Requirements of a Distributed Search System's Design

Let's identify the requirements of a distributed search system and outline the resources we need.

We'll cover the following

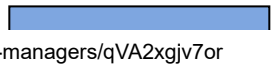
Requirements

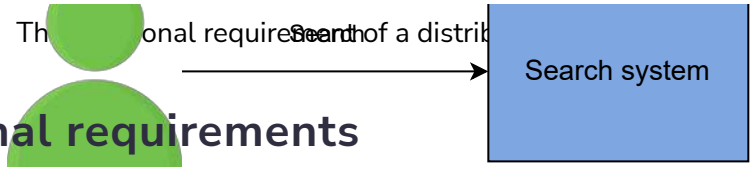
Let’s understand the functional and non-functional requirements of a distributed search system.

Functional requirements

The following is a functional requirement of a distributed search system:

- **Search**: Users should get relevant content based on their search queries.

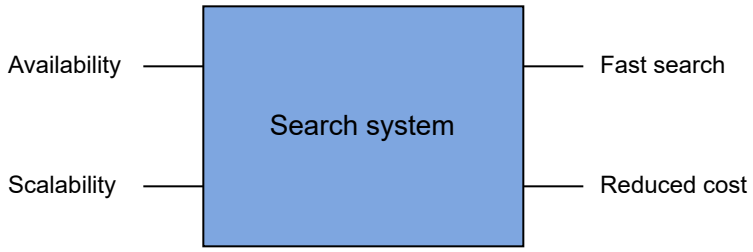




Non-functional requirements

Here are the non-functional requirements of a distributed search system:

- **Availability**: The system should be highly available to the users.
- **Scalability**: The system should have the ability to scale with the increasing amount of data. In other words, it should be able to index a large amount of data.
- **Fast search on big data**: The user should get the results quickly, no matter how much content they are searching.
- **Reduced cost**: The overall cost of building a search system should be less.



The non-functional requirement of a distributed search system

Resource estimation

Let’s estimate the total number of servers, storage, and bandwidth that is required by the distributed search system. We’ll calculate these numbers using an example of a YouTube search.

Number of servers estimation

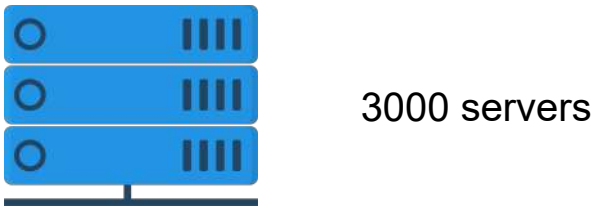
To estimate the number of servers, we need to know how many daily active users per day are using the search feature on YouTube and how many requests per second our single server can handle. We assume the following numbers:

- The number of daily active users who use the search feature is three million.
- The number of requests a single server can handle is 1,000.

The number of servers required is calculated using this formula:

$$\frac{Number\ of\ active\ users}{queries\ handled\ per\ server} = 3K\ servers$$

If three million users are searching concurrently, three million search requests are being generated at one time. A single server handles 1,000 requests at a time. Dividing three million by 1,000 gives us 3,000 servers.



The number of servers required for the YouTube search service

Storage estimation

document is uniquely identified by the video ID. This metadata contains the title of the video, its description, the channel name, and a transcript. We assume the following numbers for estimating the storage required to index one video:

- The size of a single JSON document is 200 KB.
- The number of unique terms or keys extracted from a single JSON document is 1,000.
- The amount of storage space required to add one term into the index table is 100 Bytes.

The following formula is used to compute the storage required to index on video:

$$Total_{storage/video} = Storage_{/doc} + (Terms_{/doc} \times Storage_{/term})$$

Total Storage Required to Index One Video on YouTube

Storage per JSON doc (KB)	No. of terms per doc	Storage per term (Bytes)	Total storage per video (KB)
200	1000	100	f 300

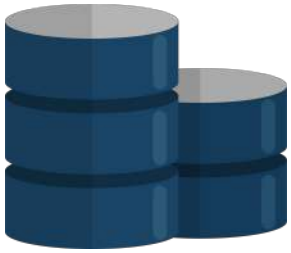
In the table above, we calculate the storage required to index one video. We have already seen that the total storage required per video is 300 KB. Assuming that, on average, the number of videos uploaded per day on YouTube is 6,000, let's calculate the total storage required to index the videos uploaded per day. The following formula is used to compute the storage required to index the videos uploaded to YouTube in one day:

$$Total_{storage/day} = No.\ of\ videos_{/day} \times Total_{storage/video}$$

Total Storage Required to Index Videos per Day on YouTube

No. of videos per day	Total storage per video (KB)	Total storage per day (GB)
6000	300	f 1.8

The total storage required to index 6,000 videos uploaded per day on YouTube is 1.8 GB. This storage requirement is just an estimation for YouTube. The storage need will increase if we provide a distributed search system as a service to multiple tenants.



