like S3, which is very elastic and basically scales up to any level of demand that we could possibly need, which makes it very useful. I'm sure some of these websites do have custom storage solutions, but at the end of the day, they're all pretty similar implementations and all end up resembling some sort of file storage or block storage in one way or the other.

That being said, while I think it makes sense to actually have the MP4 files in something like S3, for the actual thumbnail files, Graping the System's Design makes a very good point, which is that it doesn't necessarily make sense to put them in something like S3, but rather some sort of NoSQL data store where you can get better data locality. They assume that each video has five thumbnails, and while I'm not necessarily sure that this is the case on YouTube, maybe it's more relevant on Netflix if you're going to be basically storing a bunch of static images for a given video. It's really useful to have a table in something like, I don't know, Cassandra or they use Bigtable, for example, because Bigtable has column-oriented storage.

So, then basically, we could have this table in Bigtable, which, keep in mind, is the same as HBase if you've watched my HBase video. And what you would do is basically have a table called "video thumbnails" where the partition key is video ID, the sort key is thumbnail number, and that way we can get this great data locality in storing five thumbnails together. So, we're not just making like five super random accesses to disk in something like S3, but rather we get this super great data locality as a result of column-oriented storage in Bigtable or HBase.

So, that's kind of the logic that they use there, which, truthfully, I agree with. As far as loading videos go, like I said, everything that we're doing, we're optimizing for the read path, which is, if I am creating a video, it's totally fine if it takes a couple of hours to upload. But if I'm viewing a video, it better get to me as fast as possible and it better render as fast as possible.

So, the general gist of this is we know our videos are split into chunks, so once we have all these chunk IDs that we need to be able to load, what you would then go and do is load the first chunk, let it get a little bit of the way through, then load the second chunk and load the third. And what this allows us to do is further parallelize our computation in the sense that we don't have to sequentially load an entire video file at once, but rather we can wait until one is loaded, then start getting a little bit of the other. And the point is we can start watching a video before the entire thing is loaded, so that's very, very helpful.

And in addition, of course, in reality, services like Netflix and YouTube are using tons of cache for the metadata and also tons of cache in the form of CDNs for their video files, which is kind of the bread and butter of how they really speed things up. They have massive cache networks and massive CDN networks to make sure that the data is as close to the end user as humanly possible.

So, with this all being said, let me go ahead and show off my diagram and try to basically wrap up what I just said so that it fits on just one whiteboard. Well, this is easily the most complex diagram yet that I've drawn up on this channel. Let's get into it because this one's gonna be fun.

Firstly, we are going to start out with our typical client, as one does, which hits the load balancer. I didn't really state it explicitly here, but we can run this with some sort of redundant configuration in order to make sure that it's not a single point of failure, e.g., active-active or active-passive configurations. Active-active is when you basically have two load balancers running, but then you run the risk that if one goes down, the other gets overloaded. Active-passive is when the passive one basically knows the state of the active one but just kind of sits there waiting for it to die. Either works, both have their advantages and disadvantages.

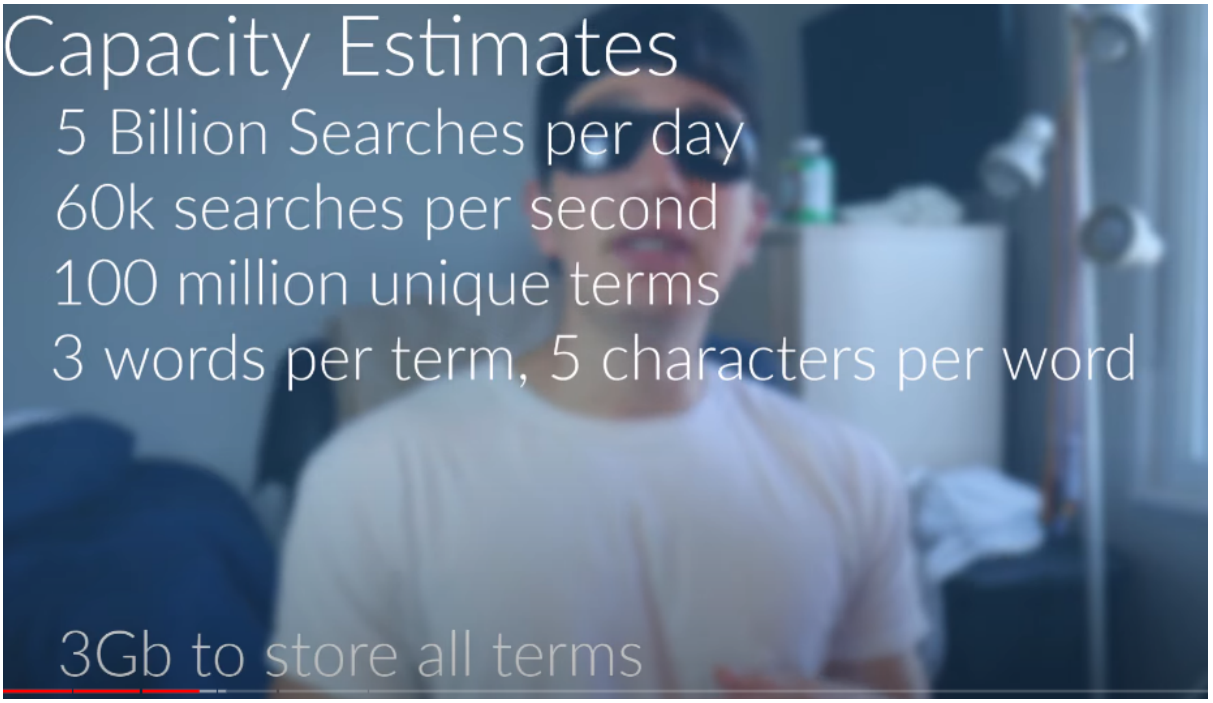
Anyways, as far as actually uploading a video goes, you're going to reach the upload service, which is then basically going to take that file or basically take all the chunks that it's received from the client and throw them all into RabbitMQ, which is a partitioned, replicated in-memory queue. No, it doesn't have to be RabbitMQ, but I think for the sake of just listing out technology so that you guys get more familiar with them, this is kind of a nice thing to do, to be able to list them like that. I think any in-memory queue here would work just fine, and you know, just justify your choice. This one's just a popular one.

So, anyways, once that basically chunk data goes into RabbitMQ, it can be load-balanced or basically fanned out into any one of a number of encoders, perhaps even multiple encoders if a chunk has to reach basically multiple levels of encoding or resolution or anything like that. So, it's going to hit the encoders, the encoders are going to do the work that they do. I personally don't know much about it, but you know, they do stuff, put that data into S3, and then from there, if they have any thumbnails, they can also go ahead and throw them into HBase. And then finally, what they're going to do is basically then take the result of the successful operation and put that in Kafka queues. Now, keep in mind that these Kafka queues are log-based, and the reasoning for this is you don't want to lose the fact that, say, any chunk was processed successfully or not successfully. It's very important that we don't lose the data here, and you know, ordering itself isn't overly important. So, maybe we actually could use a queue here, but the bigger thing for me was I wanted to ensure that the

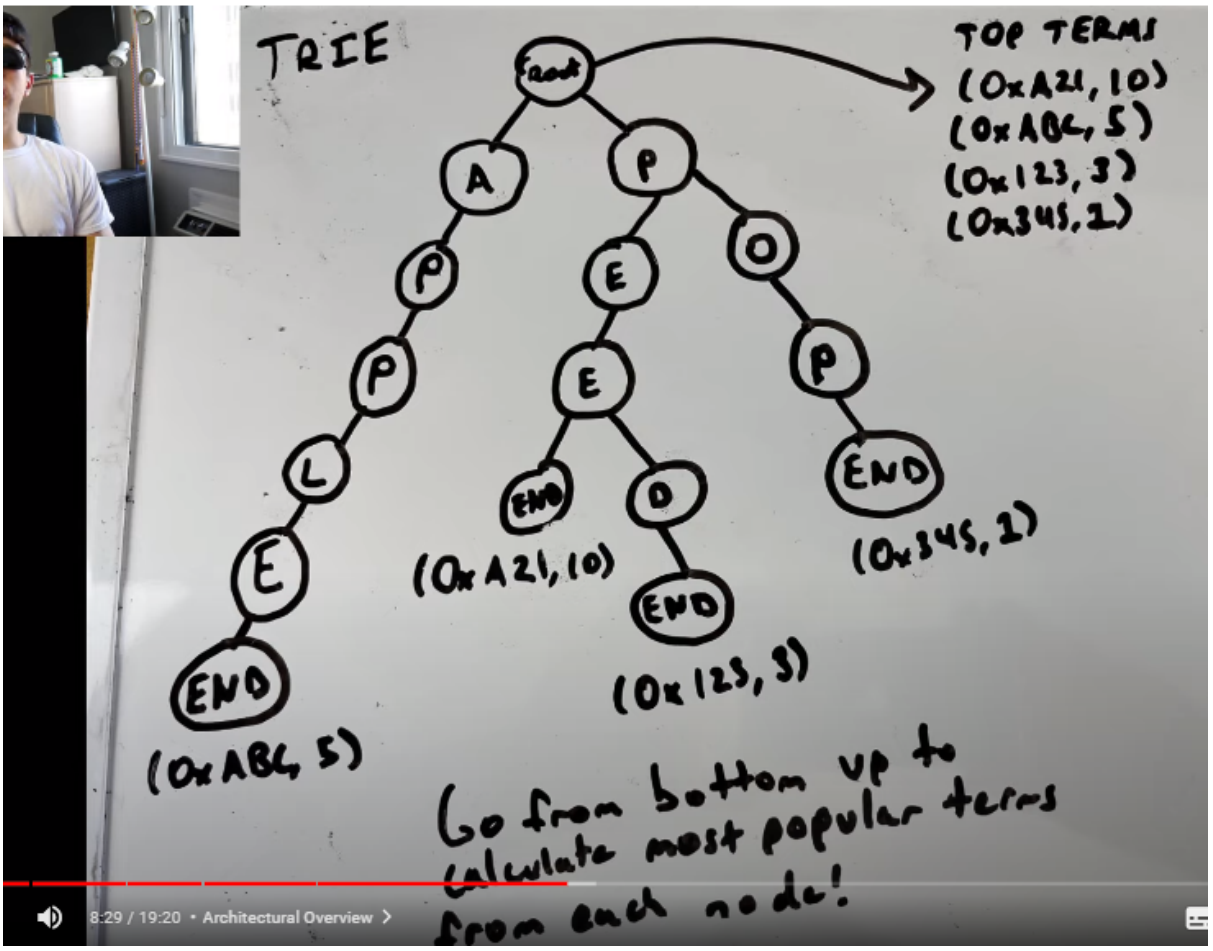
queue was going to be partitioned basically by video ID. So, for each message, you can have the actual chunk data, but then also a field for the video ID, and that way, you're basically going ahead and making sure that everything is going in the proper Kafka queue, which means that it's going to be handled by the same Flink consumer.

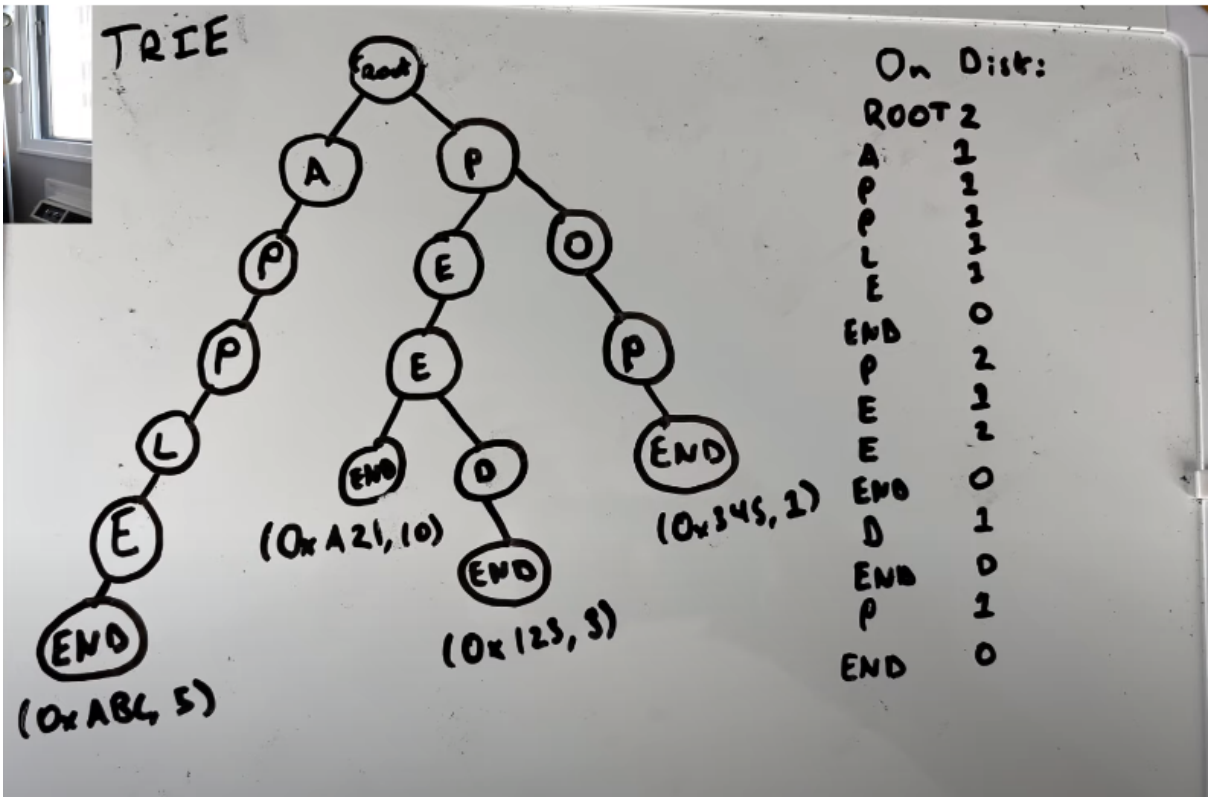
Now, what the Flink consumer is going to do is, like I mentioned earlier, it keeps track of basically all the chunks that it's seen for a given video ID. And then, once all of those chunks are present, and we can determine this by just actually putting the number of chunks for a given video ID in each message, maybe there are better ways than doing it than that, such as sending a dedicated message itself, but the point is, once Flink or the Flink consumer says, "Okay, I've seen all the chunks that I need to see," it can go ahead and upload all of the data to our metadata table, which is basically all the video chunks at once. I think personally it's important that this update is transactional, right? Because, like I said, you don't want it to be the case that only part of the chunks are uploaded and the other part are not. So, that's why personally I think that for basically all the tables in the scenario, SQL is actually a very acceptable solution. And if you're using SQL, it probably means that you're going to be doing something like single-leader replication. And in terms of partitioning, because obviously we have a ton of video data, I think that it's very sufficient to partition the video chunks table by just the video ID itself. You know, just make sure that all the chunks for a given video are on the same machine. And then for the video table, which is just like a video ID and a name, perhaps you partition that by something like a user ID of the person who posted it. So, you could quickly go to their wall and find all the videos they've posted in the past.

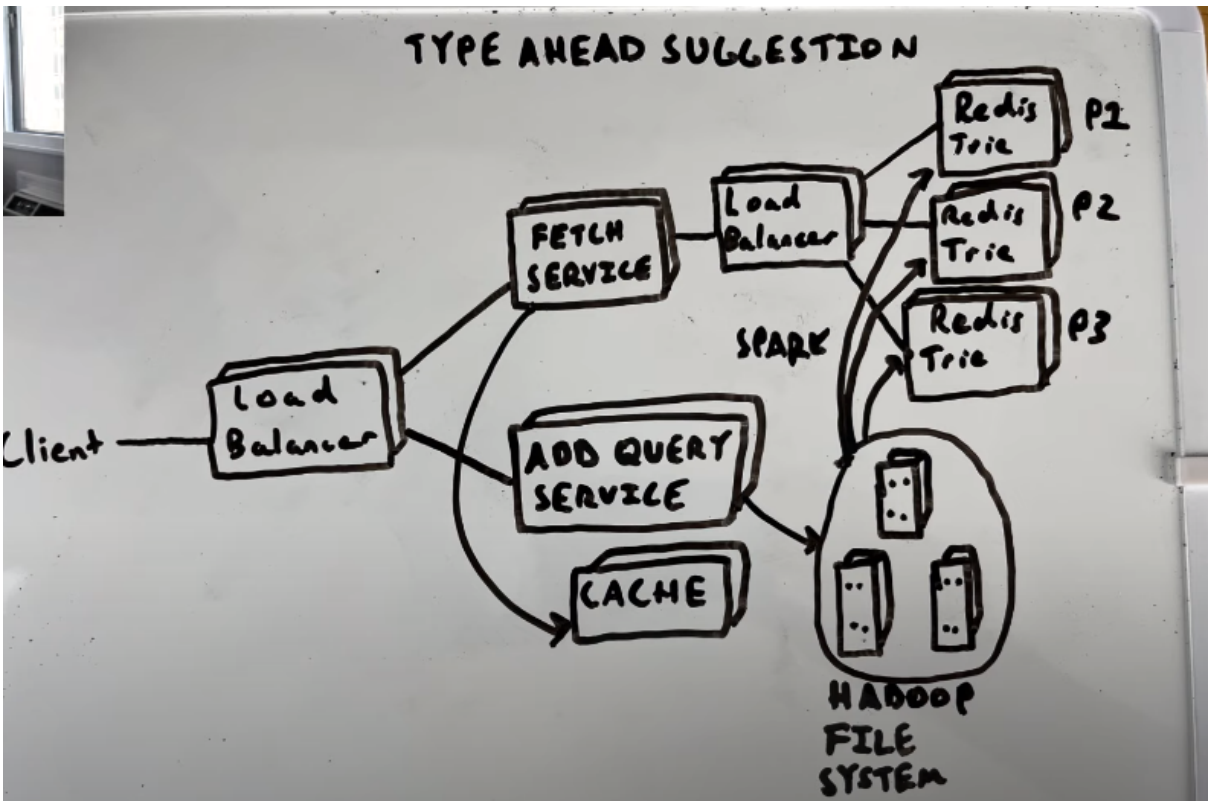
TYPEHEAD:


Capacity Estimates
5 Billion Searches per day
60k searches per second
100 million unique terms
3 words per term, 5 characters per word
3Gb to store all terms

API Design
Fetch(userId, prefix)


TRIE
Root
A
P
P
P
L
E
END
(0x ABC, 5)
P
E
O
E
END
(0x A21, 10)
D
END
(0x 123, 3)
P
END
(0x 345, 2)
TOP TERMS
(0x A21, 10)
(0x ABC, 5)
(0x 123, 3)
(0x 345, 2)
Go from bottom up to calculate most popular terms from each node!
8:29 / 19:20 • Architectural Overview


TRIE
Root
A
P
P
P
L
E
END
(0x ABC, 5)
P
E
O
E
END
(0x A21, 10)
D
END
(0x 123, 3)
P
END
(0x 345, 2)
On Disk:
ROOT 2
A 1
P 1
P 1
L 1
E 1
END 0
P 2
E 1
E 1
END 0
D 1
END 0
P 1
END 0



because everything here needs to happen with such low latency because users are typing so quickly and want those suggestions fast a database probably isn't the right choice for a solution here but rather what we are using to kind of keep track of all these suggestions should be able to fit in memory

So in terms of basically what is the design that we want to use here, well keep in mind that basically we're typing out a prefix, and then from that prefix, we want to find all of the suggestions which are easily fetchable from that prefix. In the past, when we've been using hard drives in order to solve this problem, what we would typically do is rely on something like a binary search with an index on disk where all of those terms are sorted. And then we would go ahead and run this binary search in a manner that we could quickly find all the terms that have a given prefix, and I covered that in my search indexes videos.

However, because of the fact that we're working with memory here, we can actually achieve a better solution than just running a binary search and get a better performance than just having a bunch of words that are falling within a given range of terms. So what we actually do here is we use something known as a trie. Now, I'm going to go ahead and draw a trie on the whiteboard, but the general point is this: it's a tree that keeps track of all the words that we've seen using a bunch of nodes where each node represents a character. This allows us to basically, for every single letter of a word, go through a node-by-node in the trie and see if that word is in the trie.

So what we do is for a given prefix, we'll iterate through the trie, and any of the nodes that fall under the end of that prefix are going to be search terms that may have stemmed from that prefix. Now, the thing is here, we don't want to return all of the possible search terms.

We only want to be returning the top 10. This is where we have to think of some clever solutions because if we wanted to return all of the possible search terms, that could take a super long time. Additionally, if we just want to return the top 10 most frequent ones, even if we were to have those precomputed, we would still have to iterate through all possible branches from the trie after the end of the prefix in order to figure out which the most popular terms were.

So what we would be better off doing is actually precomputing what the 10 most popular terms are from that prefix node. As opposed to just putting the entire term itself in the node because that takes a lot of space, you can put an in-memory reference to the actual leaf node that corresponds to the end of that search term. By doing so, you can quickly go to the leaf term directly. By doing this, we can quickly cache the top 10 results from a given prefix node. Basically, the trie structure here is going to allow us to efficiently solve this problem and use memory to do it.