Summarizing the storage requirement of a distributed search system for videos uploaded to YouTube per day

Bandwidth estimation

The data is transferred between the user and the server on each search request. We estimate the bandwidth required for the incoming traffic on the server and the outgoing traffic from the server. Here is the formula to calculate the required bandwidth:

$$Total_{bandwidth} = Total_{requests\_second} \times Total_{query\_size}$$

Incoming traffic

To estimate the incoming traffic bandwidth, we assume the following numbers:

- The number of search requests per day is 150 million.
- The search query size is 100 Bytes.

We can use the formula given above to calculate the bandwidth required for the incoming traffic.

Bandwidth Required for Incoming Search Queries per Second

No. of requests per second	Query size (Bytes)	Bandwidth (Mb/s)
1736.11	100	1.39

Outgoing traffic

Outgoing traffic is the response that the server returns to the user on the search request. We assume that the number of suggested videos against a search query is 80, and one suggestion is of the size 50 Bytes. Suggestions consist of an ordered list of the video IDs.

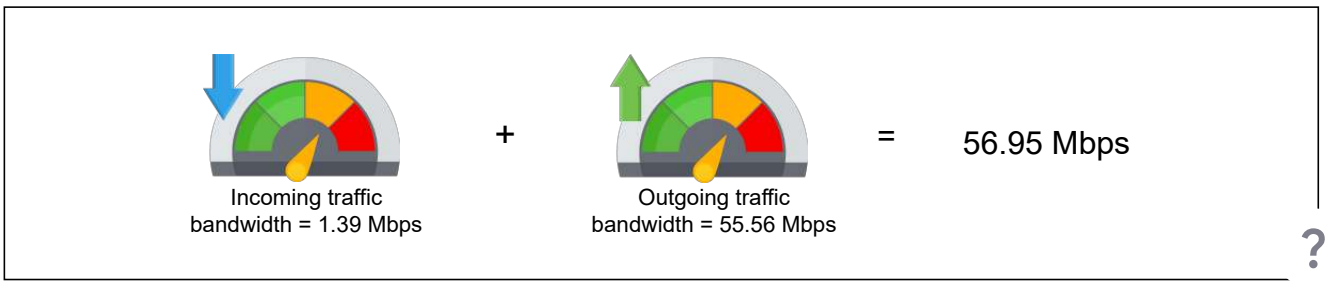
To estimate the outgoing traffic bandwidth, we assume the following numbers:

- The number of search requests per day is 150 million.
- The response size is 4,000 Bytes.

We can use the same formula to calculate the bandwidth required for the outgoing traffic.

Bandwidth Required for Outgoing Traffic per Second

No. of requests per second	Query size (Bytes)	Bandwidth (Mb/s)
1736.11	4000	f 55.56

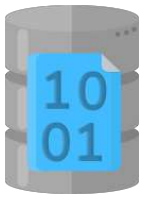


Summarizing the bandwidth requirements of a video search

> **Note:** The bandwidth requirements are relatively modest because we are assuming text results. Many search services can return small thumbnails and other media to enhance the search page. The bandwidth needs per page are intentionally low so that the service can provide near real-time results to the client.

Building blocks we will use

We need a distributed storage in our design. Therefore, we can use the blob store, a previously discussed building block, to store the data to be indexed and the index itself. We'll use a generic term, that is, "distributed storage" instead of the specific term "blob store."



Distributed storage: Blob store

To conclude, we explained what the search system's requirements are. We made resource estimations. And lastly, we mentioned the building block that we'll use in our design of a distributed search system.

Back

System Design: The Distributed Search

Next

Indexing in a Distributed Search

Mark as Completed

Indexing in a Distributed Search

Learn about indexing and its use in a distributed search.

We'll cover the following

We'll first describe what indexing is, and then we'll make our way toward distributing indexes over many nodes.

Indexing

Indexing is the organization and manipulation of data that's done to facilitate fast and accurate information retrieval.

Build a searchable index

The simplest way to build a searchable index is to assign a unique ID to each document and store it in a database table, as shown in the following table. The first column in the table is the ID of the text and the second column contains the text from each document.

Simple Document Index

ID	Document Content
1	Elasticsearch is the distributed and analytics engine that is based on REST APIs.
2	Elasticsearch is a Lucene library-based search engine.
3	Elasticsearch is a distributed search and analytics engine built on Apache Lucene.

The size of the table given above would vary, depending on the number of documents we have and the size of those documents. The table above is just an example, and the content from each document only consists of one or two sentences. With an actual, real-world example, the content of every

a fast process. On each search request, we have to traverse all the documents and count the occurrence of the search string in each document.

> **Note:** For a **fuzzy search**, we also have to perform different pattern-matching queries. Many strings in the documents would somehow match the searched string. First, we must find the unique candidate strings by traversing all of the documents. Then, we must single out the most approximate matched string out of these strings. We also have to find the occurrence of the most matched string in each document. This means that each search query takes a long time.

The response time to a search query depends on a few factors:

- The data organization strategy in the database.
- The size of the data.
- The processing speed and the RAM of the machine that's used to build the index and process the search query.

Running search queries on billions of documents that are document-level indexed will be a slowprocess, which may take many minutes, or even hours. Let's look at another data organization and processing technique that helps reduce the search time.

Inverted index

An **inverted index** is a HashMap-like data structure that employs a document-term matrix. Instead of storing the complete document as it is, it splits the documents into individual words. After this, the **document-term matrix** identifies unique words and discards frequently occurring words like "to," "they," "the," "is," and so on. Frequently occurring words like those are called **terms**. The document-term matrix maintains a **term-level index** through this identification of unique words and deletion of unnecessary terms.

For each term, the index computes the following information:

- The list of documents in which the term appeared.
- The frequency with which the term appears in each document.
- The position of the term in each document.

Inverted Index

Term	Mapping ([doc], [freq], [[loc])
elasticsearch	([1, 2, 3], [1, 1, 1], [[1], [1], [1]])

distributed	([1, 3], [1, 1], [[4], [4]])
restful	([1], [1], [[5]])
search	([1, 2, 3], [1, 1, 1], [[6], [4], [5]])
analytics	([1, 3], [1, 1], [[8], [7]])
engine	([1, 2, 3], [1, 1, 1], [[9], [5], [8]])
heart	([1], [1], [[12]])
elastic	([1], [1], [[15]])
stack	([1], [1], [[16]])
lucene	([2, 3], [1, 1], [[9], [12]])
library	([2], [1], [[10]])
Apache	([3], [1], [[11]])

In the table above, the “Term” column contains all the unique terms that are extracted from all of the documents. Each entry in the “Mapping” column consists of three lists:

- A list of documents in which the term appeared.
- A list that counts the frequency with which the term appears in each document.
- A two-dimensional list that pinpoints the position of the term in each document. A term can appear multiple times in a single document, which is why a two-dimensional list is used.

> **Note:** Instead of lists, the mappings could also be in the form of tuples— such as doc, freq, and loc.

Inverted index is one of the most popular index mechanisms used in document retrieval. It enables efficient implementation of boolean, extended boolean, proximity, relevance, and many other types of search algorithms.

Advantages of using an inverted index

- An inverted index facilitates full-text searches.
- An inverted index reduces the time of counting the occurrence of a word in each document at the run time because we have mappings against each term.

Disadvantages of using an inverted index

- There is storage overhead for maintaining the inverted index along with the actual documents. However, we reduce the search time.
- Maintenance costs (processing) on adding, updating, or deleting a document. To add a document, we extract terms from the document. Then, for each extracted term, we either add a new row in the inverted index or update an existing one if that term already has an entry in the inverted index. Similarly, for deleting a document, we conduct processing to find the entries in the inverted index for the deleted document's terms and update the inverted index accordingly.

Searching from an inverted index

Consider a system that has the following mappings when we search for the word "search engine:"

Term	Mapping
search	([1, 2, 3], [1, 1, 1], [[6], [4], [5]])
engine	([1, 2, 3], [1, 1, 1], [[9], [5], [8]])

Both of these words are found in documents 1, 2, and 3. Both words appear once in each document.

The word “search” is located at position 6 in document 1, at position 4 in document 2, and position 5 in document 3.

The word “engine” is located at position 9 in document 1, position 5 in document 2, and position 8 in document 3.