However, even still, this trie isn't perfect, and there are definitely other things that we have to talk about. The first thing to note is that obviously, as we have new search terms coming in, we're going to have to update our trie. However, we don't want to be updating our trie every single time a search term comes in because if we were to do this, we would be putting a ton of load on all of these servers. What we care more about is actually just being able to get the maximum read performance possible, and we don't want to bog down all these trie data structures with a bunch of writes. So we should instead be doing is basically taking all of our new queries that we're getting, logging those to some disk, maybe S3 or something like that, and then offline, maybe once per day, running a new batch job, either by MapReduce or something like Spark, where we aggregate the counts of all those search terms. Then we actually go ahead and update our trie.

The thing is, when we update our trie, we have to keep in mind the fact that it's possible that now some of the search terms may have been more popular than before. We're going to have to update our precomputed caches of the 10 most popular terms from any given node in the trie. This means that we basically have to go ahead and do our typical bottom-up depth-first search frequency calculations at every single node in the trie. So again, it's important that this is a batch job because we're potentially updating the entire trie, at least from the leaf to the root, every single time the leaf frequency of a given search term is actually going to be changed.

Additionally, another consideration that we need is the fact that this trie is actually going to have to be persistent, probably to disk. One thing to note is we can't really just store the trie as is in disk because the concept of pointers doesn't really work so well on disk. So we have to figure out a way of serializing our trie such that we can put all of this text on disk. The way we might actually do this is by basically taking the root node and then saying how many children it has, followed by the first child and how many children it has, and the child of that first child and how many children it has. This way, we're effectively storing a depth-first search list of the traversal, and that way, we can rebuild our trie.

But one thing to note is that our precomputed frequencies of every single term are now no longer going to be able to be stored. The fact that we're actually referencing a node in all of those precomputations because we're referencing the leaf node of a given search term is no

longer going to be valid on disk because the pointer was something that was being referred to in memory. So we're going to have to store the nodes of the trie and then upon taking that trie from disk and reinstating it in the memory of the given server, we're going to have to do that entire recalculation or precomputation of finding the most frequent terms from every single node in the trie.

Obviously, that's a significant computation. However, the point is we're only going to be doing this once we're restarting a trie server. As a result of that, it's not really a huge deal if this takes a little bit longer. And then finally, as far as replication and partitioning go, let's think about both of these. Obviously, we want replication. One server won't be overloaded by a ton of queries, and also, our data will be fault-tolerant if one of them were to go down. So replication makes a lot of sense. Also, having replicas means that we can potentially update our trie in more interesting ways. For example, if there's a master and a slave trie, you could potentially update the slave trie after a given batch job, turn that into the master, and then update the old master node. There's utility of replication beyond just fault tolerance.

As for partitioning, there are a few different approaches that we could possibly use here. We only want to be partitioning our data if it's going to make our queries faster because, keep in mind, there's only going to be a few gigabytes of data in the trie, and that's something that can fit on one server. So if it's faster to not partition that, we shouldn't be. However, there are some interesting partitioning schemas here that could reduce the load in a way that might make our queries faster.

One possible way is we take ranges of the trie or ranges of search queries and distribute those from machine to machine. The issue here now is that for a given prefix, we have to check every single partition to see all of the terms that could come after it. Considering we want such low latency, I personally don't think this is the greatest idea. Sooner we have to wait on the slowest possible partition to aggregate our results, even though these calls are made in parallel. The fact that we have to wait for this aggregation is going to take a while.

So ultimately, if anything, you would do range-based queries and possibly in a dynamic configuration to hopefully reduce hotspots. If there were to be hotspots, we would want to introduce a caching layer where, as opposed to having linear time scans where you go through the trie every single time a prefix comes in, you would just check if the result is already in the cache. If so, pull the results out from that. That would be a constant time scan as opposed to linear time, which would be really nice. The cache, like I said, would be super useful there.

Oh God, I'm actually dying. This is so brutal. Okay, let's get into this diagram. So I have my type-ahead suggestion diagram written out below, and basically, the premise is this. For starters, we have our client. Just as a little bit of a front-end optimization, we should probably make it the case that the client does two things. First of all, the second the browser opens or whatever client it is that is going to be ultimately connecting to our service, they should probably instantly establish a TCP connection to our backend. The reasoning for this being that a TCP connection, since it's a multiple-way handshake, actually is going to require a couple of round trips. As a result of that, the earlier we get this done before we really need these super low latency queries, the better off that we are.

Additionally, we should probably make it the case that we're only going to perform searches for the type-ahead once we've actually let the client wait a couple of seconds. That way, we don't basically perform a search every single time they're going to type a character because that would be hammering our servers too much. Okay, with that out of the way, the client is going to hit our load balancer, which is going to choose one of our fetch service servers. Once that happens, hit a second load balancer, which basically holds our partitioning configuration for our Redis instances, which are going to actually hold the data structure for our trie.

The trie is going to contain all of the lookup information. I would hope it would be dynamically sharded based on the actual range of strings, but again, we don't necessarily need to use sharding here. I just included it for the sake of complicating the diagram a little bit. Additionally, we have our add query service. The reasoning for this being that every single time a client actually ends up choosing what they're going to be searching, we want to take that query and put it in HDFS. The reasoning for this being so that at that interval that we specify, perhaps once a day, maybe twice a day, we're going to run a huge batch job through something like MapReduce or Spark. We're going to then update all of our Redis instances where we can basically create a copy of this new trie and then switch over on all the Redis instances to using that trie.

Note that the Redis instances are replicated so that in the event that one of those partitions goes down, the other one can take over. Whatever ends up taking over for that Redis service that went down can go ahead and re-establish from disk the trie. They can pull that from HDFS. Finally, the last thing is we have a cache layer so that the fetch service can go ahead and quickly pull from that cache if the search term was recently used by either that user or someone else. Something like an LRU policy works very nicely for this cache. You might say to yourself here, "Oh well, actually, isn't the cache going to be equally as linear time as the trie?" By virtue of using a hash map on that cache table, we're going to have to first in the cache actually convert the string to some sort of hash, which involves linearly going through the string and making that conversion.

And that's the same complexity as going through the trie. But while I might agree with you, what you could actually have is before the client even submits a request, they actually on the client side convert the string to a hash. That way, we can make it such that we genuinely just have constant time operations in the cache, and we're not overloading our cache. We quickly receive the result.

Twitter/instagram

Functional Requirements

- Make posts with text, images, or video
- Follow other users
- News Feed that quickly loads new posts from followers

Extended Requirements

- Liking & Replying to posts

200 million daily active users

each user follows 200

100 million new tweets per day

300 bytes per tweet

30gb tweet storage per day

55tb of disk over 5 years

180gb of cache per month

6tb of cache per day if each user's home feed is cached on a separate server

Posts Table

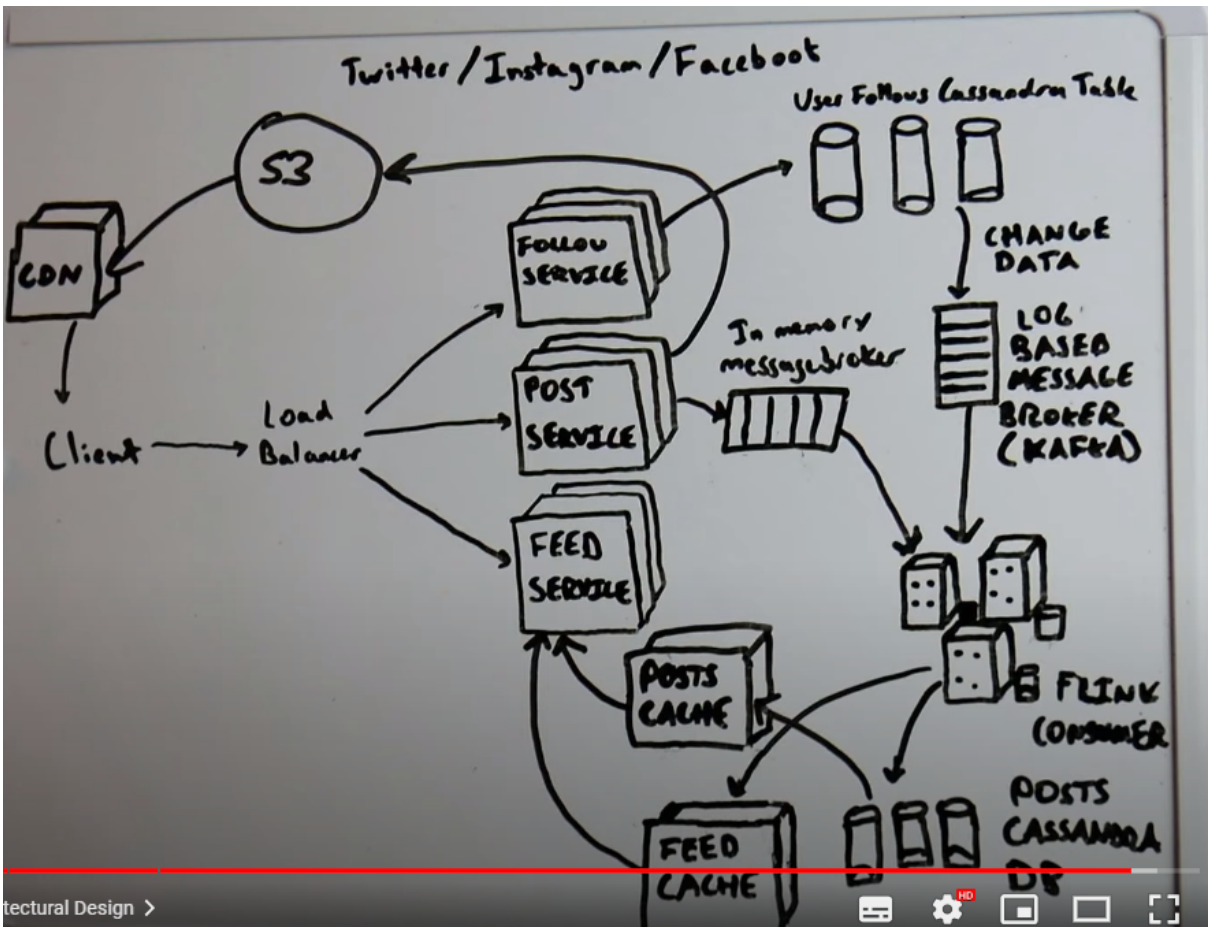
UserId: int
content: string
metadata: json

Users Table

UserId: int
email: string
pw_hash: string

UserFollows Table

UserId1: int
UserId2: int



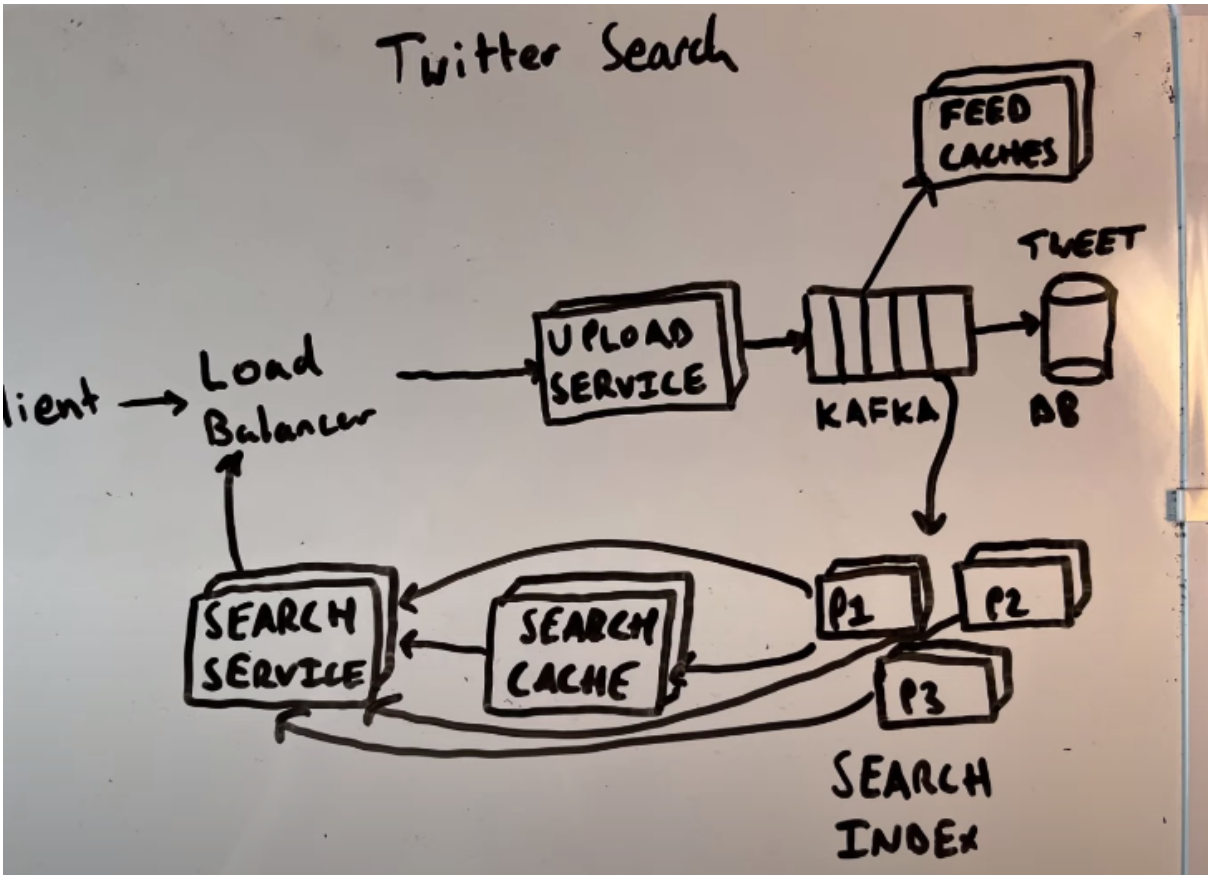
Twitter search

Capacity Estimates

- 400 million tweets per day
- 300 bytes per tweet
- 120gb of tweets per day
- 730 billion tweets per 5 years, 200 Tb / 5 yr
- 40 bits = 5 bytes per tweet ID
- 15 search terms per tweet
- 55TB of storage in index / 5 years

API Endpoints

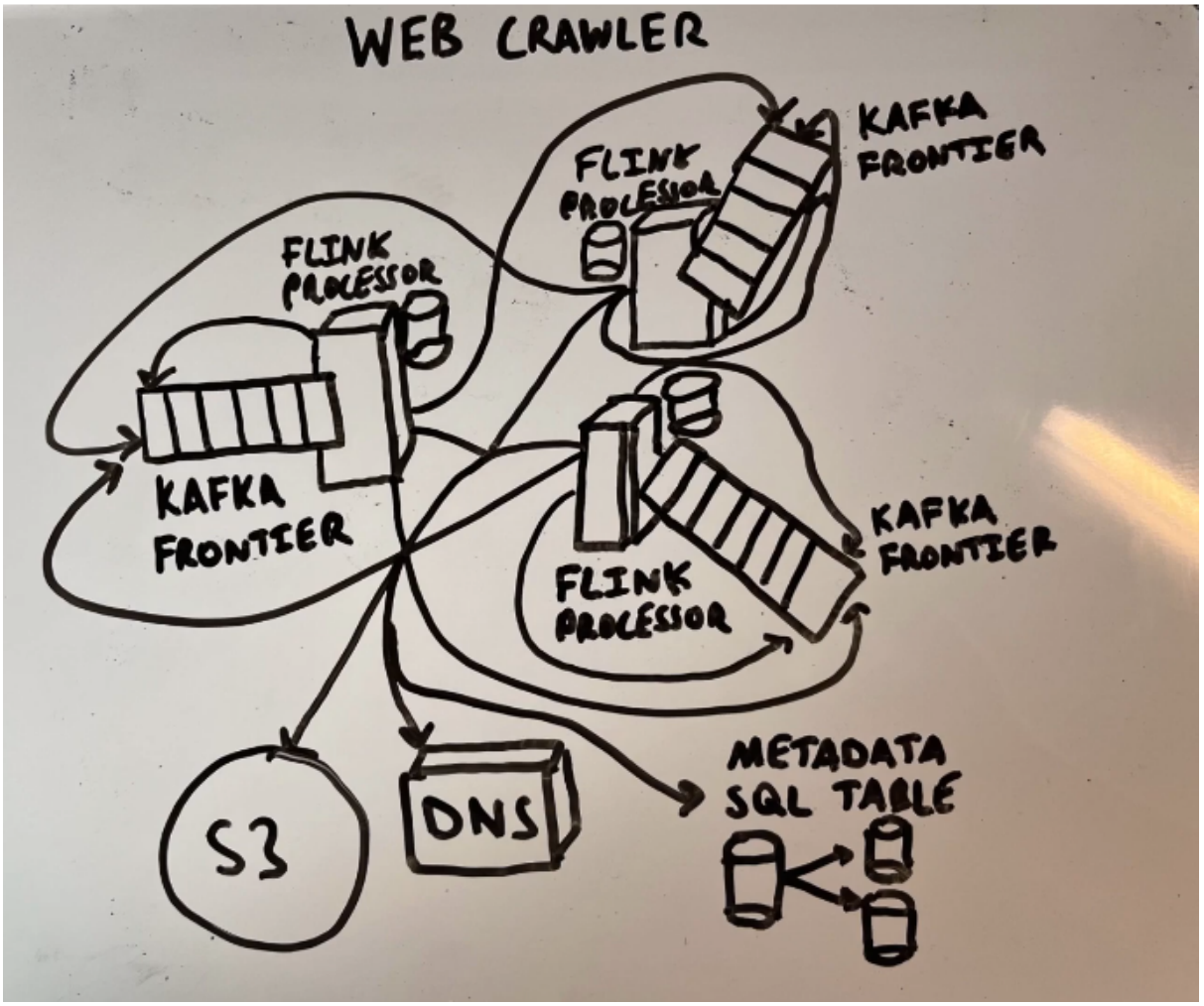
search(term, sort_order, page_num, page_size)



all right so normally in this part of the video i would do something like a database schema but it's actually the case that we're building a search index here which means there is no database so let's talk about what a search index might actually look like first what it might look like on a single node and then also at scale because obviously like i mentioned we have so much data here and so many users that we're having to do this over multiple nodes in a cluster and as a result of that there are more considerations to think about so like i said there's no database schema what we're actually going to be using instead as opposed to a traditional index where documents are indexed by their primary key or some sort of id is we're using something called an inverted index now the inverted index is basically the central part of why search indexes actually work so well fun fact i actually breaked an interview because i didn't know this and i feel stupid as a result of it basically it's going to have something like a hash map where the key is going to be a term so for example donald trump and there are plenty of tweets that correspond to this and then all of the tweet ids that have that term in there and so basically now what happens is that every single time a document is uploaded into the search index it gets parsed into its relevant terms and traditionally there's going to be some sort of extra logic here right like we don't care about terms like the a and things like that and we also want to do some additional pre-processing on some of the terms you know taking out ings taking out apostrophe s's and the way that we parse these documents is very essential to kind of what comes up when we do a search because it directly impacts for which terms which document ids are going to come up ow lucine is built a while ago but it has a bunch of super cool features like i mentioned there's prefix searching there's suffix searching you can do levenshtein distance which basically allows you to do these things called fuzzy searches so you can find similar words so if i were to you know do a search but i spelled the word wrong it would give me documents where they contained a search term that were similar to my misspelled word and then additionally it also does some ranking based on a bunch of you know nlp algorithms so if you've ever taken an nlp course or have any familiar with that it uses things like term frequency and also inverse document frequency to try and gain the general relevance of a document for a given query i'm not going to go into those too much but i figure these are things that you can always look into on your own if you want because they are interesting so we understand now generally how that single node search index may work but the point is i mentioned that it's not just one node we have to distribute this thing over a ton of nodso this actual lexing stage is extremely important there's a lot of thought that needs to go into how those documents are parsed in addition to that the terms are actually going to be sorted in the index so for example you know the term aardvark is going to come first then you'll have apple and you'll have bob the middle of charlie and the reason for doing this is that it makes it really easy to do prefix searches so like i've said in some of my previous videos one of the cool things about indexes is that you can actually go ahead and binary search them so if i have like a term that i say i want all of the documents that have like the letter like ap in them so what i'm going to do is i'm going to binary search in our index between ap and then it goes what nop qrs so between ap and aq and i'm going to binary search in our index so what that's going to do is it's going to reach basically apple apricot all of the other ap words and then we can go get all of the documents that have that prefix in there which is really great additionally you can do a similar search with suffixes where you basically reverse those document terms and then you do that same exact prefix search so you would store a second copy of the index where all of those search terms are reversed and that would allow you to do a binary search to search for suffixes and so a lot of the kind of logic for building a search index is how you can turn all of these basically string manipulation problems into prefix searches so you can binary search through the index this is um all kind of handled by this one really popular open source search indexing engine called lucian nes because we have many many terabytes of data that have to be able to be searched so now if you recall from again another one of my concept videos a super long time ago whenever we're going to be building out some sort of distributed index that's like a secondary index we always have to ask ourselves the following question do you want a local index or a global index and this is something that's extremely relevant here so in our case for the search index what a local index would be is basically saying every single node is just going to handle a range of documents and then on that node we're going to have every possible search term and only have the listings of the documents that are held on that note a global index might be say you know the term for jordan is on one node and every single possible document is going to be listed on that node and then i don't know the term for uh kdoptin is going to be on another node and every single document that lists her is going to be on that node now you might say to yourself okay well actually the global thing sounds really great here because what we want to be doing is actually optimizing for read speed and the nice thing about a global secondary index is that you can quickly grab all of those ids at once without having to rely on making multiple network connections to all of these nodes in the cluster whereas with a local index it optimizes for writes because you can basically just quickly write the document to one node and then know that it's going to go into that local secondary index without having to do any extra network coordination like two-face commit or anything like that however the truth of the matter here is that a global secondary index for something at this scale is probably not feasible the reasoning for this being that certain search terms are literally just too popular i keep using the example of donald trump because there are so many tweets about them but if you ever are going to do a search index of trump on twitter you're going to see millions or maybe even more tweets and as a result of that you probably can't hold all of those ids on a single server and even if you did that server would be so hot it would be such a hot spot because so many people would be querying it that it would generally probably just go down due to the load and even though you could probably handle this to an extent with something like caching it really doesn't help that there's going to be so much storage on the server as a result of having to hold literally tons of document ids and as a result just generally it seems with these super large search indexes that a global index doesn't make sense and as a result you kind of have to just partition it in a way that only certain documents can go on certain nodes which is unfortunate but um that is the way things go i have seen some research papers that i've run into discussing a hybrid approach where you kind of do some sort of global partitioning some sort of local partitioning but at least the super big distributed search engines like elasticsearch happen to do everything locally and so that probably seems like the best way to go here in terms of the way that we may actually end up sharding those documents because like i mentioned we're going to be doing this locally we can either shard them completely randomly where we might do some sort of consistent hashing we can shard them in a way that's based on time temporally because if we ever just only want to search the newest posts maybe it would be best to kind of batch all the documents um by like one day at a time so that all the documents from a given day are on the same instance of basically a node in the cluster and that way we can search them all really quickly or perhaps we could somehow put related documents together with you know search terms in common on the same cluster obviously that would be the most ideal thing but it's not always possible without some sort of you know algorithm and then we would have to keep track of the mapping of where each document is actually held in the search index so with this in mind now every single time that we call our api and want to get the search results we're going to have to aggregate the results from a bunch of these individual nodes in the cluster which is unfortunate but at least on the bright side it can be parallelized we don't have to wait for one request and then sequentially get another request and sequentially do that we can have an aggregator

server that's going to in parallel make requests to all of these indexes and then you know aggregate the results based on some sort of sort function that the user is provided that could be on something based on score that could be on you know the date the post was made the number of likes it has anything along those lines but the point is we're just going to have to aggregate the data the last thing that i really wanted to point out because i didn't really like the way that grocke and the systems design talked about this or rather they didn't talk about it is that it seems that um basically whenever you're using something like a search index what we're putting in there is derived data now derived data means that the search index itself is not our main data store and we shouldn't be using a search index as our main data store the reasoning for this is that generally speaking the documents are not going to be indexed by some sort of primary id and as a result of that you know just being able to search for documents say based on a user or something is not as good and oftentimes a lot of this stuff is in memory and so the data is not as durable and generally speaking it's just not a good idea to use a search index as your primary data store so what we do have is a database where tweets are going to be held what we should be doing is instead of just every single time a tweet is uploaded hitting our server and saying hey send this tweet to both the database and send it to the search index um hopefully we realize that something like that is problematic the reasoning being that we have then an atomic commit situation on our hands what if the database right were to succeed in the search index one were to fail what if the search index were to succeed in the database were to fail these two could become out of sync and as a result whenever we're dealing with derived data one cool thing that we can do is actually use a queue here so we can basically take something known as the change data from our database so every single time the database receives a write it can propagate that right to something like a log base message broker by passing something known as its change data so it takes the change data it passes that right to a log based message queue such as kafka and then that can then go ahead and be propagated into a search index which will automatically update itself by doing this we can ensure that the search index is going to stay basically in sync with our database we don't need an expensive two-phase commit protocol and additionally assuming that we have item potent operations on our search index meaning that they only occur once if we were to upload a document to it or you know if basically having to upload a document more than once only results in it seeming like it was uploaded once then all should be good there and we won't have any consequences if a message were to be replayed in our log-based message broker in the actual twitter example that we built ourselves in a previous video what we had is all of the tweets first going to a message broker before actually hitting the database because we wanted them to also be sent to all of these redis caches representing the feeds of users so if that were the case we could also just pull from that message broker and send it to our search index effectively keeping everything in sync which is going to be great what i've just described is basically what elasticsearch does and elasticsearch these days is effectively the leading distributed search index and kind of the one thing that i haven't really mentioned that makes elasticsearch able to run as fast as it is is basically the very very heavy use of a ton of caching obviously if we're going to be using a search index we want the fastest possible reads and we don't really care as much about write speed with that in mind we want to basically be caching everything as much as possible of course if we have popular terms that are being frequently searched we want something like an lru cache to be serving the majority of our requests additionally another cool thing that elasticsearch actually does is as opposed to just caching individual pages of an index because basically elasticsearch runs using um ss tables and lsm trees or i should say lucine does as opposed to just putting the entire ss table in memory which is what most caches would do for a database elasticsearch actually caches the entire results of a query so it might cache something that's even more complex than just a full ss table and as a result of that it's really able to speed things up by saving a ton of computation time by not just caching a page of the index but actue client

Web crawler

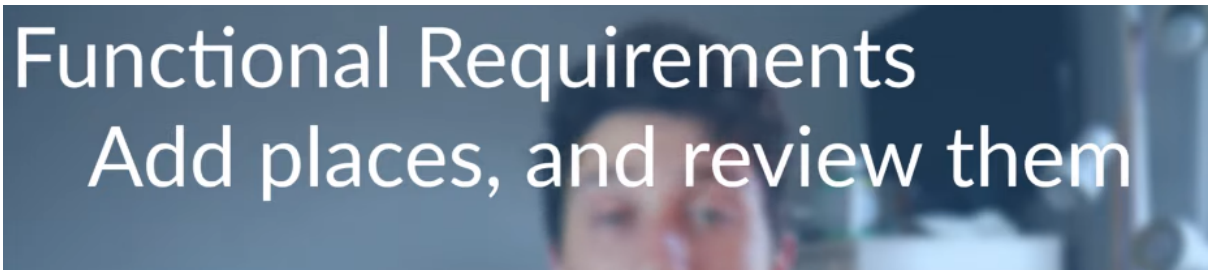


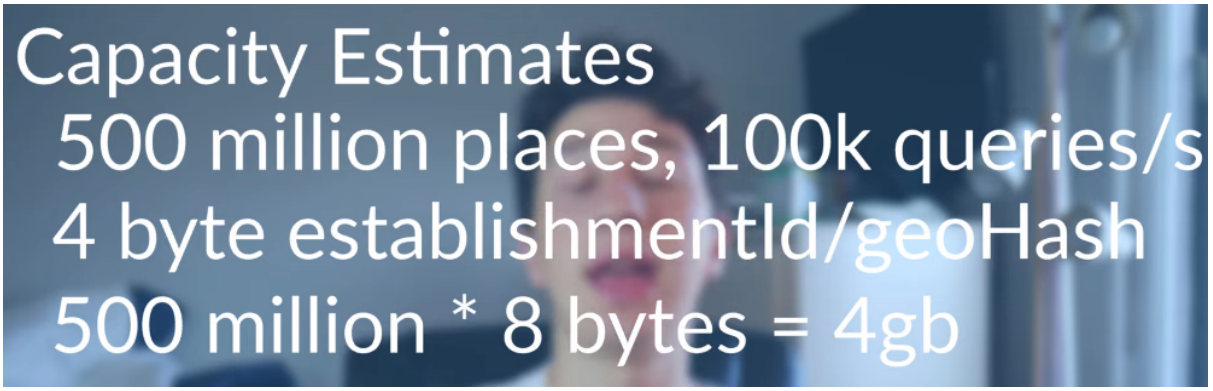
o basically for this video i'm going to start out by just giving the initial outline of a web crawling algorithm and then i am going to talk about some places where i feel like it can be optimized a little bit um basically optimizing from the grocking the systems design solution which i as per usual feel like doesn't go into as much depth as it could with the actual technologies that should be used in this situation so anyway here's the general process for how we would go about building something like a web crawler if any of you guys are familiar with graph traversals and have ever done basically an iterative breadth-first search or a depth first search you know that we basically need some sort of data structure that is going to store all of the nodes that we're planning on visiting next for now let's call this the frontier so imagine we were doing breadth-first search as we all know from our lovely data structures and algorithms the way that you implement a bfs in an iterative fashion is by using a q similarly we're probably going to be using a q however this is just more so going to be in a distributed fashion so if you're already thinking about it i'm probably going to be talking about message brokers lately and in the sense that we want durability probably some blog based message brokers as well so basically we are going to start with some seed urls which we're going to place into our url frontier and basically every single iteration of this web crawler we're going to select some of these urls from the url frontier that we haven't yet visited we're then going to consult our domain name services where dns is basically just a mapping from a human readable host name to an actual server ip address so for example wikipedia.com or my personal favorite pornhub.com is going to have a single ip address that is held by dns servers all over the world unfortunately these dns servers are themselves sharded and so it's probably going to be the case that we're actually going to have to make a network call to them rather than just locally storing the entire thing which would be ideal for processing of the scale but i'll touch upon that more later so basically after we call dns and get the ip address of a given server what we're then going to do is fetch it on some of our own servers get that html parse it in order to find other urls to be added to that frontier assuming we haven't yet seen those urls and we can basically check to see if we've seen them and then additionally what we're now going to do is process that html document again this would be the extra stage that we've basically said we're abstracting away for now we're not thinking about it but this could be you know placing it into a search index just taking the html itself and putting it somewhere into some blob store or something like that and at this point what we probably want to do is perform some sort of deduplication step the reasoning for this being and this was actually new to me is that there are literally just tons of websites mirroring other websites on the internet where they have literally the exact same content so as a result it's important for both space and just kind of the validity of your results to say not only do we not want duplicate urls to be searched but we also have to look at the content corresponding to that url and make sure that it's not duplicated you may start thinking now back to something like our you know our youtube problem for example where we were thinking about getting rid of duplicates or even potentially that dropbox problem because kind of the methods that were used in those are going to be pretty relevant here and then finally once that document has been processed or fetched onto our nodes we can go ahead and add those to some sort of storage that we have because we have 1.5 petabytes of possible documents that we're going to need to store it's probably best that those are going

in something like s3 or some other sort of object storage where we basically have limitless elastic scaling abilities and can cheaply just put a bunch of crap in there so anyways i'm now going to talk about some optimizations for this algorithm because i more or less gave a general overview but i think there are certain technologies that we can use here that is going to really simplify this process and give us a potentially better experience than if we were to just kind of do this all in a really wishy-washy way so the first consideration that i actually want to talk about is the type of graph traversal that we want to do technically we could do breadth first search using something like eq we could also do a depth first search depending on you know what data store we want to use as our queue and i guess instead of using a cue we can basically just implement a stack instead um i guess it doesn't really matter as much i think breadth first search probably makes the most sense to think about conceptually where you're kind of just taking all the urls from a page and then probably going to be parsing those around relatively the same time another way of actually basically traversing pages is rather than going from all of the links that you find from one given page once you reach basically a given website so say we have you know wikipedia dot com slash a slash b you also are traversing wikipedia.com and also wikipedia.com a the reasoning for this being that there's likely going to be stuff in all of those sub domains and then we can probably quickly iterate through that entire host name the one thing to note here is that if we are to then basically take all of those links and put them back in our queue for traversal and then soon after traverse all of those we may incur a lot of load on wikipedia servers and basically break the policy of their robots.txt file which more less says just don't violate me when you web call me so you know sites have their policies they don't want to be reamed by our web crawlers as much as we would love to and as a result of that you have to be careful when you're doing things like this kind of path-based searching so for our next consideration the thing i want to talk about is basically sharding and data locality because that's going to be how we're able to get the best possible performance that we can and additionally there are some other concerns like you know using multi-threading for certain places and i'll talk about that in a bit too but basically we've said already that we're going to be fetching 6 200 urls per second which is an absolute ton and we know that and obviously no single computer is capable of doing that so this workload is going to have to be distributed and now the question is if we're going to be distributing this workload how should we do it in a way that makes sense well the answer at least what i think should be the answer is we should probably be distributing it on nodes where basically the work is um hashed out or sharded based on the hostname so for a url like wikipedia slash a slash b c dot txt the host name is going to be you know www.wikipedia.com so all wikipedia links are going to go to the same computer and there's a couple of reasons for this and i'll explain those right now the first is that we don't want to search duplicate urls so if we're hashing things basically in a way that you know is consistent and not just you know sending random urls to random places if we ever see a duplicate url it's going to go to the same node that's processed and that means that we can basically store for a given node locally which other urls it is seen so that way we don't need some sort of centralized database in order to go ahead and consult that and say hey have we seen this url yet but rather every single node can keep a local store of all the links that it's seen before and as a result it can quickly perform that check of saying hey have i seen you or have i not furthermore this type of partitioning schema can really help us take advantage of data locality as far as dns goes as i mentioned dns is huge in the sense that it's basically mapping every single possible host name to an ip address and as a result of that we probably aren't going to be able to put an entire dns locally on one of our processing nodes as you know say we've said we might do in the past when doing something like stream processing or batch processing and doing a join so instead what we're better off doing is using caching so basically each of these nodes that is processing um you know a given set of host names and we can kind of just choose that using consistent hashing so they each get a relatively even subset of the host names and an even workload any node doing this processing can locally cache all the dns results that it's getting the reason this is really useful is that a hostname may have many sub urls right like you can have all these slashes after a host name but it's only going to have one ip address that we care about and so by caching the first call to dns we can then make very few network calls in the future which is really great the one thing to note is that because of the fact that we don't want to be hitting a given server too many times consecutively it might limit the efficacy of this cache because you know say that result may have expired by the next time we get around to this hostname so in an ideal world the cache will be big enough such that we can hold all of the relevant dns results for that node but practically speaking we may have more cache misses than we might hope so anyways like i mentioned we can basically cache a bunch of these dns results and additionally we can cache or hold in some sort of local storage all of the urls that we've processed on this node so we can quickly tell whether or not we've seen it so so far we've actually been able to take advantage of data locality really well and i'll explain in a little bit what technologies i think we should use in order to do that but the one area where we really can't take advantage of data locality is basically checking for duplicated html files the reasoning for this is just that i you know i can have wikipedia link a over here and then i can have i don't know some random blog over here and they may be the same html page but at the end of the day because they're on different servers their host names are different and as a result they're not going to be sharded to the same server most likely so generally speaking we're probably going to need some sort of centralized metadata store that is going to contain hashes of all of those html files so that way we can check for duplication that's really annoying obviously because it's going to more or less require extra rpc calls or http network calls in order to check for duplicated files and obviously that's pretty frustrating but at the end of the day there's really nothing you can do about that being said the one thing i would recommend here is probably doing this deduplication process in a different thread as the actual fetching of the html the reasoning for this being that even once we've parsed that entire document and seen the links that the document points to we can really quickly on our main thread or basically our fetching thread tell if you know we should actually be searching those links in the future and on the other hand it's going to take a much longer time to do the deduplication and so we could probably leave that to a separate thread which then goes ahead and checks for whether or not that html file is in a centralized database by you know basically making a query based on the hash of it and then if it's not in the database we can upload it to our blob store such as s3 just a quick thing to touch upon is that this means that we're probably going to need some sort of metadata table which is matching a host name to some sort of hash or s3 link probably both and then additionally what that means is this gives us an opportunity to further parallelize uploads of html files to s3 because we can break those into chunks again and what that also means is that it gives us opportunities to further check for basically duplicates of all the chunks and then if only some of the chunks are duplicates and some aren't then we can only upload the differences and then basically construct that file in our metadata table so if we are using chunks then our metadata table should probably be something like a relational database simply because then we're going to want transactions to actually go ahead and make sure that all the metadata chunks have been logged in the database maybe you could use something like cassandra and you know the partitioning would work out nicely but again i feel like having the acidity aspect of it when you're dealing with chunks in multiple database entries is really useful to have and then finally in terms of the actual technology that we want to be using one thing that kept popping into my mind the more i heard about this problem was you basically want a bunch

of nodes that are you know doing some sort of processing work they're keeping some sort of internal state it's very very important that that internal state is durable and can be reconstructed right like we don't want to be able to parse some urls and then you know one of our parsing machines goes down and then we completely forgot the fact that we parsed that url or now we're not going to you know parse all the things it leads to and then in addition to that we have this frontier which has a generally cue-ish nature and that also has to be sharded in a certain way and also has to be replicated and to me the more i thought about this it just kept sounding like a perfect scenario to be using something like flink and kafka where kafka is basically going to be our url frontier and that's going to be a partition queue where every single time you are figuring out new urls on a flink node that have to go into the frontier you just send them right back to kafka and that can also be done in a separate process so as to speed things up a little bit and then every single time you're now pulling in a new url to process you pull it in from a kafka queue to a given flank node which is done one at a time that is going to be completely durable because the kafka queue is going to be based on disk and then you know if any of our processing nodes ever fails we can use the basically the way that flink checkpoints state through a bunch of distributed snapshots in order to basically restore our state on either another node or something like that and then continue processing our web crawler it seems like the the flink kafka combo is a really valid way of actually doing this web crawling and obviously like i said we're going to be caching a bunch of state on each flink consumer such as the scene urls and previous dns results for a given hostname so it seems like that's kind of the best way to do this so with all this being said let me go ahead and draw a diagram okay finally let's take a look at this diagram got a little messy with these arrows kind of reminding me from my poop from earlier but with all things considered basically we're going to have the following because i'm trying to emphasize this here as you can see we've got all these partitions of flink processors corresponding to a partition of a kafka frontier so basically the point here is that i only drew out three but in a real web crawler we probably have hundreds of these where all the fling processors can write both back to their own queue they can also write to the other cues because sometimes they're going to parse urls that are in different partitions than theirs and if so they're going to have to send that to the right queue fortunately kafka does manage that load balancing for you which is great and then in addition to that not only are they basically consulting their local state which you can see based on that little database that i've drew next to every single fling processor but also they're going to have to be uploading chunks to s3 they're going to have to be getting dns results and caching them in their local state and then finally they're going to have to put corresponding chunks or at least the hashes of the chunks and their s3 links in a metadata sql table which again i just use sql because it's simple everyone knows it it's transactional and here i'm just imagining that we'll be using something like single leader replication realistically there's probably going to be too much metadata to just go ahead and have it all in one huge table so that's probably going to have to be sharded in one way or another which is honestly no big deal it can basically just be done in a manner similar to uh something along the lines of the way that the flink processors are sharded but yeah it doesn't really matter there's no relations here we just have to get our sql table up and accessible by anyone so they can do those deduplication checks

Yelp design





API Schema

add_review(location_id, user_id, stars, text)

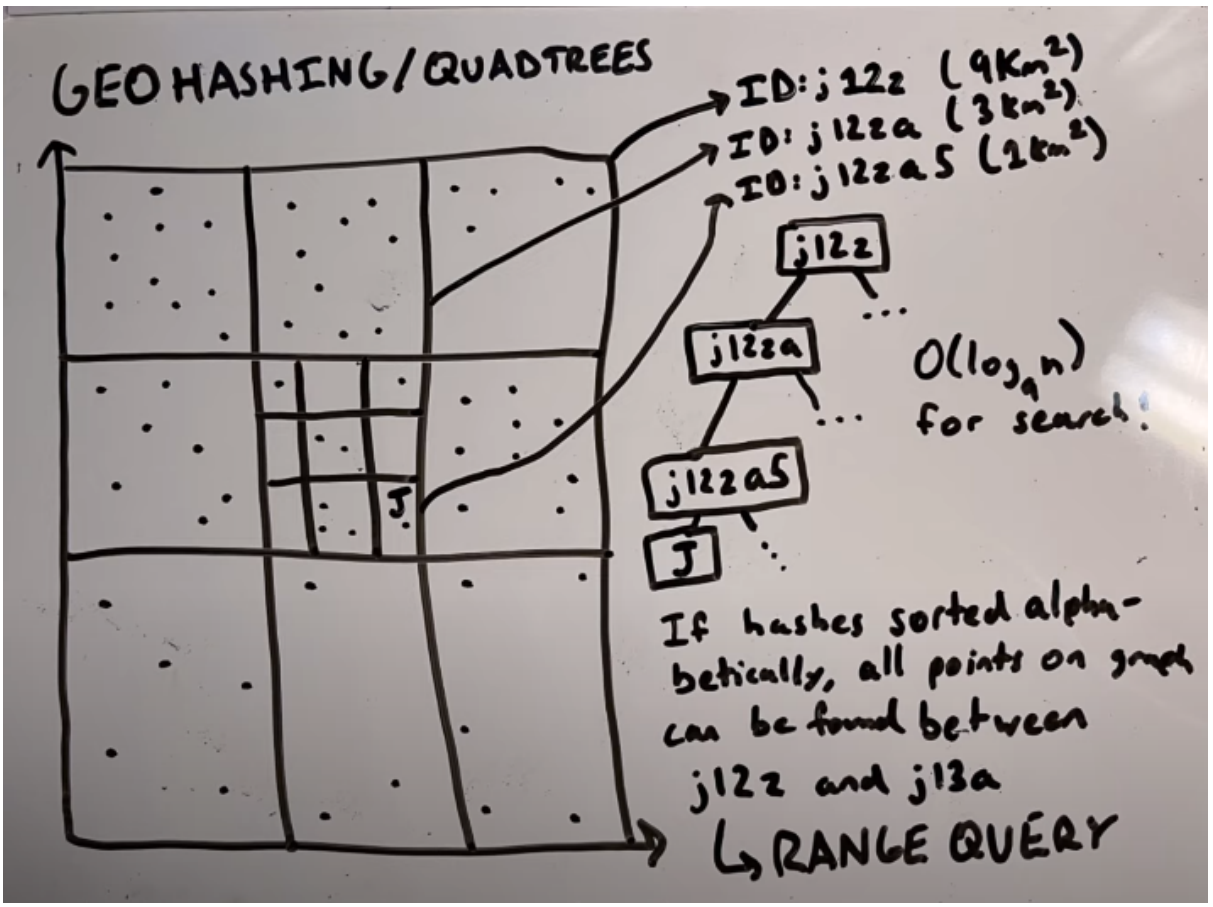
add_location(lat, lng, desc, category)