A single term can appear in millions of documents. Thus, the list of documents returned against a search query could be very long.

Question

Would this technique work when too many documents are found against a single term?

Show Answer

Factors of index design

Here are some of the factors that we should keep in mind while designing an index:

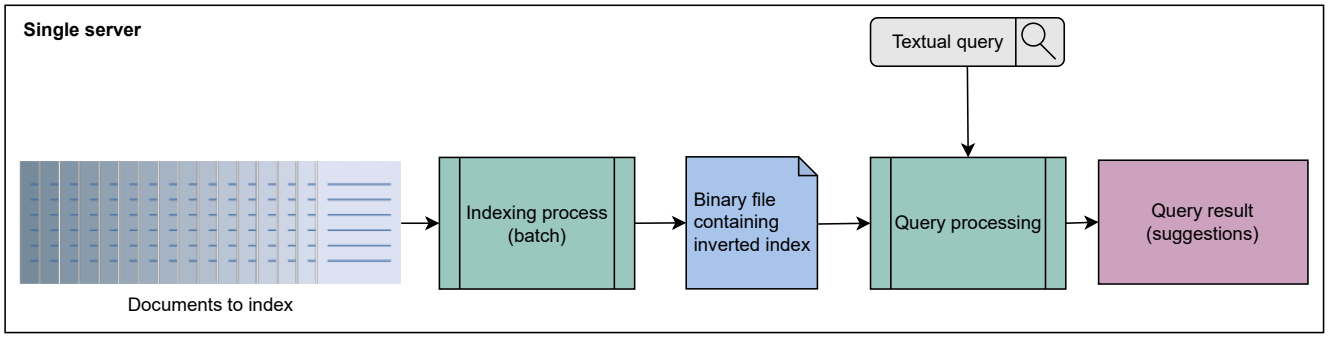
- **Size of the index**: How much computer memory, and RAM, is required to keep the index. We keep the index in the RAM to support the low latency of the search.
- **Search speed**: How quickly we can find a word from an inverted inde

- **Maintenance of the index**: How efficiently the index can be updated if we add or remove a document.
- **Fault tolerance**: How critical it is for the service to remain reliable. Coping with index corruption, supporting whether invalid data can be treated in isolation, dealing with defective hardware, partitioning, and replication are all issues to consider here.
- **Resilience**: How resilient the system is against someone trying to game the system and guard against search engine optimization (SEO) schemes, since we return only a handful of relevant results against a search.

In light of the design factors listed above, let's look at some problems with building an index on a centralized system.

Indexing on a centralized system

In a **centralized search system**, all the search system components run on a single node, which is computationally quite capable. The architecture of a centralized search system is shown in the following illustration:



The architecture of a centralized search system

- The **indexing process** takes the documents as input and converts them into an inverted index, which is stored in the form of a binary file.
- The **query processing** or **search process** interprets the binary file that contains the inverted index. It also computes the intersection of the

inverted lists for a given query to return the search results against the query.

These are the problems that come with the architecture of a centralized search system:

- SPOF (single point of failure)
- Server overload
- Large size of the index

SPOF: A centralized system is a single point of failure. If it's dead, no search operation can be performed.

Server overload: If numerous users perform queries and the queries are complicated, it stresses the server (node).

Large size of the index: The size of the inverted index increases with the number of documents, placing resource demands on a single server. The bigger the computer system, the higher the cost and complexity of managing it.

> **Note:** With a distributed system, low-cost computer systems are utilized, which is cost effective overall.

An inverted index needs to be loaded into the main memory when adding a document or running a search query. A large portion of the inverted index must fit into the RAM of the machine for efficiency.

According to Google analytics in 2022, there are hundreds of billions of web pages, the total size of which is around 100 petabytes. If we make a search system for the worldwide web, the inverted index size will also be in petabytes. This means we have to load petabytes of data into the RAM. It's impractical and inefficient to increase the resources of a single machine for

indexing a billion pages instead of shifting to a distributed system and utilizing the power of parallelization. Running a search query on a single, large inverted index results in a slow response time.

> **Note:** Searching a book from a shelf that holds a hundred books is easier than searching a book from a shelf holding a million books. The search time increases with the volume of data we search from.

Attacks on centralized indexing can have a higher impact than attacks on a distributed indexing system. Furthermore, the odds of bottlenecks (which can arise in server bandwidth or RAM) are also lower in a distributed index.

In this lesson, we learned about indexing, and we looked into the problems of indexing on a centralized system. The next lesson presents a distribution solution for indexing.

← Back

Requirements of a Distributed Search Syste...

Next →

Design of a Distributed Search

Mark as Completed

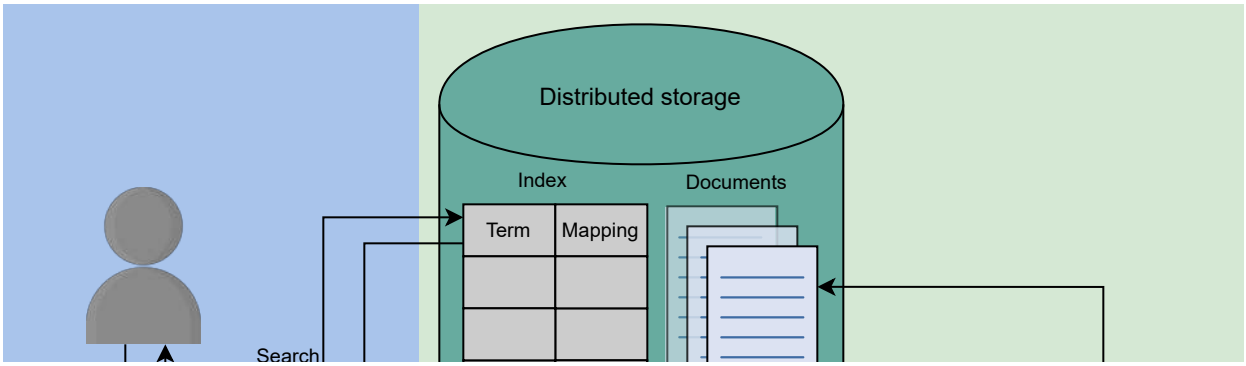
Design of a Distributed Search

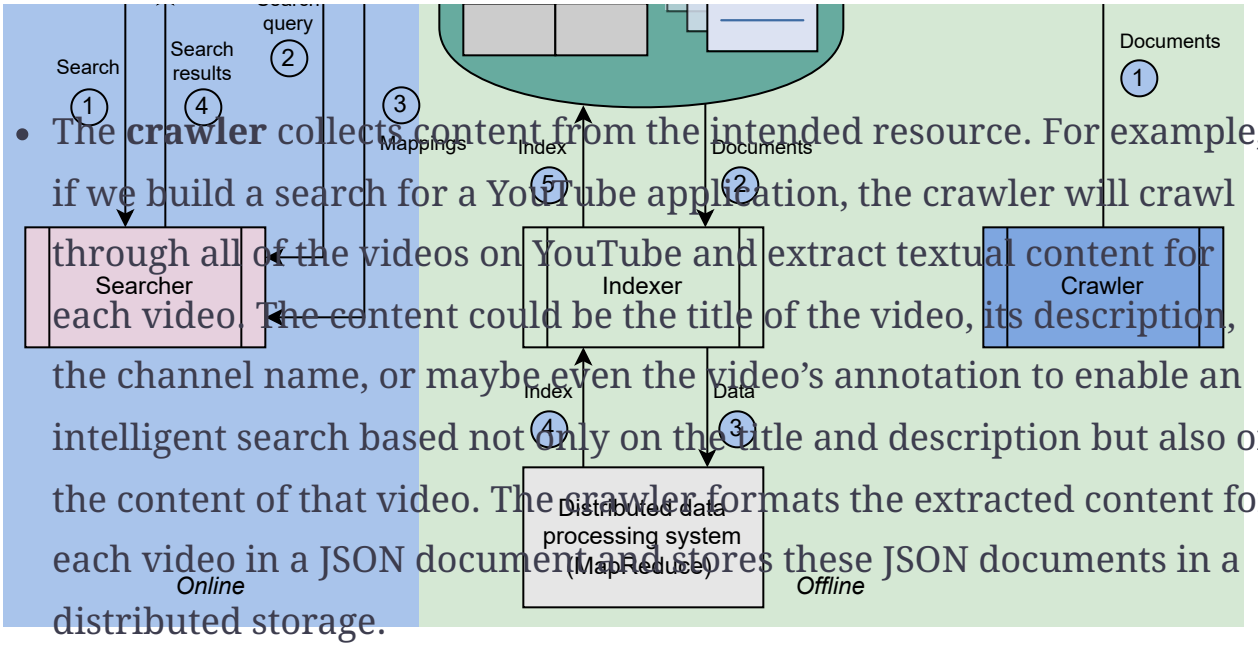
Get an overview of the design of a distributed search system that manages a large number of queries per second.