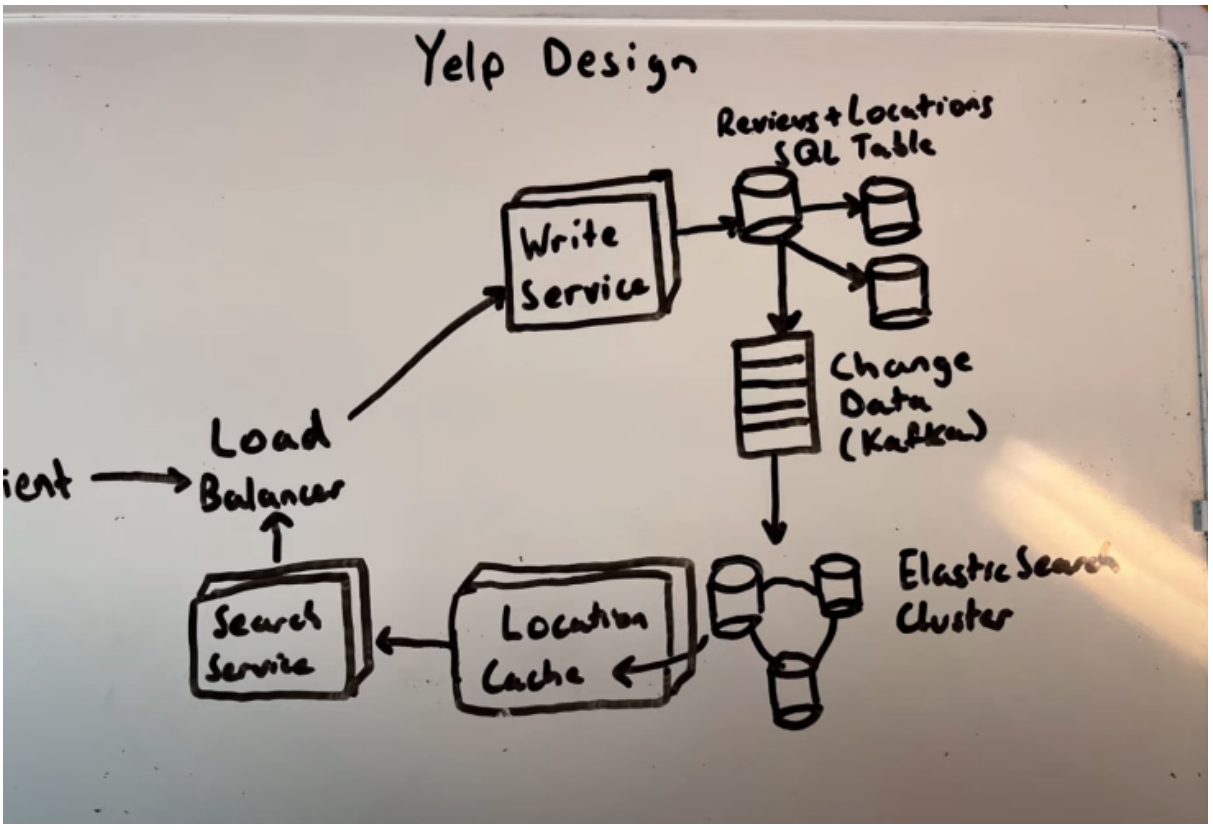
Database Tables

User(user_id, email, pw_hash)

Location(id, lat, lng, name, desc, category)

Review(user_id, loc_id, stars, text)


GEO HASHING / QUADTREES
ID: j12z (9Km²)
ID: j12za (3km²)
ID: j12za5 (1km²)
j12z
j12za
j12za5
J
O(log_q n) for search!
If hashes sorted alphabetically, all points on graph can be found between j12z and j13a
↳ RANGE QUERY


Yelp Design
Reviews + Locations SQL Table
Write Service
Change Data (Kafka)
Load Balancer
ient
Search Service
Location Cache
Elastic Search Cluster

all right let's talk about how we can now go ahead and kind of create an optimal solution here for this yelp problem to go ahead and find a ton of points near a given user so i'm going to kind of preface this all by saying i'm a visual learner and i'm sure many of you guys are too and i think it's going to help a lot if i kind of use the whiteboard here to pair with this problem and demonstrate to you guys how we're best going to solve it so to basically preface everything i am basically going to say i am point j on this 2d plane that i have pictured on the whiteboard and i have a bunch of points around me and what i want is to find all the points within a given radius so that's going to be some sort of circular area so as you can see i have my circle basically on that 2d plane now what if we were to basically have it such that all of the points on our 2d plane were just in a traditional database right so we're using some sort of index that is comprised of either an ss table an lsm tree or a b tree and to remind you all what that means is we have basically very efficient range queries on fields that we have indexed and as of now basically all of these points has a latitude and a longitude where we can basically just imagine the latitude and the longitude are kind of like their x and y coordinates but you know obviously it's a little more complicated than that but for now we can kind of just abstract that away so we have these kind of x and y coordinates of every single point and we have indexes on both which means we can perform these really efficient range queries over the x and y coordinates of all the points but not both at the same time so what are our options well basically all we can really do here in this kind of naive solution case is we can find all of the points that are going to be a certain x distance away from our original user and all of the points that are going to be a certain y distance away from our original user then once we have those we basically aggregate them together and filter all of those places out that aren't actually within that circle of a given radius as you can see that's obviously going to be a very naive and not great solution because it means that we now have this kind of quadratic complexity where we're considering basically a lot more points and as opposed to getting this really small area that's just surrounding our original user we're opening ourselves up to potentially having to search the entire globe at a given latitude and given longitude and the globe is obviously huge there are going to be a ton of points there and now we have to do all of this server-side filtering and it's going to take a really long time and be very inefficient so what is actually going to be a better solution so in an ideal world what we would have basically is some sort of bounding box right where we take our user we place him in the bounding box and then just by virtue of having a bunch of points mapped into this bounding box we can return all of those points really quickly and that's kind of the philosophy behind the geographic index that we're going to be building so instead of just having one bounding box we're going to actually have a ton of bounding boxes all of various sizes and all of basically varying locations and the goal of this more or less is to make it such that as opposed to having this big 2d search problem where we're searching across two possible coordinates instead we're basically just going to search for a given user's bounding box or perhaps the surrounding bounding boxes of it and then that way we can really quickly access all of the points that they contain so we're going to basically go ahead and split our 2d plane into a bunch of these bounding boxes but we're going to do it into a clever way that makes it easy to search using our typical database index so let's see how we can actually do that so i'm now going to bring up this other image and it's going to look a little bit weird if you haven't kind of watched my primer video on geographic indexes that i made a while back but basically the premise is this as you can see we have this entire 2d coordinated space split into a bunch of boxes and each of those boxes can have sub boxes and each of those sub boxes can have sub boxes and the premise is this more or less each of those boxes is going to have an id every single one of its child sub

boxes is going to have a similar id however it's going to have some sort of suffix so for example you can see the root box that i have there has a hash of j12 z1 and one of its children say the middle one in this case is going to have the hash of j12 z1a and that child one of its children is going to have the hash of j12 z1 a5 so as you can see we kind of keep appending to this original string the reason for that is as follows if we have a database and we keep all of these ids sorted it means that say we have a big bounding box we can really quickly find all of the points inside of it by performing a range query so for example if i wanted to see all of the points that i've drawn out in this part of the coordinate plane i can go ahead and perform a range query from j12 z1 to j12 z2 and that way i can get every single one of these points contained in that big bounding box additionally it means that if we want to basically get one of the smaller bounding boxes we know that all we have to do is append some sort of suffix to it and we can really quickly search through our database index knowing that everything is sorted and this gives us really great time complexity such that we no longer have to do this 2d search anymore and it completely lowers the search space if we're trying to look for a point so what i have drawn out here below is known as basically a geohash and typically in a geohash what you'll see is something like each bounding box will have nine children but you may have also heard of something known as a quad tree where each bounding box has four children and the way you would delineate a child is by taking basically the hash of the parent node and then adding either a 0 1 0 0 or a 1 1 or a 1 0 to it delineating one of the four directions that the child node could basically be in and then what you can do is you can take all of the sizes of the squares and normalize them such that for a given length of the hash we actually know how big that bounding box is and that's going to give us a lot of really cool capabilities so now basically if you know we're looking at user j and we want to find all the places within say a three mile radius what we can first do is find the actual geohash of user j from the lat long and that can either be something that is kind of cached in a database or it can be calculated on the fly which isn't too hard because basically this entire thing is effectively a tree and so as a result if we're searching for the geohash for a given lat long we can do that using logarithmic time complexity and so once i have the geohash for a given point we can use the surrounding bounded boxes in order to find all of the points their locations that are nearby and that's going to be super useful and it's very fast to do you may note that it's very possible that user j is at the edge of a bounding box so we may have to consider multiple bounding boxes that are neighbors of it in order to make sure that we're really encompassing all of the nearby points and that's something that's relatively easy to do if you just keep using bigger and bigger bounding boxes and then do a little bit of just logic as far as picking the right neighbors this is also something that can be parallelized so it's relatively easy to do overall okay so now that we've basically talked about our main algorithm about how we're going to find the surrounding neighboring points given the latitude and longitude of just one point let's quickly touch upon some other things as the actual technology that we may want to be using to hold our geographic index as well as some things like partitioning and replication because that's definitely going to be important for supporting very high read throughputs so firstly as we can see um you can use something like a traditional database and put a geographic index in it i know that postgres definitely has a feature in order to do this kind of geographic indexing however if you do recall i mentioned that one of kind of the parameters of this problem was that not only are we going to be searching on geographic location but we're also going to be searching on the actual name of the place so with this in mind perhaps what we're best stopped doing is actually using a search index to perform this functionality the reason i say this is that something like lucine which i mentioned in our previous video has a ton of functionality that can all be integrated in

order to perform very complex searches so not only can lucine basically support a geographic index but you can also store the actual name of the place within lucien and by virtue of doing so you can kind of cross-reference both the name and the geographic location in order to perform very good searches that will basically give us quick results obviously it's going to be the case that since we have such read heavy workflows we're going to want a lot of caching abilities in order to basically store the results of certain calls and like i mentioned what elasticsearch will do is if you're making these kind of compound queries where it's like hey both filter on geolocation and filter on name you may be able to even do it such that you can kind of just cache you know part of each query for later reuse so perhaps you could just cache the geolocation part and then you know as people kind of type in different um actual search queries you can use that cache geolocation query and then kind of combine it with the filter of the search term but anyways yeah i mean that's kind of just my motivation for trying to use a search index here as opposed to just using something like redis which i'm not sure has the functionality to combine something like a text search and a geolocation index it may but i just haven't really looked into it but additionally we have to make sure that if we're having a search index it's up to date our search index is not going to be our primary database for a variety of reasons one of which being that the data just isn't as durable in there and so as a result we want to make sure that any changes that we make to any of our yelp locations are going to eventually be reflected in the search index it's not overly important that those are instantly reflected and we definitely can't have strong consistency without something like two phase commit but we can either use something like a batch job to go ahead and update our search index say once a day with all of the changes from the database or perhaps even better if we want to make those changes visible quicker is to use some sort of stream processing so every single time that our locations database gets updated and we can just use like mysql as a database there for all of our yelp restaurants or anything like that we can go ahead and stream that change data from our mysql table to our search index and that's how we can actually keep things nice and up to date and that's a pattern that i've you know implemented many times in some of my designs on this channel and i think it's very feasible here as far as partitioning goes this definitely deserves a discussion because you always want to be very careful partitioning your geographic data in a way such that it makes a lot of sense for your end user and getting the lowest latencies possible so one really intuitive way to partition things is again whenever we're sharding data we want to keep in mind that we want basically all data that's going to be returned in the same query as close to one another as possible so if i have a bunch of restaurants in our search index we want it to be the case that all of the restaurants that are close in proximity to one another are going to be on the same note it is very possible however though that this is going to lead to hot spots right so say i were to basically say all of the points in this bounding box should be going on one given node in our say search index cluster it's very possible that if it happens to be the case that that bounding box was downtown manhattan they're going to be you know hundreds of thousands of points potentially in that relatively small bounding box and so as a result of that we kind of want to be doing this sort of dynamic sharding where we want to make sure that there aren't any hot spots and if one node were to kind of be overwhelmed by too many places on it what you would then instead do is basically use a smaller part of our quad tree or our geohash and use a smaller bounding box as the thing to kind of shard on so in that sense we're not using fixed size parts of the grid to shard but rather maybe shard based on hey as soon as this part of the grid has too many locations in it we should split that into it's either four or nine child pieces and then in turn shard those obviously it's going to slow down certain queries in

the sense that we may have to do some aggregating over multiple nodes in our search cluster but at the end of the day we don't really have any choice additionally like i said we're going to be having a lot of caching and that can help kind of mitigate hot spots in the sense that you know if me and a million other people are trying to see our yelp results for downtown manhattan the cash is going to be returning most of those and it can kind of hide all of that user load from our actual search index itself as far as replication goes we obviously do want to ensure fault tolerance however since we're not really dealing with the need to have super high right throughput here and nor are we dealing with the need for strong consistency something like asynchronous single leader replication i think suits this problem well both for our databases and our search index and you know cache and all of that which can just be lru by the way because i haven't mentioned it yet alrighty i got myself a nice little minimal diagram here so let's go ahead and walk ourselves through it first of all if we're the client we're going to hit a load balancer every single time we have some sort of request to make we can make that redundant through some sort of active active or active passive configuration there um i'm assuming we're going to talk about the right path first so let's imagine i'm adding a new place to our database or adding some sort of new review we're going to first hit our write service which we can basically horizontally scale out hence the load balancer and then the first thing that's going to do is just hit our sql table which contains all of the reviews and locations and you know we can obviously have separate tables for that a quick note to think about is that as far as reviews go we should probably shard them by location id just so that whenever you do query a location you can really quickly pull up all those reviews instead of having to aggregate them in parallel then furthermore as you can see we have some single leader replication going on there but in addition all the changes that are going to happen are going to be streamed to our log-based message broker aka kafka the reason we're using a log base message broker here is for durability of our actual rights so that nothing is lost if that message broker were to go down and then those are then in turn going to be propagated to our elasticsearch cluster i should mention that even though i'm using kafka here which insinuates that we're doing some sort of stream processing all of these updates could be written in batch say once per day at a time where you know there's relatively low load on the service and the reason that might help is it might allow us to perform some more aggregations of the data especially in terms of you know the overall number of stars that a given location has in terms of reviews or something like that and that in turn may help our elastic search cluster return more popular locations as opposed to ones that are less popular so anyways like i said the elasticsearch cluster is going to be updated either in batch or using streaming i think probably more practically someone would do it in batch but um you know if you want the changes reflected more quickly than stream processing is the way to go so i am using kafka here but it does put a little bit more load on our cluster so anyways the elastic search cluster then anytime the user is going to make a search for nearby locations is going to basically go ahead and search through its shard for a given locations grid and especially if that query needs to be expanded because either there aren't enough places or the location that we're searching for is at the edge of the grid we're checking then you can basically parallelize this query by checking the grids on other nodes of the cluster obviously whenever we're doing some sort of sharding we want to be using basically some sort of consistent hashing scheme to make sure that everyone is getting relatively even load in terms of the nodes in the cluster at least relatively even load proportional to their compute capacity anyways once those search results are returned they can basically either be in the location cache or not and either way they're going to hit the search service and be returned back to our client

Uber

Functional Requirements

Show nearby cars to potential riders, and keep their location up to date

Capacity Estimates

Push location changes every 3 seconds

1 mil active daily riders, 500k active daily drivers

~20 bytes of data to store per rider/driver in geospatial index

1.5mil * 20 bytes = 30Mb

API Design

Search(lat, lng, radius)

Match(userId, nearbyDriverIds)

GetRoute(startLatLng, endLatLng)

Database Schema

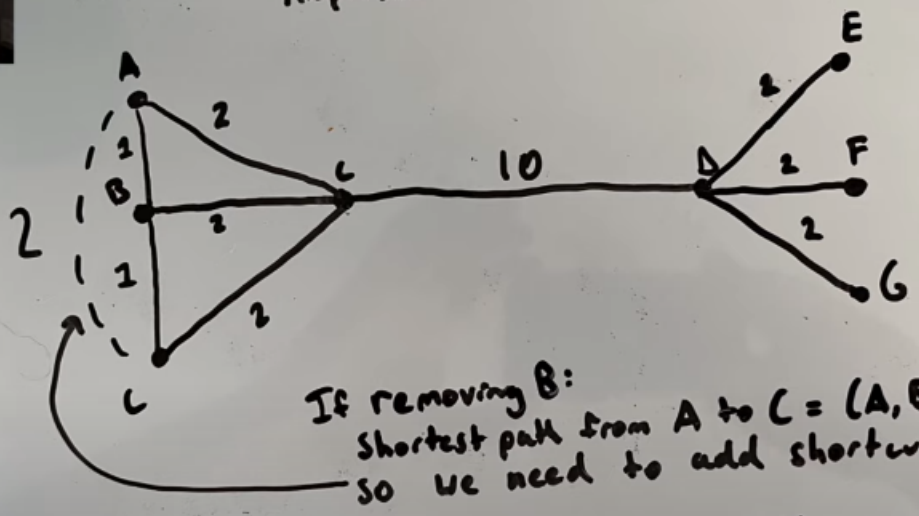
Users(id, email, pw_hash)

Drivers(id, email, pw_hash, in_ride)

Rides/Trips(user_id, driver_id, route, time)

COMPACTION HIERARCHIES

Goal: Create smaller version of graph with only "important nodes"

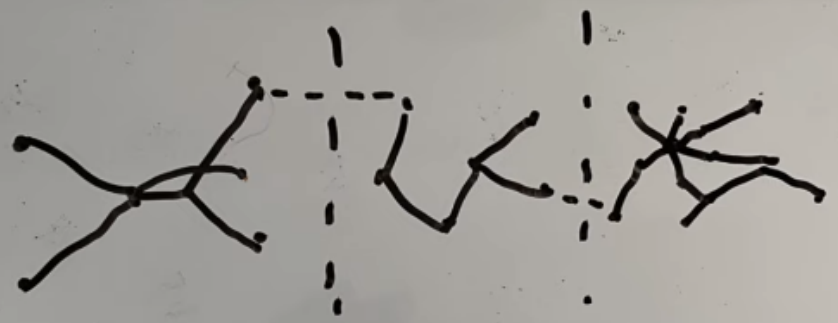


If removing B:
Shortest path from A to C = (A,B) + (B,C)
so we need to add shortcut edge!

We want to remove least "important" edges first!
→ Less important nodes require fewer shortcut edges to be added when removed
→ C and D are "important"! Do not remove

Architectural Overview

Graph Segmenting



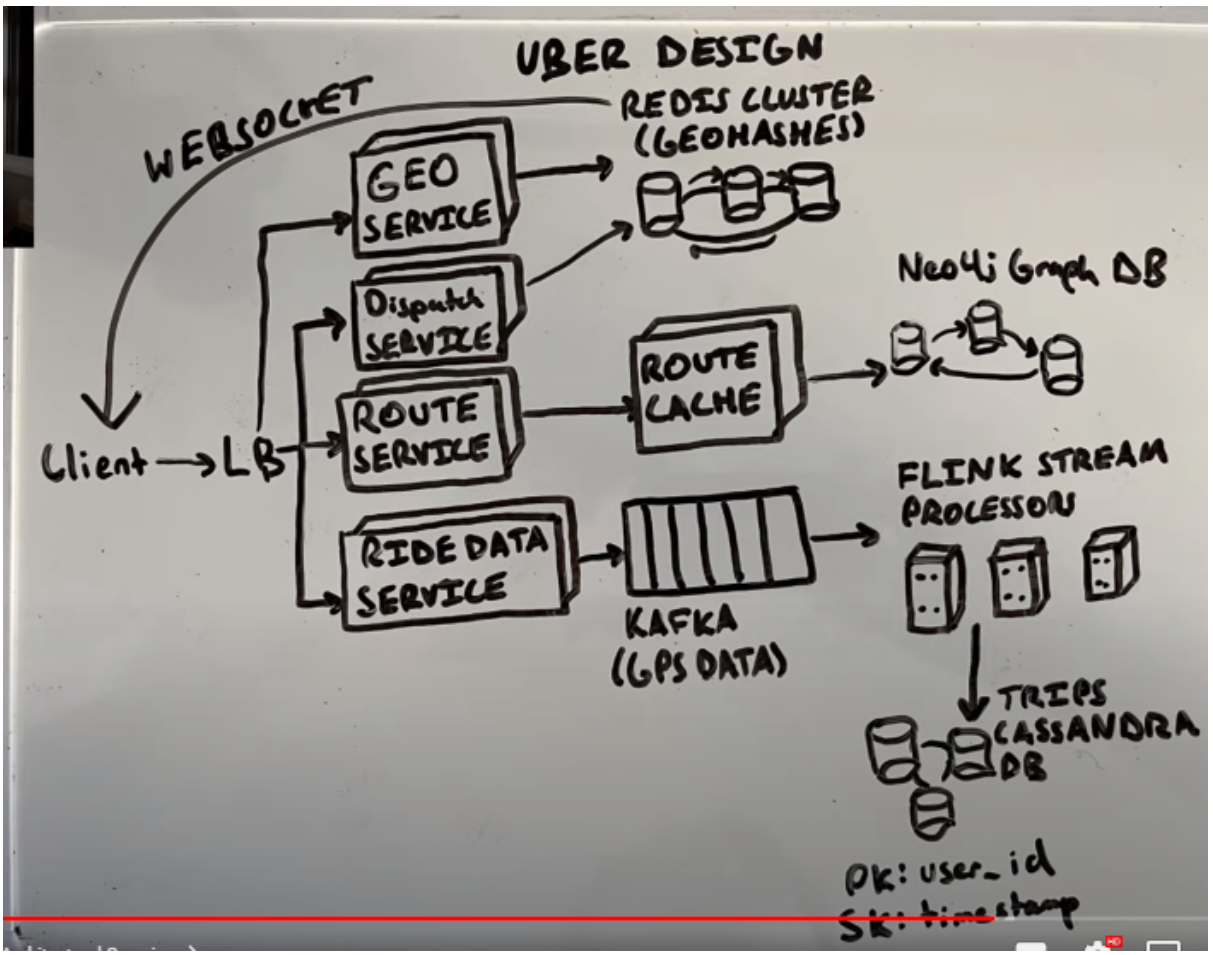
Split Graph into pieces (segments), and determine distance between segments! Use segment boundaries for Dijkstra's algorithm!
(Greatly decreases search space)

ll right so as far as the design goes for this uber back end i really do feel
like rocking the systems design interview leaves out a lot of key details so i'm going to obviously as
usual try and emphasize those but i also have to talk about the parts of uber that they also mention
which is generally very similar to our yelp back end so let's get started with that
as we know one of the main functionalities that we have to be able to support is a given user who wants a
ride should be able to see all of the other drivers in their vicinity so if you've watched the yelp video the first
thing you should be thinking is all right we need ourselves a geospatial index recall that a geospatial index is
basically an efficient data structure that allows us to do geographic queries to find basically some sort of entity
within a given radius of a point that doesn't require using some sort of 2d scan of an entire range of points over a
2d coordinate plane that a traditional database might support so this way using something like a quad
tree or a geohash we can really easily find all of the drivers in our vicinity
however in this case that's not enough like i mentioned one of the things that we want to be able to do is make sure
that none of our geospatial indexes are a getting overpopulated and b that
they're remaining up to date because drivers and users are constantly in motion so we're going to have to make
some modifications from the yelp problem the first thing as far as i'm concerned is making sure that our geospatial index
is up to date and that the driver basically has the right entry for its geohash or something like that
so within that here's kind of what i'm going to propose we know that we want these really low latency queries we know
that every three seconds we're going to have to be updating our index and additionally we know that the actual
durability of the data is not as important right like at the end of the day it's not the end of the world if you
know our server went down and some of the locations of our drivers were lost because eventually they would
just reconnect to the app and could you know connect to say a different partition of our geospatial index or a
different replica but with all these things in mind it seems like it's a very natural choice to use a
memory based solution for our geospatial index and as a result of it being in
memory and just also so that we don't overload some servers what we want to be able to do is basically partition these
all in a very complete and smart way so like i mentioned in the yelp video a
supernatural way to partition something like a quad tree or a geohash is to basically partition by the
bounding boxes themselves however obviously some bounding boxes are going to have more points more cars more
riders than others and so we want to kind of do this in a dynamic way where
say one node in our redis cluster might have this huge bounding box and another node might have a very small bounding
box if that bounding box is corresponding to say downtown manhattan on a saturday night
when a ton of people are going out and want ubers back because they're too drunk to take the subway so
with this in mind a dynamic basically partitioning schema can be really useful in order to split up the data so that
none of our servers get overloaded we can do this using some sort of consistent hashing schema and either a
coordination service like zookeeper or using a gossip protocol and it seems like
in real life i think uber does actually use a gossip protocol here to keep things very fast and then basically
forwards all the writes from node to node in that redis cluster but that's not overly important for the moment so
anyways now we basically have all of these redis nodes where they are sharded
by these geohashes or these bounding boxes that correspond to 2d locations on the map
basically what we can do is say as a driver or a user is moving around every three seconds we're going to push their
latitude longitude location to the right geohash node so
basically we're going to have to use this partitioning schema and make sure that the record for a corresponding user
or driver is in the right node in that redis cluster and also if they were to
leave a given geohash area such that they should no longer be held on that node we have to make sure that that
entry is also being removed from the node it's important as well like i mentioned
that a lot of these locations the most updated ones are being propagated to users in real time so what we're going
to do is basically once that user requests i want 15 drivers close to me
right now because i'm about to request a ride the second they do that that redis node starts sending them some sort of
push notification through something like a websocket which can be done in real time or a server sent event or even a
long poll so that they can see the actively changing lat lawn of all those drivers
so that's kind of how we would deal with the real time aspect of this additionally in order to kind of help
with matching a given user with the driver once you've kind of figured out
which redis cluster or redis node to pull a driver from you can basically internally
sort those through something like their you know number of stars or their rating score in the past and that way you can
pick which driver a user should be matched with another point which i should quickly touch upon even though
it's not overly relevant which you guys might find cool is that in reality uber doesn't actually use quad trees
nor does it use geohashes instead they've created their own system known as h3 which basically partitions the
world into hexagons because they find that it's a little bit more accurate in terms of doing these radius calculations
and you can really nicely split the world into these hexagons while maintaining this hexagonal structure
that kind of resembles a circle and as a result it more accurately reflects basically a radius query when you're
just trying to get all the points within one hexagon again that's a huge aside and i'll put an image up of it just so
you guys can see but i thought that was kind of a cool thing to show okay with all of that out of the way we can
basically say all right we've covered the part of uber that is very similar to yelp right we've dealt with our
geospatial index we've said to ourselves here's how we would find all of the nearby drivers
which is again very similar to yelp we've said here is how we would update all of the locations of both drivers and
riders which you can do just using a websocket to the proper redis node but now we have to kind of address the
new unique part of the problem and this is something that both uber and google maps share in common which is once we
know basically the source and the destination of a given ride how is it that we're actually going to figure out

the route to take between them so before we get into this let's talk about some sort of data structures and
algorithms concepts that you guys may remember or you may not remember as much
if you have a graph of a bunch of nodes and edges and in this case the nodes would be representing basically a bunch
of intersections on you know our world and our road system and every edge would be representing a road and we want to
find the shortest path there are a few algorithms you can use to do this the first one is dijkstra's algorithm where
dijkstra's is basically a breath first search where you gradually expand out from your
starting point into a set known as the frontier set and then using basically some sort of
dynamic programming every single node in the frontier set you gradually calculate the shortest path to the node in the
frontier set and then you can create a new frontier set from the old frontier set where you get the shortest path for
each of the nodes in the new frontier set so that's dijkstra's and optimization on
this is known as a star which i personally don't even know too well but
i do know that it's certainly a faster algorithm and you guys can feel free to look into that it's not as relevant for
this problem but the issue with both dijkstra's and a star is that if we're trying to find the
shortest paths between two nodes in an actual graph of you know our earth with
all the roads and the possibilities and the interstates that there are algorithmically this can take a really
long time and it's not really good if that computation takes a while because obviously the end
user expects it to be very fast they want to basically say hey get me the directions and return them very quickly
and give me a time estimate of how long this route is going to take so what can we actually do in order to
kind of speed up this process well the answer as with most things when you want to speed up is some form of
caching and the algorithm that i'm going to be talking about in particular is known as contraction hierarchies and
it's relatively new but it seems to already be in use for a lot of basically gps systems both in google and other
places and the general premise is as follows like i said running dijkstra's on the
actual world map is way too expensive there's too much to consider and you're basically running this huge
breadth-first algorithm over a gigantic search space and there's really no time for that however what we
could do instead is basically split our graph up like so we know that generally speaking the
majority of nodes in our graph are pretty much useless to us right like the majority of nodes are going to be on
back roads that we don't care about because the truth of the matter is when you're driving you generally want to be
taking highways from place to place until you get off those highways so what can we actually do well we can basically
take our original graph of the entire world which you know i guess google has or someone else might
but the whole point is that you're going to start removing nodes from that graph
and this is known as a node contraction so basically any node that isn't very
relevant and we know it's not very relevant because basically we've deemed it unimportant and i'll discuss a little
bit more about what that means we're going to remove it from the graph so basically we're going to do the
following let's imagine we want to remove a node v from the graph we can do so but we also can't take away
any of the roads that go through it because well let's say i need to go through v to get from point u to point w
i need to make sure that there is still a route from u to w and we know how long it takes in order to get there so
what we can do is the following if we're removing a node let's say it's v and we have two nodes u and w and the
shortest path from u to w is through v when we remove v from our graph we can
actually just create a new edge from u to w which reflects the actual distance from
u to w as if we had gone through point v but now the node for point v is no longer there and i'll make sure to do
visual representations of all of this so basically we've now created the shortcut edge and what these shortcut
edges allow us to do are to remove a bunch of nodes from our graph itself
such that we can pre-compute the distances between points that we deem to be more relevant
now how can we basically decide what nodes they are that we should be removing first well one good heuristic
to use here is to basically remove all the nodes that require adding the fewest
shortcut edges meaning you know say i were to remove a point on a major highway
then i would have to add so many shortcut edges to our graph because there are so many fast routes that go
through that highway however if i were to remove a node in some farmland in kansas
it's not really going to add a lot of shortcut edges because it just happens to be the case that that was a back road
and it didn't really get us to a lot of places efficiently so with that in mind we basically want to be removing all of
these relatively irrelevant nodes such as these back roads and alleyways and
things like that that don't really get us places quickly and then what's going to be left over after a few iterations
of this are these pretty much compacted graphs where there are many fewer nodes and
basically the majority of those nodes remaining are on more popular roads and highways and things like that
and so now what we've done effectively is we've segmented our graph in a way that we can really quickly calculate the
distance between very popular points and once we have those pre-computed distances between popular points
you know we can get a popular point that's close to both our source node and our destination node and then from there
we run a very limited dijkstra search between the source to the first popular point that we care about and the
destination and the second popular point that we care about and so by doing so we're able to
basically pre-compute all of this direction and eta information and as a result of that we can make this query
basically 15 times faster than what it was and their actual numbers to support that basically a second and a half to
100 milliseconds is the number that i keep seeing and this is basically the technique that
all these companies use because by you know pre-computing all of these numbers
we can effectively cut down on our search space by a ton and even though it
is the case that we won't necessarily have the fastest route because to some extent we still have to do some
approximations in this whole precomputation method with compaction hierarchy we're going to have a very
generally good route which is going to make it very quick in terms of the

latency of our queries the final thing that i would like to note is and this is you know back to kind of
being unique to uber is there is a ton of background processing that has to go
on both during the ride and more importantly after the ride that tends to be things like collecting
payments um collecting data about the actual location of the driver during the
ride so for example you're gonna get a ton of gps signals from that car during the
ride itself so that you can basically show the rider the route that the driver took after the fact
and that right there probably involves some sort of stream processing where all of those gps signals are
ingested into a kafka stream which are eventually hitting some sort of processor node which knows once it's
received all of the gps signals and then it has to actually go and create a route along a gps map and
that's actually not as easy as you might think because the gps signals themselves are a bit imprecise right like they
might be you know 10 feet 20 feet off the road so one way that i've seen
that uber will actually account for something like this is by using something known as a hidden markov model
where basically it's saying okay considering all of our gps signals and considering how likely we are for a
given road to basically get a gps signal that we've gotten and considering the possibility
that you're going to go from road a to road b which is known as a transition probability and then there's also the
emission probabilities of saying what's the likelihood that we get you know the gps signal that we see here
from road a you can basically map out this hidden markov model and then run a dynamic
programming algorithm known as the viterbi algorithm in order to basically quickly compute the solution of what is
the most likely route the driver took from the source to the destination of the rider
i know that was kind of a very high level overview of what's actually going on and the reason i don't go into it
further is because truthfully i just don't really think it's that necessary for the systems design interviews
something like markov models and the viterbi algorithm is more relevant for something like machine learning which i personally think is
super cool but i
also want to generally keep these videos to around the half hour mark and i don't think it'll make anyone happy if i spent
all day talking but i am going to link a bunch of articles below so that for those of you who are interested or just
want to learn more about how this stuff works you can alright apologies as i passed
out before i could do the last part of the video but as per usual we will be talking about the diagram this back end
particular is quite large but not large enough such that it's bigger than my back end but like i said still pretty
big so anyways let's get into it and diagnose this thing and you know go
ahead and go through it so first of all we have our client which is going to hit a load balancer as per
usual this is going to be a single point of failure and as such we can run a second load balancer in either active
active or active passive configuration as we said one of our functional requirements is that at first the client
is going to basically query the surrounding area to see all possible drivers that could potentially pick them
up as well as getting real-time updates as to their latitude and longitude as it changes
as a result they're going to first hit this geo service which is going to be basically horizontally
scaled out and afterwards hit our redis cluster which we described earlier which is basically
our cluster of in-memory databases that are sharded by geohashes so you know we
can dynamically shard them based on the size of the geohash but ultimately the point is a bunch of cars within a given
you know box or geo hash bound are going to be on the same redis node and as a result we can quickly return the results
back to the client as well as all of their latitude and longitudes and we can do this via a websocket because as we
said we want to be updating all of these positions in real time which insinuates that we want some sort of real-time push
technology like a websocket as opposed to typical polling next after that once the client has
basically seen all of the nearby drivers it's going to actually request a ride and this is done via the dispatch
service the dispatch service is yet again going to hit this redis cluster and in particular it's going to hit the
node that the client is a part of or perhaps an even bigger node or multiple of those redis nodes if we have to look
at multiple bounding boxes because there aren't too many drivers available and then once it does so the dispatch
service is basically going to look at all of the possible drivers nearby in the redis cluster based on that
geospatial index and select one or possibly even more than one based on the
actual popularity of the driver and we can do that using you know say something like the number of stars
afterwards one of the drivers on their own client app is basically going to say i accept this ride and then both the
client and the driver are alerted of the fact that they have now been paired up after the actual pairing itself the
client now needs to see the route that they'll be expecting to take and this is important because the route service is not only going to
basically fetch the
directions for the driver but additionally and more importantly it is going to give an estimate of the pricing
because it will give our distance so the route service will first check a route cache um to basically see if the route
is in there and if not it can go ahead and actually consult the database now if
you'll see here as opposed to using something like a relational database or you know in particular a nosql database
like a wide store or a key value store i've chosen to go with a graph database and there are a couple of reasons for
this the first is that as opposed to just keeping all of our graph data in memory and using a bunch of pointers we
know that you know the graph of the world is really large and there's going to be a ton of data in there so it seems
pretty natural to use something like a distributed graph database in order to basically hold all of that data
and then obviously for the popular routes we want those to be stored in the cache and when i say popular routes you
know not every single route is going to be in the cache but perhaps we could store kind of the common parts of a
route so for example like i mentioned because we're using these contraction hierarchies what we're basically going
to do for a given source node of a ride and a given target node of a ride is assuming that the source and target
themselves aren't in that contracted graph of only the popular nodes we can
basically run we can basically run a first search from both the source and target node until we

find one of the nodes in that contracted graph which is going to have only the popular nodes in our database and then
basically we can cache only the popular route so basically the routes from one node in the contracted graph to the
other node in the contracted graph and as a result of that we can basically cache these really popular components of
a lot of routes but not necessarily the entire route and that's going to help us a lot another reason that using a graph
database here is actually going to help a lot with neo4j in particular is that not only are things like breadth first
search depth first search and distress algorithm built into graph databases but more importantly graph traversals are
actually faster the reason for this is that graph databases will actually have pointers to the location of connected
nodes on disk this is really useful because as opposed to basically non-native graph databases which are
basically just an abstraction over a typical database index what is good about a graph database that
is native is that you're using disk pointers to go from node to node if you didn't do that you would basically have
to say okay node a has an edge to node b it would say id6 now i have to go
through my index and actually find the node with id6 and so every single time you traverse from node to node you have
to go through an index to find the node on the other end of the edge and what that means is that because an index
basically is holding all of the vertices in the graph the amount of time it's going to take to
traverse from node to node is going to scale proportionally or rather proportionally but logarithmically to
the size of the index because we can binary search the index so basically if you're using this a non-native graph
database every single time you add a node it's going to increase your graph traversal
times by a factor of log n where n is the number of nodes in the graph so a
native graph database on the other hand is very useful and you can see me talk more about that in my actual graph
database concepts video but like i said having a graph database here is going to be super useful because we have a ton of
graph data and it's going to allow us to perform all sorts of cool searches and traversals over our data
then finally we have our ride data service the ride data service is basically going to be ingesting a bunch
of points and data points that are going to come up while the ride is actually occurring or the trip is occurring so because like
i mentioned a lot of this is going to be gps signals from the car the driver itself we can plop those into a kafka stream
which i'm using kafka here in particular because it's log based and that means those points are going to be durable and
replayable and that is usable very well with our stream processing framework
flink and the reason flank is good here is because flank allows us to basically keep state when doing stream processing
so we can ingest all of these gps data points for a specific ride and we can basically just partition these kafka
streams and flink processors such that for a given ride id all of the gps points are going to the same flink node
and then we can send one more signal when the ride is done through our kafka stream and basically once the flink
processor receives that done signal it can go ahead and cache all the gps signals that it's received and after
receiving the done it can perform the computation that is necessary in order to kind of calculate what root was
actually taken and we can do that using that hidden markov model and viterbi algorithm that i mentioned previously
and then once that calculation is complete once we've basically guessed which route the driver has actually taken we can basically get
the distance
that corresponds to that and a price and we can put all of that information into our trips cassandra database
now the reason i'm using a cassandra database here is because it allows for really quick inserts
we can basically store data at great scale and we don't really care about things like eventual consistency here
because they're all just going to be going on in the background so cassandra acts as a really easy way to just store
all of our trip data and additionally the fact that there's no schema makes it easy to kind of add additional data over
time to our database and then the way that i personally would probably go ahead and shard out our
cassandra database is by using the user id as the primary key for the trip aka the writer id and then similarly
the secondary key or the sort key to be the timestamp and what this will allow us to do is for a given user we can
really easily find all of their rides and also sort them by you know most recent to least recent
um yeah i mean obviously there are some details that i left out such as payment information
and things like sending emails to a rider once their ride is done and same probably goes to the driver but overall
i think this acts as a pretty good overview of things that can be done um you can always do some more like stream
processing as things get added to the trips database and then connect that to something like a notification service or
an email service but overall i feel like this is a pretty comprehensive diagram of how something like uber might
actually work if we're dealing with google maps obviously the majority of that computation is going to be the route
service component where we're basically taking this huge graph database caching a bunch of kind of the popular
components of many routes and then using something like contraction hierarchies in order to simplify our calculation a
lot and then combining that with graph traversal algorithms such as dijkstra's or a-star in order to get our results



Ticket Master:

Capacity Estimates

500 cities, 10 theaters per city,
2000 seats per theater, 2 shows per day

50 bytes per row of booking data

500 * 10 * 2000 * 2 * 50 bytes = 1gb/day

API Design

getCities()

getMovies(cityId)

getTheaters(cityId, movieId)

getShowTimes(theaterId)

bookSeats(showId, seatIds)

Database Schema

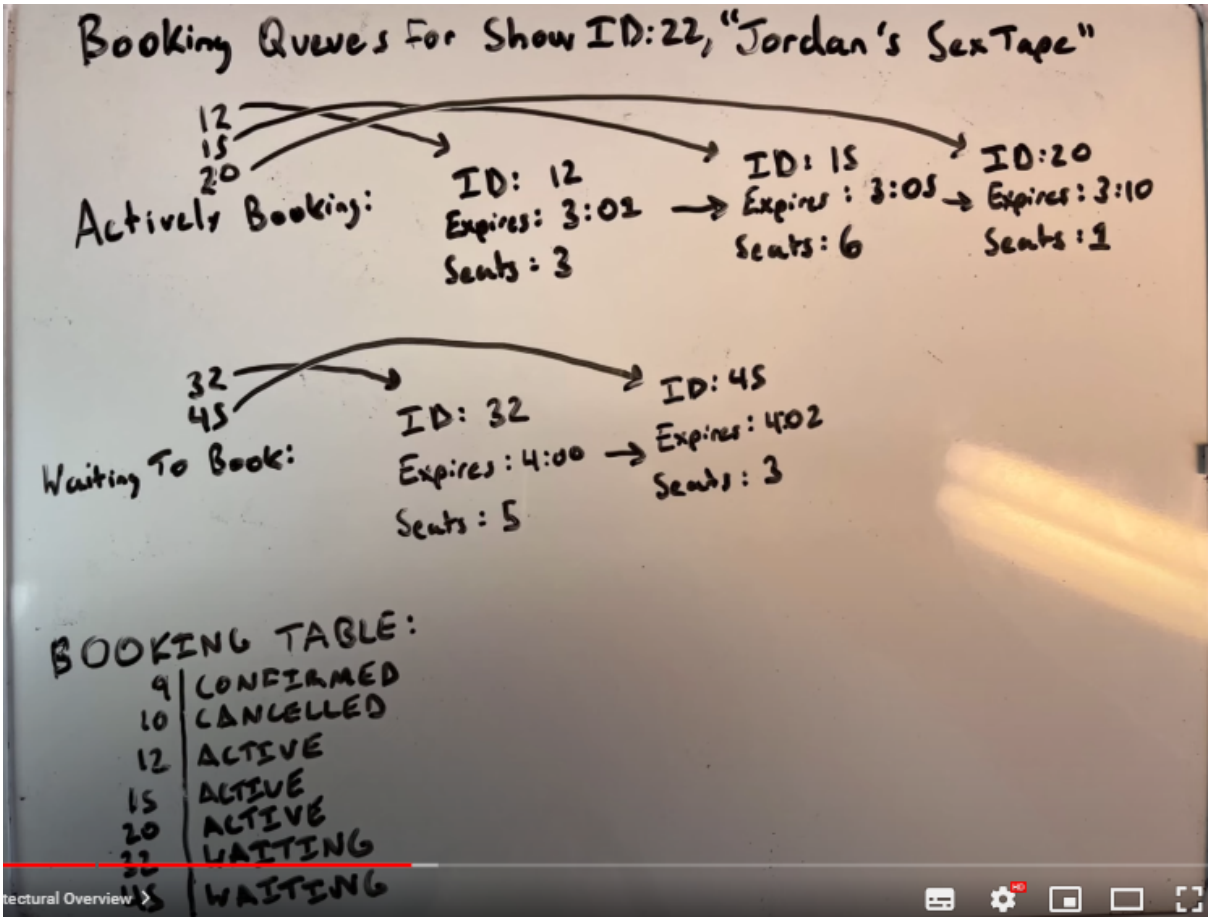
Cities (id, name)

Movies (id, name, cityIds)

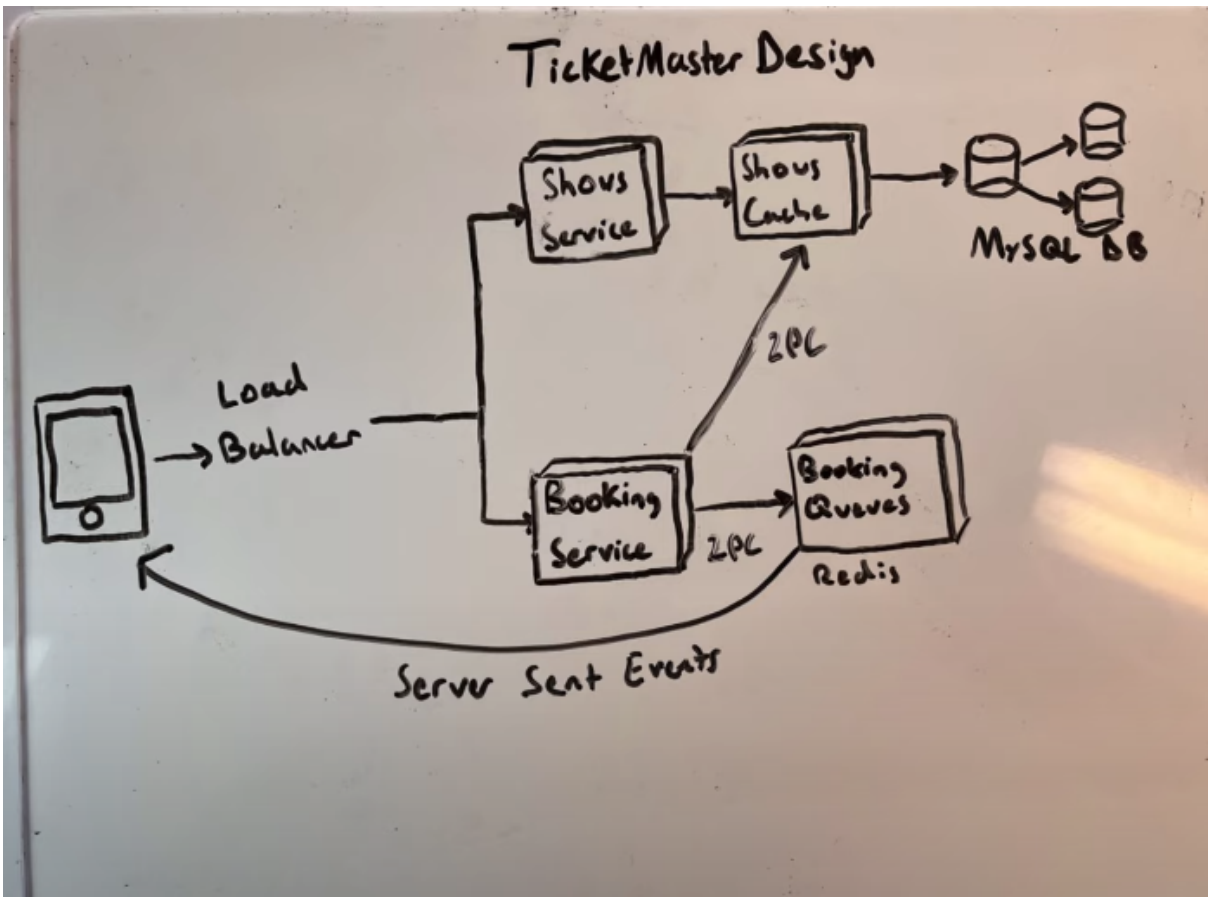
Theaters (id, name, address, cityId)

MovieShow (movieId, TheaterId, time)

MovieSeats (id, showId, name, status)



architectural overview



all right so now that we've kind of laid out the background for this problem let's talk about some considerations that we have to think about and some of the design choices that we're going to make for how we can properly implement ticketmaster so the first thing is that if you guys haven't thought about it yet i already kind of hinted at this but i basically said it's very important that when a user chooses a certain number of seats it is important that they make sure to book all of those seats and it's important that if they don't book all of those seats they're not getting a partial amount they're getting none that should imply atomicity to you what that means is that we want a database supporting acid transactions in the sense that basically either all of these rights go through or none of them do additionally what acidity is going to help us do here is make sure that we don't have any issues with concurrency imagine that two users both try and book the same three seats at once and you know it's me and my friend and i get one of the three seats and he gets the other two that wouldn't be good we want to make sure that basically somebody outright wins and somebody outright loses and that is kind of defined by serializability there's no interleaving of those three operations either they all go through together and they all either happen before or after someone else's operation so as a result of that it's very important that we use a database that supports acid properties and generally that's going to be a relational database like mysql or postgres so we'll discuss about how we can go ahead and shard and replicate that data in a little bit but that's a very important thing to note because using something like a nosql database here that doesn't support transactions would give us a huge risk of dealing with all of these issues with concurrency where we have all of these partial bookings and then nobody ends up happy okay additionally another thing i want to basically go into a little bit is kind of some of the functional requirements themselves because they're a little ambiguous

so i've mentioned that what really happens in at least this problem is that the user basically says i want you know three seats i want four seats and then our service is going to say okay well there are three seats left so you can go ahead and try and book those or you know there are three seats left but someone else is trying to currently book those so go on the waiting list well in reality people don't really book that way the way they book is you know they want a certain few seats and then i would imagine that the service would basically say well these are either available or they aren't or you know maybe someone else is trying to book them but if that doesn't work out you can book them yourself the issue there is then what would need to happen is for every single seat we would need to basically keep a list of all the users waiting for it and keep track of who's waiting for every single seat and this gets really complicated because what it implies is that you know i could be waiting for seats four five and six and then i have user one currently booking c4 user two currently booking c5 and user three currently booking for seat six and what might end up happening is that now i'm waiting on three people in order to basically get my own booking working and say you know user one drops out so now i can get c4 but now i'm you know still waiting some unclear amount of time for seats five and six in order to kind of figure that all out so i don't know i i'm not exactly sure what ticketmaster actually does in terms of this whole waiting list queuing thing but i guess if we wanted to do kind of seat level waiting list that would be very complicated maybe what would be a little bit more practical is you know for things like sports scams where there are a lot of different prices of tickets and you know basically if i'm sitting in the upper deck it's going to be a lot less expensive than if i was sitting on like the field level perhaps what you would do is you would split the stadium into sections and then you would act like each each section is its own show right where you know all of basically the waiting is for a given section as opposed to the entire event as a whole so i guess that would be kind of a clever solution but then you know you're limiting users to only being able to book within one section at a time which shouldn't be a huge deal because generally speaking if you're buying tickets you want to be sitting next to you know the people you're booking with so i just wanted to consider that but you know that kind of opens the floor for how we're actually going to solve this problem so basically like i said for every single potential booking and that means one where either you have seats reserved or you don't and you're waiting for them we have to keep track of that and the reasoning for this is let's say i'm waiting on seats that someone else currently has reserved but they might drop out it might be the case that other people are also waiting on those same seats and we've basically said we want to do whoever kind of becomes the next person to get access to trying to book those seats in a fair way which means we need to implement a queue or a linked list of sorts where we're keeping track of who basically said i want these seats first and as a result when they come available it's the person who came in first that is going to get the first access to trying to book them again so how can we actually do this well we've already mentioned that we have this relational database of all of our seats but in addition to that we should probably have a relational table called bookings so every single time that someone requests seats whether they're actively trying to reserve them and have them reserved but haven't finished the payment yet or they're in this kind of waiting list for them they should be assigned some sort of booking id and what we can do with this booking id is we can put all of these booking ids somewhere in memory and maintain a data structure that's going to help us keep track of them such that we can perform the handover if someone were to kind of back out of trying to perform a booking or we can basically remove them from the waiting pool if another booking was successful and now their booking becomes invalid so let's talk about two different types of memory structures that we can use in order to maintain basically all of the people who are

actively trying to book something and all of the people who are waiting for seats to become available so let's assume we're going to basically be using two linked hash lists so i should kind of basically say that's a linked list where there's also a hash map pointing to each element of the linked list so why is this useful well the hashmap is going to point to each element of the linked list by booking id and then the linked list itself is just going to order all of the elements inside of it and i'll explain why that's useful in a second so let's imagine that we have uh you know a movie theater and it has a show for i don't know batman so batman is going to open up and we have a ton of users and they're all going to try and get seats and now let's say what we have is 10 users are able to basically make seat selections and get those seats reserved so now they all have five minutes to make their booking all 10 of those people are going to go in one linked hash list so we have all of their booking ids in this linked list and now the following is going to happen we basically have a process saying i'm only going to check the head of the linked list but because the linked list is ordered i know that if the head is expired i can start checking subsequent elements i you only have to check the first element for being expired because you know that all subsequent elements in the linked list were added on later than the first element so the linked list makes it really easy to basically check for the expiration of certain elements additionally what the linked list or the hash part of the linked list helps us do is say okay let's say one of these users was successfully able to book they got their payment information in those seats are now reserved so now we're going to make a call to our database using a transaction so we make sure that all the seats they selected are given to them once that happens the seats are given to them and then the booking is confirmed and we can go remove that booking id from the linked list and we can do so by finding the location of that booking in the linked list using the hashmap component of it so you basically get the value corresponding to the booking id and so that way we can really quickly remove that value from the hashmap and update the number of seats left once we do that we can say oh well let's take a look at our waiting linked hash lists let's say there used to be 10 seats left but now there are only five seats left all the people in our waiting hash list who wanted six or more seats we can iterate through that list and remove them out additionally what would happen if someone from the actively booking linked list were to cancel their reservation well once that happened we would use the hash map again to basically remove that node from the active linked list and then we would pick a node from the head of the waiting linked list to put at the end of our active waiting link list again we're running processes such that we're making sure to basically expire elements of both linked lists as the actual time expiration period passes but i'll make sure to have this all written out on a whiteboard so that it's easier to visualize for you guys as for clients on the wait link list obviously we don't just want to leave them hanging and never alert them and make them just like constantly refresh their page to see if they can finally make a booking so what we could do is use some sort of real-time pull technology such as web sockets or long polling or server-side events i think long polling or server-side events are probably the better options here just because it's not a bi-directional communication so maybe something like server-side events would be good where you're just sending basically all the users in the waitlist what position they're currently on and also things like how many seats are left in the show to potentially be booked so they can get a sense of whether things are looking good for them or not another thing to quickly note is that any time we're dealing with memory there are a lot of you know considerations to think about especially as it pertains to race conditions and concurrency um even though i've mentioned in a past video that i believe redis is single threaded if we aren't using redis here and say we're just running a second process on our node holding our database we have to make sure that we're using locks on our linked list

because otherwise we can have concurrency bugs where by using that read modify right cycle to write to the tail of a linked list you end up not actually writing to the tail node but writing to say the node before the tail node because you read the old tail node and then tried to make your right so it's very important that you know when dealing with in-memory data structures if you have multiple threads on say a server that are trying to deal with all these bookings that you're not having concurrency issues and that you're locking where need be in order to make sure that certain entries of that linked list aren't basically being overwritten another consideration to mention is the following like i mentioned you could use redis as the place where you're holding these two linked lists combined with hashmaps however it may be a lot easier to just put this as a separate thread or a separate program on your actual database node and here's my reasoning for saying so if you have this linked list that needs to basically be coordinated with all of the bookings in your database and they're on different nodes then you need something like a two-phase commit protocol to ensure that they are basically coordinated over the network because otherwise what would happen is a booking might be successful but then you never remove that booking from your active linked list and as a result of that what's going to happen is no one from the basically the waiting linked list will be notified that there are fewer seats now or that you know they're no longer eligible to actually get those seats and the same applies for if someone were to cancel their booking no one in the waiting linked list would basically be promoted so that's why you basically need this two-phase commit if they're on different nodes and have to communicate over the network is because you want them to stay in sync however two-phase commit is a very slow thing it requires a lot of network coordination and overhead and as a result of that it would probably be better if these two processes were at least on the same node additionally as far as fault tolerance goes you could definitely replicate both of these linked lists to other nodes or other redis instances or something like that in a single leader replication type of configuration and then furthermore as far as sharding goes it makes a lot of sense naturally speaking to basically shard everything based on the actual show id itself so you want to shard in a manner such that all of the linked lists corresponding to one given show or one given event is going to be on the same note there are ways you could share it otherwise such as you know like the same city or the same movie theater or the same movie itself but that doesn't really make sense because you want it to be the case that that linked list is on one machine and that makes it easier to get everything expired coordinate all those bookings together take advantage of better cash locality so you know it's important to use some sort of consistent hashing schema to basically shard based on the event id and then finally speaking of caching one thing that you could do to actually speed things up is i mentioned that it's possible that the linked lists and the database containing bookings are going to be on different nodes however if that's the case what you could do is use a sort of write-back cache on the node containing the linked lists such that it's very easy to calculate the number of remaining seats for a given event and that way you don't have to pull from the database every single time and say oh how many seats are left and you know are all of these waiting users still valid or do we have to remove them from the waiting link list so something like a right back cache could work well here or it doesn't even necessarily have to be right back but the point is by caching the number of remaining seats we can really quickly um figure out who should still be in the waiting list and who should not be anyways i hope that made sense to you guys i'm now going to draw out a diagram alrighty well unlike my ex-girlfriend this diagram is actually pretty straightforward when you try to figure out what's going on so let's go through it real quick as you can see we have a client i've drawn out the phone this time because i was feeling artistic which hits a load bouncer can be in passive

active or active active configuration for some redundancy and then first of all we're going to have our show service this is useful for all of our get endpoints that i previously mentioned and those can be run through a cache because a lot of these results are just going to be duplicates it's going to be the case that me and anyone lives near me is going to get a similar result from this so having a cache in front of this and possibly even a cdn for something like static content or images that correspond to movies or theaters is going to speed things up quite a bit obviously that's going to hit all of our mysql databases which i've covered those earlier in terms of the schemas that we're holding that's things like the cities table movies table movie shows table movie seats table bookings table etc etc and we can just use single leader replication for those because there's going to be relatively infrequent rights and what we really care about is more the consistency of our data and as a result just having a single place to write to and possibly some sharding is going to make our life easier than using any sort of leaderless replication schema on the other end of our service we have our booking service now this is where as of now i've currently got some sort of two-phase commit protocol where every single time we're going to be either canceling or confirming a booking we have a two-phase commit to our booking queues which are held in redis and also our either right-back shows cache or a mysql database you can choose what type of cache you want but either way as we know like i mentioned two-phase commit is pretty inefficient and if we really want to be optimizing for speed here even though ticketmaster probably doesn't need it that much because there's still just like a limited amount of shows that people can you know register for and also a limited amount of seats per show then what we could do to optimize this is run basically a separate queue process on some other one of our servers probably our database servers and that way we wouldn't have to coordinate over the network itself finally once we have all of our information in these queues we can go ahead and use a real-time protocol such as servers and events to keep all of our clients who are waiting for seats to open up and basically alert them where in the queue they are and that that'll get them a sense of whether or not they basically need to cancel whether or not they're waiting or they should stay in that queue um yeah so as we know there's going to be some sort of process basically going through the queues and expiring things as we go and when that happens again there's going to need to be some sort of two-phase commit protocol where we're also updating our mysql database at the same time so there's definitely a lot of coordination that needs to happen the important amount of sharding that needs to be done should really just be sharded by like show id or event id and that way we can keep all elements of the cues in the same redis instance and similarly all potential bookings for a given show on the same my sequel node and all of basically the seats for a given show on the same mysql node and that's good so we don't have to do cross-partition transactions when we do transactions to book multiple seats at the same time all right guys i hope this video was helpful

Google docs design -

Functional Requirements

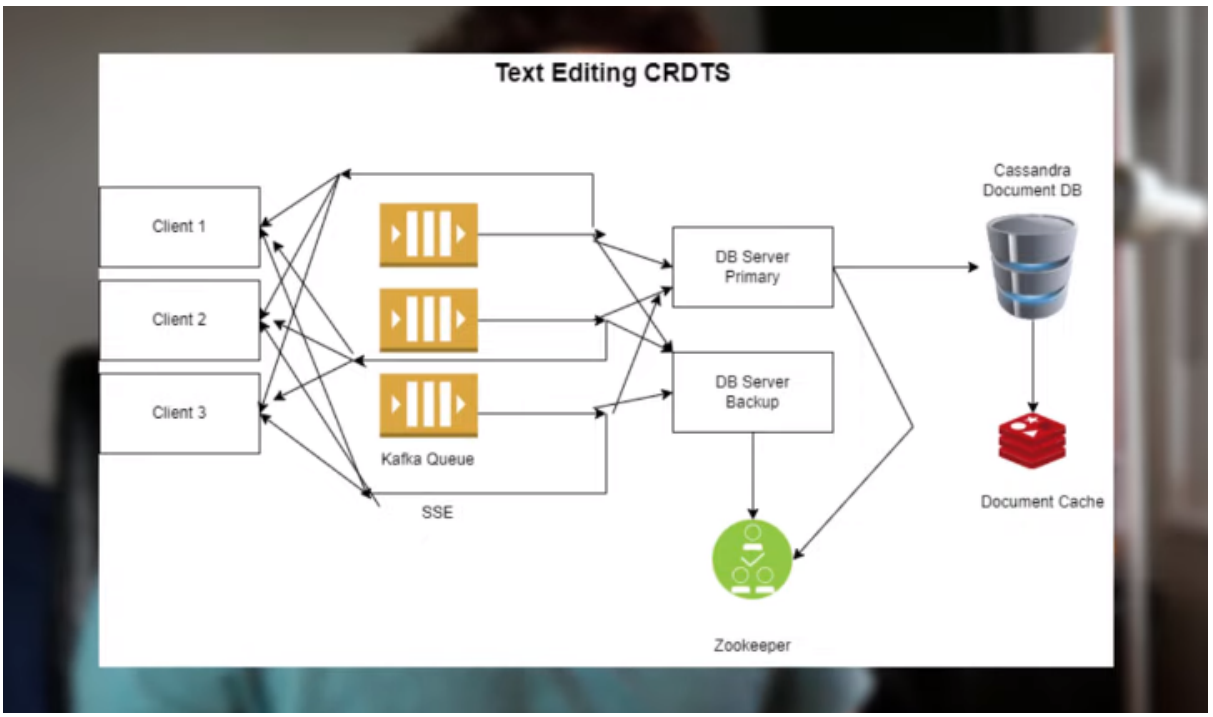
- Edit text documents, support concurrent editing (both online and offline)
- All clients eventually see the same version of the document

Capacity Estimates

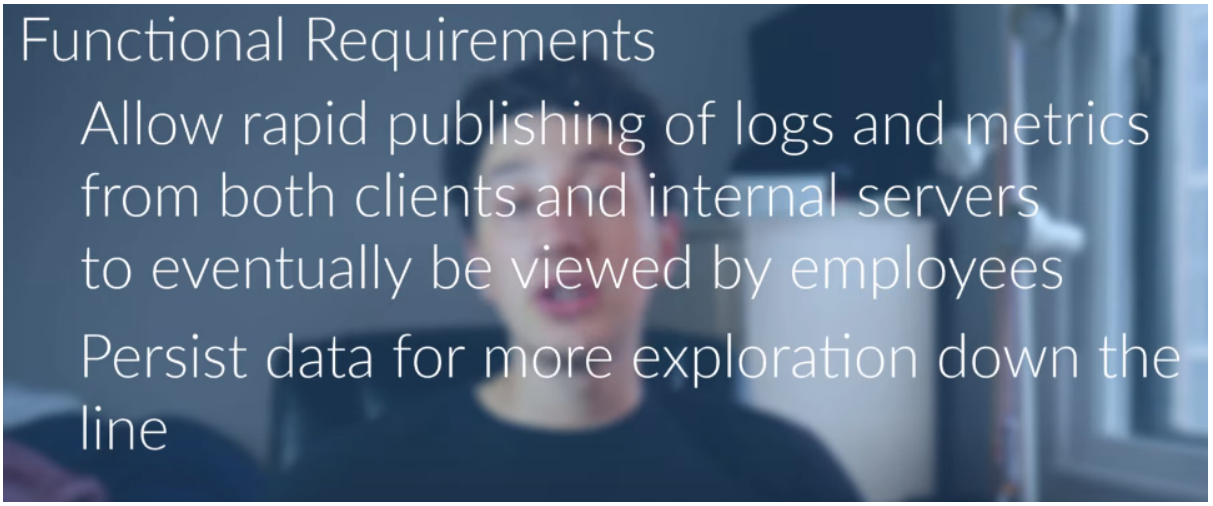
- 1 Billion documents per day
- 1kb per document on average
- 1 Tb of storage per day

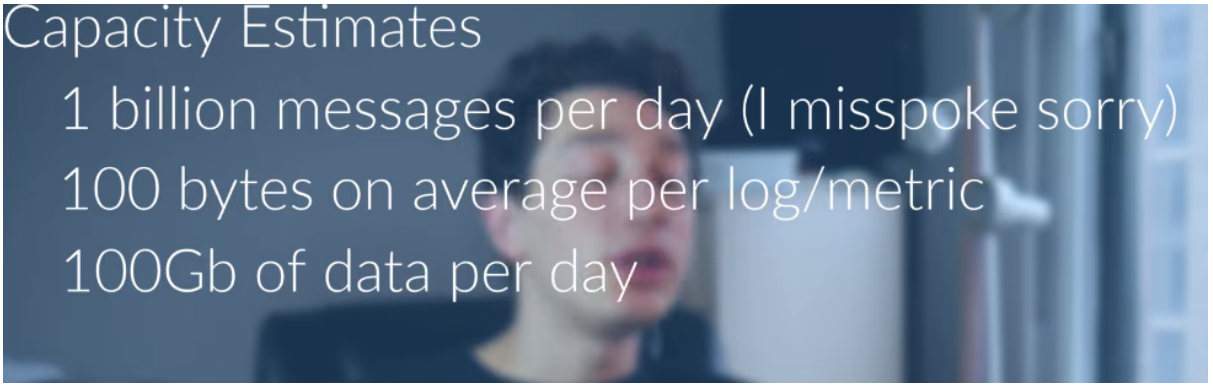
API Design

- FetchDocument(docId)
- InsertChar(docId, position, char)



Distributed Metrics/Logging

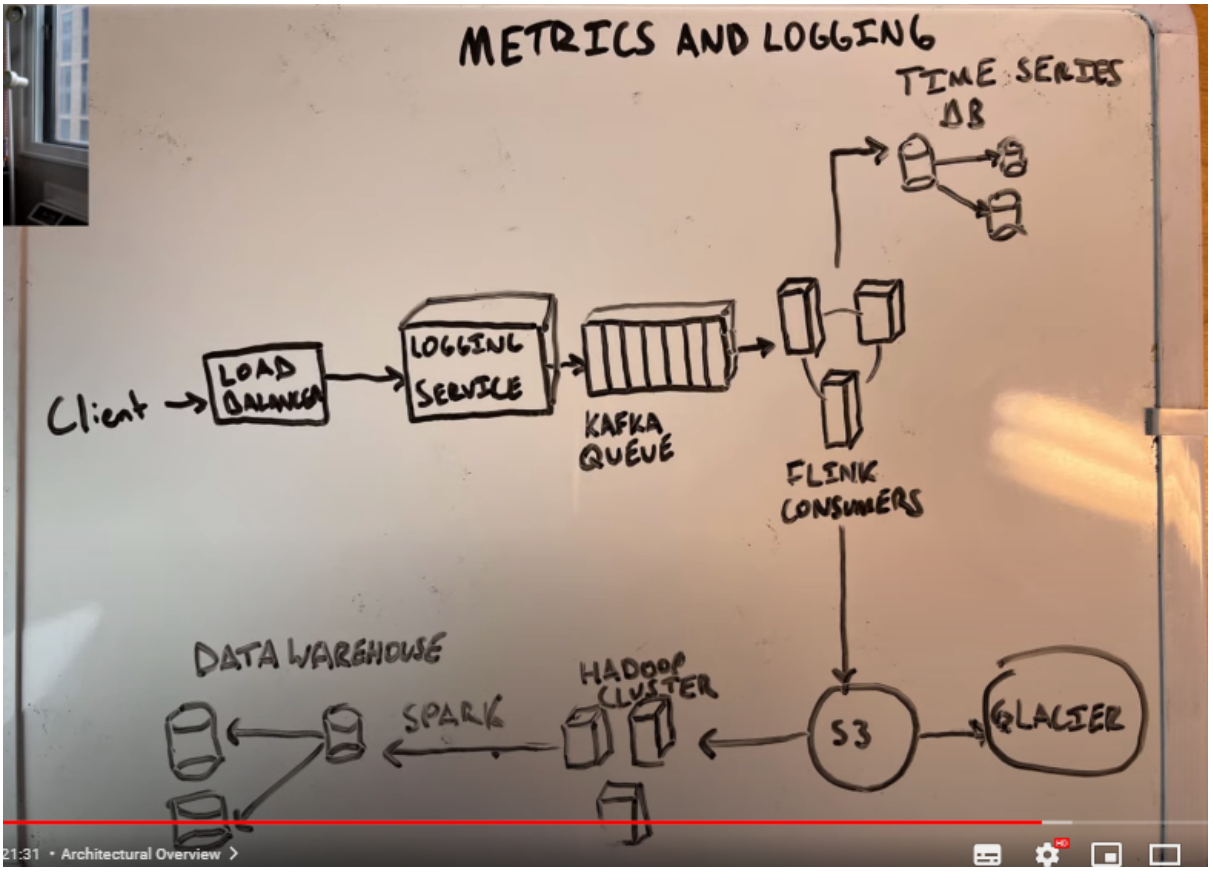




API Endpoints

sendMetric(source, payload, timestamp)

fetchDataset(seriesId, timeRange)



Okay, I'll mention some API endpoints here, but truthfully the majority of this is probably going to be done over something like an RPC call, and I'll discuss the reason for that later. It kind of has to do more with all that compression stuff, but you know, if we're just talking about ways that our client might interact with our application servers, there's probably going to basically be one to actually go ahead and send the message. So that's going to have like a payload, a timestamp, maybe a topic if you wanted to go to a certain place or another.

And then also, a second endpoint could just be to fetch a given data set. You know, if I'm an internal user, or rather, if I'm an employee for the company and I want to basically go ahead and load a data set, I'm going to have to tell some sort of service, "Hey, fetch this for me," and then it'll return some sort of visualization or the data points within that set. But again, we'll talk about that all more later.

All right, now let's get straight into designing. The reason I'm skipping over the database schema is that it's kind of open-ended for a problem like this, right? You know, the database

schema itself isn't really clear about the exact data we're storing. So, I'm going to talk about the types of databases and technologies we'd use rather than the specific data.

Let's start with an outside view into our system and then break things down further. We have clients and different servers throughout our system sending messages. In theory, we could just put each message into a database as it's generated, but it's not that simple. Each message requires processing, both in real-time and possibly later as batch processing.

Where can we put our message data to enable analytics? Generally, for generating messages from various sources and handling them consistently, we'd use a message broker. The question arises: in-memory or log-based message broker? While in-memory is great for stateless consumers, a log-based broker like Kafka is better for durability and replaying past messages.

Consumer nodes hold various states. For instance, stream enrichment, where a consumer joins messages with relevant data. Stream-stream joins are possible too. Also, time-based aggregations like windows (tumbling, hopping, sliding) to summarize metrics over time.

For stream processing, tools like Flink and Spark Streaming can be used. Flink is better for real-time, while Spark is more for mini-batch processing. Data can go to a time series database (optimized for metrics) or an S3-based data lake for less formatted data.

From S3, data can be moved to a Hadoop cluster to run Spark jobs. Spark excels in batch processing. Data can then be placed into a data warehouse for analytics. Column-oriented storage is useful, compressing data with tools like Parquet.

Eventually, archived data can go to slower storage solutions like AWS Glacier to cut costs.

Keep in mind that the advantages of using an in-memory based message broker are that when you kind of just want to take all these messages and send them to consumers that are stateless because you just want them to run some job on it and then you know once the job is done all is good. That's kind of when an in-memory broker is really good. However, for our case, it seems like a log-based message broker may be a little bit better, and there are a couple of reasons for this. One of which is the durability of messages because everything is persisted to disk. We can be sure that none of our metrics or number logs that we have are getting lost, and our data can be as accurate as possible.

In addition to that, because of the fact that we're going to have all these consumer nodes which are doing different things with our messages, and these consumer nodes are often going to be maintaining some sort of internal state based on the messages that we've seen, having a message broker that allows us to replay messages that were in the past will allow us to:

a) Add new types of consumers that can do new types of processing on the data.

b) Basically, get their state back if they were to crash because then they could keep track of the last state checkpoint that they have, which was probably going to be an S3 or something, and then they can replay the messages after that point.

So, it seems that even though the throughput in theory is a bit less, the log-based message broker is going to be a little bit better for our design here. And that's something that I see actual companies doing. For example, I watched a Pinterest video to see how they did their distributed logging in order to help make this video, and they're very much using Kafka as opposed to something like RabbitMQ.

Okay, so I keep mentioning that we're going to have all these consumer nodes and that they're holding some sort of state. But what are some examples of state that they may be holding, and why might they have it? Well, the first thing is stream enrichment. A lot of these messages are going to have things like a user ID, but they're probably not going to include a bunch of relevant data with that user which we may actually want in our logs, so we don't have to query an SQL table every time we see a given log. In terms of stream enrichment, something like either a Flink consumer or a Spark Streaming consumer could be keeping a local copy of a given table on it in order to basically join with every single one of these messages so that it could eventually be placed into a database with more information.

Additionally, as opposed to just doing the stream enrichments, we could even be doing stream-stream joins where we have a given consumer pulling from two different Kafka queues. Basically, two queues of different topics, and then they're pulling from one partition of each of those topics and then they're joining that information together. Again, that can help us make these really insightful logs, and I've mentioned cases of this in my stream processing video.

Another thing that we can do is time-based aggregations. Oftentimes, it's not enough to just take the metric from every single second that we're getting them and say we want five-minute aggregations, ten-minute aggregations, things like that. So, there are three types of windows that I'm going to address here, and here's how the stream processing nodes can do it:

1. **Tumbling Windows**: These are basically fixed length and fixed start time non-overlapping windows. It's basically saying one-minute windows starting at the beginning of every single minute.

And it's really easy to do those. You basically just take the messages as they come in and you know cut off the part that you don't care about and say, "Oh, it's going into this window." And then there's also hopping windows, which is a question of basically: I have, you know, five-minute windows, but I can have one that starts at 12 o'clock and it goes to 12:05. And then I can have one that starts at 12:01 and it goes to 12:06. So, hopping windows can overlap with one another. The way you would create those is by having these tumbling windows which I just mentioned and actually aggregating, you know, say, five one-minute tumbling windows to create all of these hopping windows.

The third type of possible window is a sliding window where even though it has a fixed length, there's no necessarily constraints on the start time or the end time. So, you know, even starting within a half a second or something, I can have this sliding window and then it just lasts five minutes. The way you would do that is by basically maintaining something like a linked list in memory and using one process to add elements to the end of the list and another process to remove elements from the beginning of the list once they've been expired from that sliding window.

Those are some examples of stream processing that we can actually be doing. For more examples of those, I recommend watching my stream processing video.

But let's actually talk about which technology we should use as our stream consumers. One example would be Flink, where the main advantage here is that even though both Flink and Spark Streaming are the two examples I'm going to address, they can do stream processing and maintain state and have reliability by being able to get their state back if, for some reason, it gets lost. Flink is handling messages in real-time, whereas Spark is handling messages in mini-batches.

So, kind of why is one case good for certain scenarios and mini-batching might be better for different scenarios? Well, in terms of handling messages in real-time, it means that, you know, if we're putting those in a database, I'm going to be able to see that message in the database more quickly. On the other hand, if we're handling messages in mini-batches, so say, as a consumer node, I get 10 messages at a time, it means that I can also upload those messages to the database in a mini-batch, and as a result, there's less network overhead in sending all of them.

I think it's probably not really a huge difference which gets used in these cases, but that is kind of a difference to know between the whole real-time stream processing versus semi-real time using mini-batches.

Furthermore, the next kind of piece of the puzzle is where are we actually going to send the data after it goes through these consumer nodes? Well, there are a few options. The first is going to be a time series database. So yeah, obviously, we could just throw it in, you know, Cassandra or something or HBase and call it a day. However, the really useful thing about a time series database is that it is specifically created in order to handle this timestamp data like metrics and logs so that it can be both written and read really quickly.

How do they actually do that? Well, what a lot of time series databases will do is, as opposed to having one huge index over the entire table to both read and write into, they'll actually split out the table, which I've seen called a hyper table, into a bunch of these mini indexes where each index represents one source of logs. So, like sensor one and then the time range for those logs, sensor one January first, and then we would have sensor one January second, and that would be a different index. So, why is this really helpful? Well, the main kind of thing is that all of these writes for you know, a given day are probably only going to be going to a couple of these mini indexes. And as a result, we can cache the entire index in memory and get really, really good write performance and read performance by doing that. Whereas, you know, say we had like a B-tree-based index and split out all over this tree, it

would be hard to kind of cache only the relevant portions for us. It's really useful to split out this hyper table into these smaller indexes.

Another reason that's super useful is a very common functionality or use case in time series databases is to actually delete certain data once it reaches a certain age. The reason that's bad in say LSM tree databases, for example, is that there are no actual deletes on keys in LSM databases. All you can really do is put a tombstone over a key where you just mark it as deleted. And as a result, you have to wait for that SS table compaction to go down before the key is actually removed. But because we have these indexes and they're all much smaller, once an index representing a certain sensor and certain time gets old enough and we want to get rid of it, we can just actually go ahead and delete that index on disk as opposed to having to go through this whole expensive process of creating a tombstone and then eventually compacting it.

So, that's why time series databases are really useful. But, you know, for more detail on that because I kind of brush through it, I recommend watching my time series database video.

Okay, so I mentioned that some of these logs and some of these metrics either aggregated or unaggregated can go into a time series database, and that's going to be really good for both reads and writes and things like data visualization if you want to use something like Tableau. However, a lot of this stuff is going to be less formatted or kind of in a jumbled mess. And if we do have things like that we want to be able to process it later, perhaps the better place to put it would be something like S3. And we would use S3 as something known as a data lake where we're basically just throwing in a bunch of jumbled unformatted data. And then we know that we're going to be doing some processing on it later.

By putting it in S3, there are no requirements on, you know, the actual formatting of the logs, and that can be really useful sometimes because logs are coming from a ton of different sources and as are metrics, and it can be a pain in the ass to have to format them, and it might just be better to get them in the database and deal with that later.

So what might we do once that information is all in S3? Well, typically what we would do is then basically export our data from S3 to something like a Hadoop cluster because you can't do data processing directly from S3. It doesn't have good data locality, unfortunately, but it's just cheap, so that's why companies use it. You would use something like a Hadoop cluster and run Spark on that data once it's loaded in the cluster.

Spark is basically just a batch processing service that, you know, is super widely used. It's better than MapReduce for many reasons, which I've spoken about in the past. Basically, you can use Spark to do any complicated processing that you may have to do and potentially even format your data in a way so that you can throw it in a data warehouse.

So, what's really useful about having data in both a time series database and a data warehouse? Well, oftentimes, it's actually stored in a column-oriented format. And what that means is if you're trying to basically rip all the results from one single column, say there's one metric that you're really interested in, then being able to store all of your data in a column-oriented format means that all of the basically entries in a single column are going to

be in the same file, and you're going to get much better data locality when trying to make those reads, and it's going to be a much faster read.

Additionally, a nice thing about column-oriented storage is that you could use something like Parquet, which is just an open-source framework, to further basically compress your column-oriented storage because a lot of these metrics are going to be very similar to one another. And as a result, you can use strategies like dictionary encoding or bitmap encoding to really reduce the size of your data, and that's really important, obviously.

Another further thing to actually be able to do is if you can, if you know the format of all these logs as they're coming in, or you can kind of specify them beforehand, you can encode a lot of these messages using data encoding frameworks like Avro, Protocol Buffers, or Thrift. And that's going to be super useful because not only does it reduce the amount of data that has to be sent over the network, but more importantly, it is going to reduce the amount of data that's actually being held on disk. So, it's going to take the fact that you're storing a ton of data and ideally greatly reduce the size that you're actually having to store and hopefully save a bunch of costs for the company. At least I can say with certainty at Google, we pretty much use Protocol Buffers for everything. And the reasoning for that is, you know, you get huge network gains, and you also get huge disk gains. It's going to save a lot of time and money.

And then finally, eventually, you know, we have a ton of data. What can we do to basically, I don't know, say not necessarily get rid of some, but how can we cut costs because a lot of this data is going to become more and more irrelevant over time? Well, there are a few ways. For starters, you could do things like sub-sampling, where you only take a certain portion of the data and keep that in your databases. Another thing you can do is, as data gets older and older, you can move it to kind of an archiving solution. So, for Amazon specifically, you have S3 for the data that you really care about. And then as it gets old and it's less important to you, you can put it in a slower storage solution, something like AWS Glacier.

So, I hope this makes general sense for the most part. Logging metrics are obviously super important for any company, especially once they want to be running things like machine learning models on that because it's the way that they basically get better as a product. And, you know, it's kind of just the best way to improve your service. So, this is obviously something that every company cares about. With that being said, I've kind of described everything that you might want in a logging service. Let's go ahead and take a look at a diagram.

Alrighty, let's do this diagram. Sorry about my pen. It's flaking on me harder than a girl who stands you up on a date. But anyways, let's get into it - metrics and logging. So first, we have our client, or you know, this might even be just another internal server that's sending metrics. It might be an IoT device, basically any source of a metric or a log that, you know, we want to be keeping track of.

The metric that it's ultimately going to be producing is getting sent to a load balancer because everything is going to be horizontally scaled here because we want to be doing it on consumer hardware. And the load balancer can be in either an active-passive or

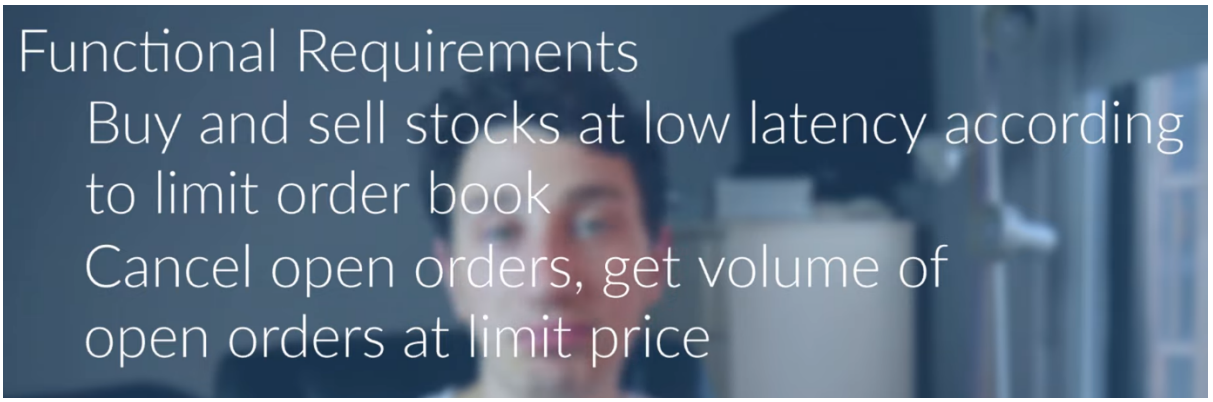
active-active type of configuration. So anyways, once we hit our logging service, we can go ahead and start throwing those messages into our Kafka queue where they're going to reach our Flink consumers.

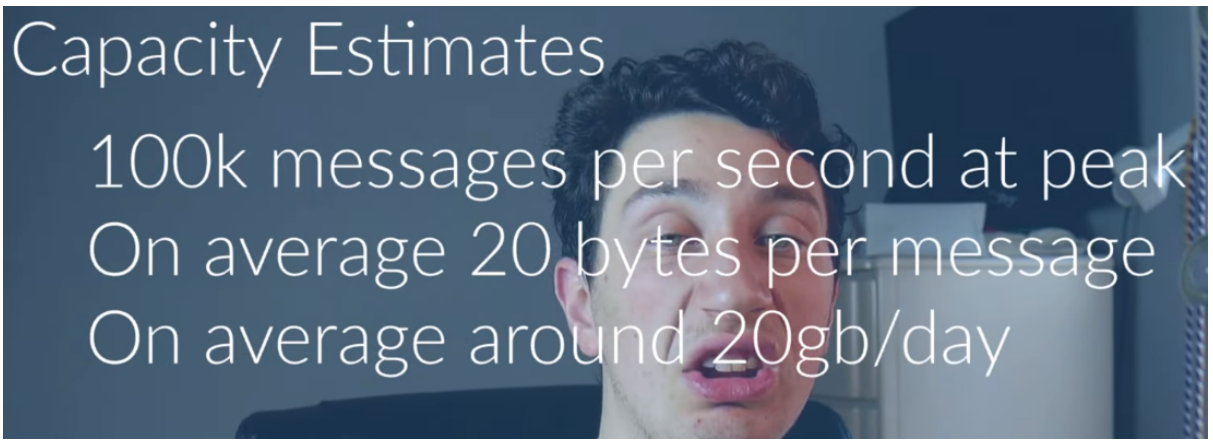
Ultimately, I chose Flink just because I wanted some real-time processing that is going to get things into our time series DB quickly because sometimes you want to be able to, you know, get at least close to real-time updates on the time series data. And you know, if, um, you're looking at server logs, you don't want like a couple of seconds lag before you see those messages. But obviously, you're going to get some lag, so it is what it is. The time series database can just use something like single-leader replication in order to, you know, ensure fault tolerance.

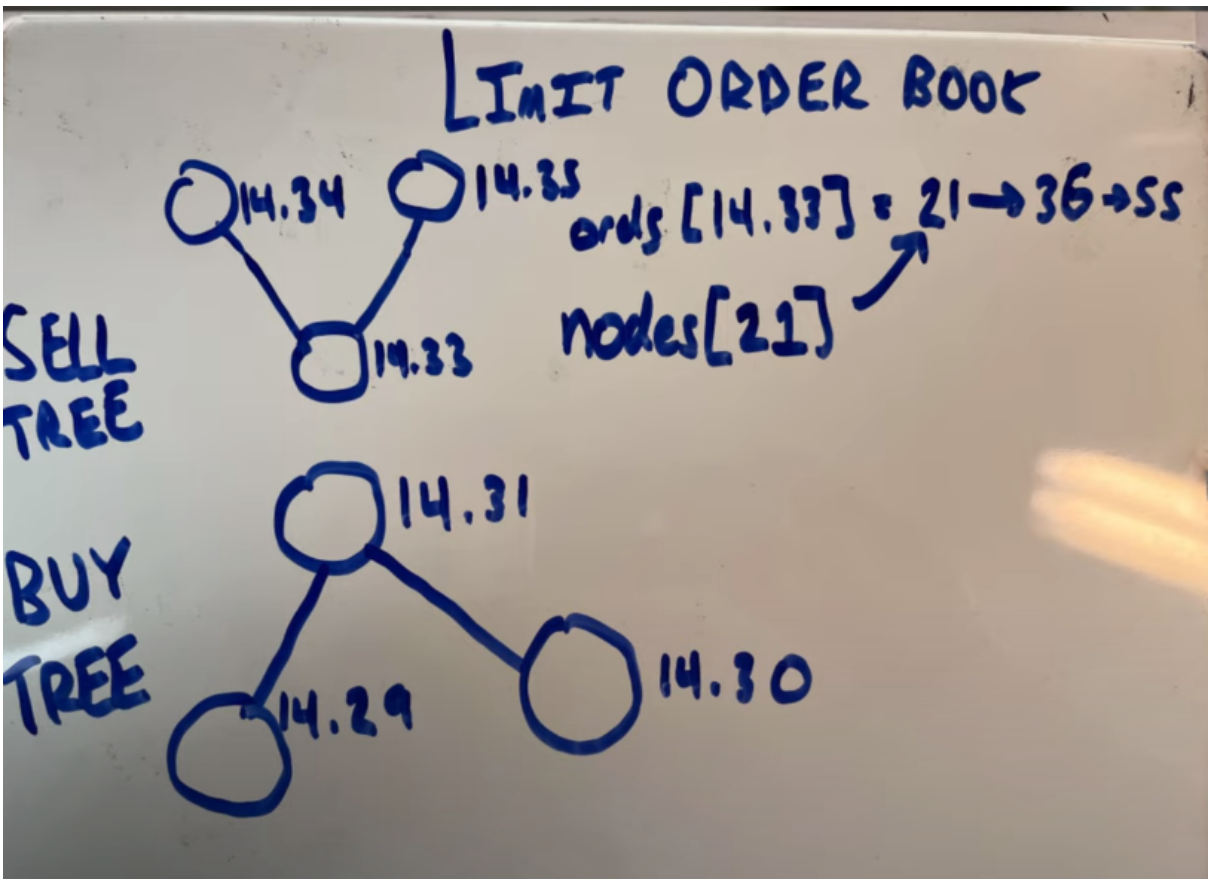
Another possible sink for data to go to from our Flink consumers is actually going to be S3. This is where it can be in something like, you know, an unformatted setting. And as a result of being an S3, we can either put it in Glacier if it's really old or better yet, we can go ahead and transport that data into a Hadoop cluster where we can start running all sorts of Spark jobs to put it into a more formatted setting such as a data warehouse where we're going to be storing all of our data in a column-oriented fashion, running compression on that. And that way, we can get something like SQL insights for our business analysts.

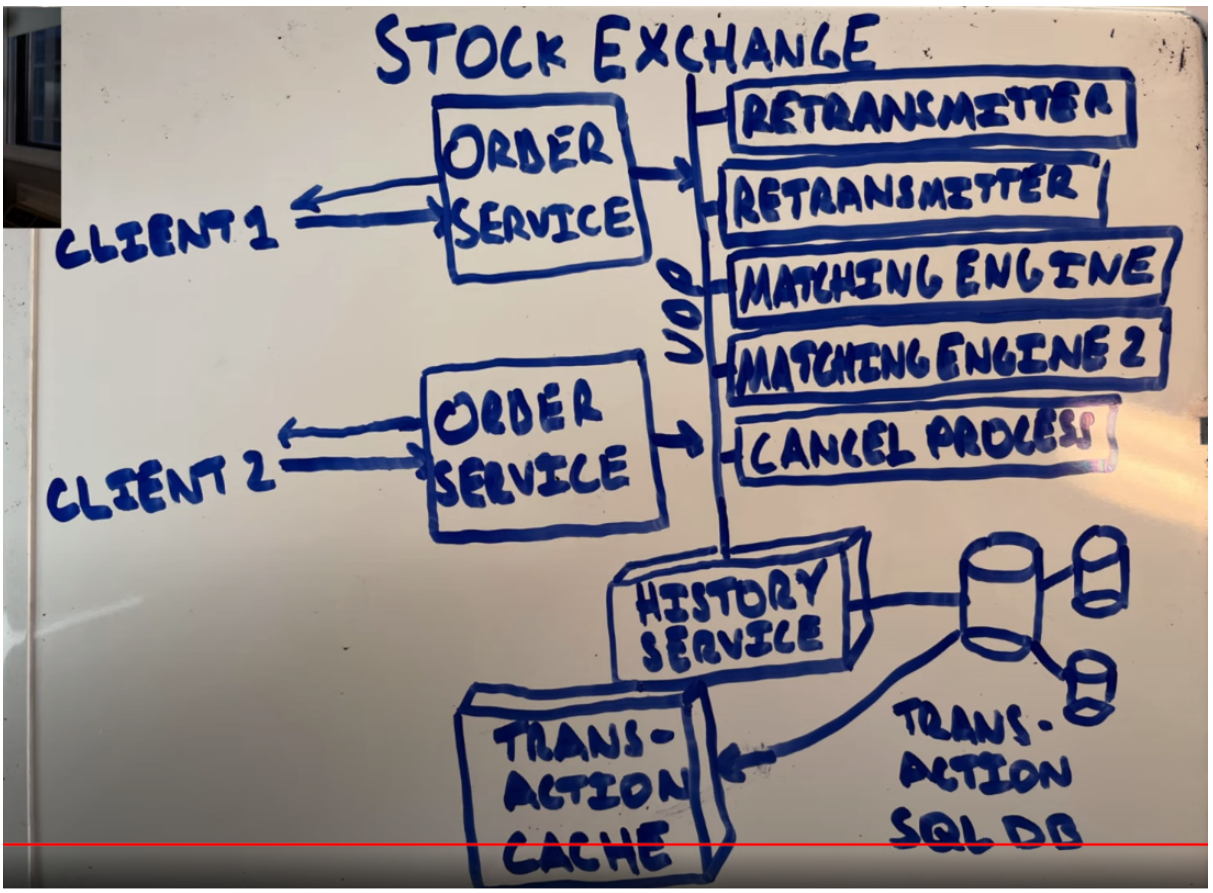
So yeah, I mean, a pretty quick overview of things. Obviously, in practice, there are going to be a lot more microservices that are going to be reading from the data warehouse. They're going to be reading from the time series database. They're going to be doing visualizations. There's going to be a ton of caches on top of all of these because, you know, when there's a ton of data, obviously we want to be caching a bunch of that, especially if it's going to be recent and a lot of people are going to be looking at it. This is kind of the initial building block for what it is that we're actually working with. However, at the end of the day, this is very much just a starting point. Metrics and logs are only as useful as what you're actually going to do with them. So this is how we might populate them, but at the end of the day, you know, you can make a whole other video devoted to just the types of insights that you might make once all of those logs are in a warehouse or in a time series database.

Low Latency Stock Exchange




Capacity Estimates
100k messages per second at peak
On average 20 bytes per message
On average around 20gb/day


LIMIT ORDER BOOK
14.34
14.35
14.33
ords[14.33] = 21 → 36 → 55
nodes[21]
SELL TREE
14.31
BUY TREE
14.29
14.30



Functional Requirements
Okay, so as per usual, let's quickly do some functional requirements and all the other steps that we're going to have to implement in order to start really talking about some design here. So as far as the things that we're going to have to implement in order to build our stock exchange, we want to be able to buy and sell stocks according to the spread, which means basically that stock should be sold when the highest potential buying price crosses the lowest potential selling price. We'll discuss more about this algorithm in a little bit. It's called a limit book, but like I said, we'll discuss it in a bit. Don't worry about that. Additionally, we want to be able to cancel orders if a client has already placed one. Furthermore, for those who are just generally inspecting market data, we want to be able to quickly see the amount of volume at a given limit price that people either want to sell at or buy at. We'll discuss some of the best algorithms to do that.

Capacity Estimates
Okay, as far as some capacity estimates go, empirically the stock market has proven to be very popular, and there are often 100,000 messages per second. If they're about 20 bytes per message, that means we're dealing with, say, 20 gigabytes per day or something on that scale.

API Design
As far as a pretty rudimentary API goes for our stock exchange, here are some of the things that you probably want to be able to do:
- Buy a security at a given price

- Sell a security at a given price
- Cancel an existing order with an order ID that's been provided to you
- Get the prices or the volume at a given price

In addition to that, if we're not only just building a stock exchange but also acting as if we're kind of like the broker as well, like a Robinhood or something, then you want to be able to see your past orders and things like that. That's where we would be adding a database to this whole mix. But for now, I'm just abstracting everything away and really just thinking about building a very low latency stock exchange. We'll talk about how we're going to do that.

Architecture overview

okay so let's start talking about an algorithm the reason again why i'm skipping our database schema here is because the database schema itself is not going to make sense when we have no database in actual stock exchange we need to be executing orders on the order of magnitude of nanoseconds basically and that's obviously not going to be suitable if we're reading from disk so everything is going to be done in memory here and ideally a lot of it in cache so let's start by actually talking about something called our matching engine so the matching engine is basically going to encapsulate all of the functionality of the limit order book so what is a limit order book because this is basically the algorithm of how we're going to determine when an order should actually be placed so you know as a client i can place buy orders i can place sell orders and what i'm going to do is i'm going to place them at a certain price now let's say i have a stock and i want to sell it at 20 bucks and my friend wants to buy it at 15. obviously things aren't going to work out between us and no order is actually going to be executed because we can't agree on a price on the other hand if i'm willing to sell my stock for twenty dollars and someone else comes along and you know names a price that they're willing to buy it at that's greater than or equal to then an order is going to be executed so i'll paste an image of a limit book on the slide but generally the point is this the limit book keeps track of all open orders for buys and sells of a given stock and once that spread is crossed basically meaning that someone is willing to buy at a price greater than or equal to someone else willing to sell then you go ahead and fulfill that order and it may not be at all the volume that either the buy or seller wants it may just be a partial order but the point is that things are going to start to get fulfilled so because we need to actually be running this order book algorithm so quickly it's extremely extremely important that everything is happening in memory and not only that but even within memory you want to be avoiding locking basically as much as possible and doing this algorithm as quickly as possible such that every single time a new order comes in we can quickly determine if there should actually be some buying and selling transaction and if so send it out such that everyone knows it so i've kind of said that this entire algorithm is going to be running on something called our matching engine so before i actually go into kind of the whole distributed design of our system let's quickly talk about the data structures and algorithms that might be running in our matching engine so based off one paper that i read online a really interesting data structure and algorithm that could be used for our magic engine is a type of basically tree and this tree can be organized in a way such that you have two separate trees for all the open buying orders and all the open selling orders and that at each node of a tree we have a linked list containing all the potential orders at a given limit price

this way every single time that an order comes in you basically put it in the proper node of this probably binary search tree or potentially even a self-balanced tree at the proper limit price node and in addition to the trees we have two sets of hash maps one hash map that basically points to every single start of the linked list at a given limit and then a second hash map for within each limit pointing to each node so what's the point of these hash maps well let's say you want to get the volume of all the orders at a certain price you can use the first hash map to quickly fetch that linked list and then iterate through it finding out all of the open orders at that limit secondly and more importantly if you want to cancel an order because cancellations are actually a very significant amount of the messages that go into a stock exchange you can use the second type of hash map to basically go ahead and pull that element right out of the linked list so those are basically going to be more or less constant time operations however in addition to that we also have buying and selling if we chose to use something like a heap here instead of just binary search trees then we could quickly um in you know peak from an element of the the by tree and the cell tree and that way we could see if any new order actually went ahead and crossed our spread and then in addition to that on inserting a new order all you have to do is use the hash map to quickly find the correct limit linked list that it goes into and then basically put the order at the end of that linked list making it a constant time operation so as you can imagine i'm sure because this is a very important system here this algorithm for kind of implementing an order book you know in memory has been very well thought out but chances are in an interview especially if it's not for something like a high frequency trading company and it's more so from the perspective of distributed systems something like that is going to be a little bit less important but now that we understand what our matching engine is what is it that we're actually doing with this matching engine or what functionality do we need to have well every single time a matching engine you know executes an order or more or less even just confirms that a client put in an order it has to go ahead and broadcast that information out to all of the clients to any people in the public who care about that information to any third parties that have to clear the trade which basically means they're responsible for the actual financial transaction going down the transfer of the money and so this matching engine is not only responsible for doing the logic but also for actually broadcasting everything out that it does to all interested parties and not only do we have to broadcast this but we have to broadcast it in a fair way which means that everyone who's interested should generally be getting all this information at the same time so how can we go ahead and do this well the technology that we would actually use is something called udp multicast now you may be asking well why use udp when something like tcp exists well there are a couple of reasons one because multicast makes sure that all that information that's being sent over the network is being sent out at the same time so you're achieving maximum fairness whereas tcp you can only get basically two nodes to connect at the same time so you have to do all these handshakes in a sequential order and then it's completely unfair because one client is going to have its handshake done first in addition to that udp multicast is generally faster because like i said it doesn't have to establish a handshake but it comes at the cost of not having sequence numbers and it also comes at the cost of not doing any flow control so if you're not really sure what these things are i recommend you watching my video about tcp versus udp but basically what this means is that when i'm the matching engine and i say oh you know what this order has just been executed um here are the two parties involved here's what happened i can't be sure that basically all of my client nodes are going to be receiving that message and i'm you know there's no telling that they didn't just drop a message and now they have no idea that their order was placed so how can we actually ensure that everyone is keeping track of all the

messages being sent out from the matching engine well we can actually create a couple of other nodes in our system known as retransmitters and i'll show these in a diagram in a little bit but the point of these retransmitters is to basically take in all of the outgoing messages from the matching node that say you know i received this order i received this order this transaction has just been executed and the entire job of the retransmitter node is to basically keep a log of all those outgoing messages so that if any of the clients or interesting parties or anything like that realize that they missed a message they can just ask the retransmitter node for it and then the retransmitter node can send that message this is also really useful if you know say you're a new client or a new server in our system that needs to kind of be up to speed the retransmitter node can basically just send you all of these messages so now we've kind of dealt with how one would deal with dropping udp messages but at the same time we don't have any fault tolerance in our system why because all these messages are going through our matching engine and the big issue with that is what if the matching engine crashes well in order to kind of deal with this what we need is obviously going to be some sort of secondary matching engine now the secondary matching engine is basically also just going to sit right alongside the matching engine probably going to be on a different machine and it's also just going to be running in memory so now in the past when we've seen single points of failure what will typically happen is you know we might use something like zookeeper or coordination service running some sort of consensus algorithm in order to keep a distributed log between them so that they can have the same state but that's completely infeasible for this type of solution because as we know consensus takes a long time it takes multiple network round trips between multiple nodes and it's probably all going to be over disk and so as a result we don't have the luxury here of using something like a coordination service so that the primary matching engine and the backup matching engine stay in sync rather instead we can really only do the following you might think at first oh hey actually what's going to happen is all the messages being sent to the primary matching engine should also just be sent over multicast to the backup matching engine and that's kind of the right intuition however in reality it doesn't work because all of these messages are going over the network and as a result there's no um guarantee over their ordering more or less so we know that there's this kind of arbitrary ordering of all the messages being received from the clients and especially due to all these high frequency traders they're all kind of trying to be the first one but only one of them really is going to be first and the kind of node determining the order of all the messages being received is the matching engine the secondary matching engine as long as it is the secondary should not be the one deciding the order it shouldn't have the ability to do so and in addition the only thing it can really do is default back to the primary matching engine for the ordering of all the messages in the stock exchange so how can it actually go ahead and take account for all of this and then be up to date if the primary matching engine is to go down well the only real option is when the primary matching engine is broadcasting all of its messages back to all the clients or to any of the interested parties the secondary matching engine is also listening to these and not only is it listening to these but it's updating its local state where its local state is going to be that limit order book so basically the secondary matching engine in tandem is updating its limit order book as the map as the primary matching engine does and sends out messages and that way if the primary matching engine were to fail uh you could use some sort of consensus system between them like zookeeper in order to basically determine if the primary went down and if so the secondary matching engine is going to take over and it is going to have an up-to-date state which is really good now the final kind of piece of this puzzle is can we make this thing any faster because obviously that is the main concern of

any stock exchange how fast can we make it because the faster stock exchanges get the more accurately the prices are reflected the less opportunities there are for people to kind of arbitrage individuals and take advantage of them in the stock market so speed is kind of everything here but the only other thing we can really do is kind of shard it and so how can we shard well the obvious way to do it would be on tickers right there are a bunch of different stocks out there you know there's apple there's google there's facebook there's microsoft all these things and we can make it such that you know we take our diagram our entire system of you know clients connecting with our couple of matching engines and a couple of pre-transmitters and all that and we just duplicate it you know 100 times over for all the tickers or something like that but the truth of the matter is even though this will make things more efficient we would also lose some abilities a lot of places um you know want to kind of make conditional trades where they're going to say well i'll do this order if you know both of these ticker uh transactions go through but otherwise cancel or something like that and if we were to shard our system this way then people kind of lose the ability to do multi-ticker transactions so that's something to keep in mind but um generally it has been shown that this like relatively simple design of keeping everything in memory using state machine replication because you know the secondary matching engine is basically just replicating all of the operations that are causing state in the primary matching engine and by doing all of these things and kind of handling the pitfalls of udp we can make sure that we actually have a very functional and quick stock exchange and in terms of achieving high throughput really the only thing that we're limited by is the latency of our operations the reasoning for this is that you know because udp has no congestion control if there are too many operations and we can't process them all in time a single one of these nodes that are our servers are going to get overloaded so the faster that we can process these messages the more actual network messages we can take in and then we won't have congestion issues over the network so in you know the stock exchange latency is throughput anyways let's get to a diagram so i can make this more concrete for everyone but i hope at least what i was saying is starting to make sense and then looking at the diagram will kind of hammer it down for all of you okay so as far as the diagram basically at the core of this we'll show off more or less a couple of clients and the clients are connected to our order services which are just you know typical web servers and all they're going to do is place an order to our order service and then once that's done the order services are more or less just starting to you know hammer the matching engine proposing messages to it now the matching engine is more or less going to be choosing one of them putting it in its limit book and then performing the necessary operations in accordance with the limit book algorithm as the matching engine more or less sends messages back up kind of the chain of command towards all of those order service servers towards the re-transmitters towards the backup matching engine the backup matching engine can go ahead and update its state in order to replicate it and then furthermore the re-transmitters can keep track of that message in the event that any of the other nodes in the system drop it so that they can send that message right back to them you'll note that there's a cancel process here and the entire purpose of this cancel process is basically just to decouple all the cancel functionality from the actual matching engine itself i'm not sure if this is necessary but it is something i've noticed in prior diagrams i guess you could have a separate thread running in order to basically remove an element from those linked lists and then i guess you would probably need a lock for something like that and then yeah i mean basically the point is from here even though this is kind of the atomic building block of this whole stock exchange this is where other relevant kind of micro services are going to do stream processing on all the transactions that come out you can have a mysql

database to deal with the actual exchange of money like i said there's gonna be some stream processing uh you can use that to run through some sort of machine learning model to see if there's fraud or fraud detection put things in a time series database and all that good stuff but generally speaking the important part is you want the matching engine doing as little as possible other than dealing with the order book and sending out those messages over udp and that's kind of the important part of this and it is the main thing to note then obviously the clients can go ahead and receive new data through those order services and perhaps that could be through tcp but the point is you know within kind of our network is the exchange where our network is our servers and the matching engine and the re-transmitters everything is unordered with no flow control and so we have to manually keep track of the order of messages so yeah that's about all i got there it's kind of a shame how many times