We'll cover the following

High-level design

Let's shape the overall design of a distributed search system before getting into a detailed discussion. There are two phases of such a system, as shown in the illustration below. The ***offline*** phase involves data crawling and searching for results against the search query by the user.





- The **crawler** collects content from the intended resource. For example, if we build a search for a YouTube application, the crawler will crawl through all of the videos on YouTube and extract textual content for each video. The content could be the title of the video, its description, the channel name, or maybe even the video's annotation to enable an intelligent search based not only on the title and description but also on the content of that video. The crawler formats the extracted content for each video in a JSON document and stores these JSON documents in a distributed storage.
- The **indexer** fetches the documents from a distributed storage and indexes these documents using **MapReduce**, which runs on a distributed cluster of commodity machines. The indexer uses a **distributed data processing system** like MapReduce for parallel and distributed index construction. The constructed index table is stored in the distributed storage.
- The **distributed storage** is used to store the documents and the index.
- The **user** enters the search string that contains multiple words in the search bar.
- The **searcher** parses the search string, searches for the mappings from the index that are stored in the distributed storage, and returns the most matched results to the user. The searcher intelligently maps the incorrectly spelled words in the search string to the closest vocabulary words. It also looks for the documents that include all the words and ranks them.

API design

Since the user only sends requests in the form of a string, the API design is quite simple.

Search: The `search` function runs when a user queries the system to find some content.

```
search(query)
```

Parameter	Description
`query`	This is the textual query entered by the user in the search bar, based o are found.

Detailed discussion

Since the indexer is the core component in a search system, we discussed an indexing technique and the problems associated with centralized indexing in the previous lesson. In this lesson, we consider a distributed solution for indexing and searching.

Distributed indexing and searching

Let's see how we can develop a distributed indexing and searching system. We understand that the input to an indexing system is the documents we created during crawling. To develop an index in a distributed fashion, we employ a large number of low-cost machines (nodes) and partition or divide the documents based on the resources they have. All the nodes are connected. A group of nodes is called a **cluster**.

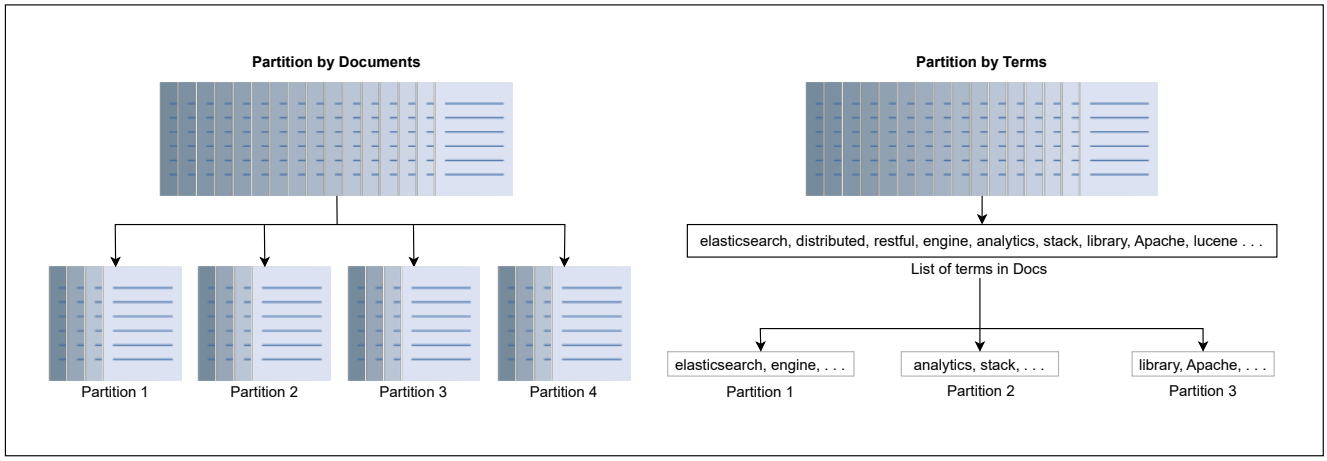
Tip

We use numerous small nodes for indexing to achieve cost efficiency. This process requires us to partition or split the input data (documents) among

these nodes. However, a key question needs to be addressed: How do we perform this partitioning?

The two most common techniques used for data partitioning in distributed indexing are these below:

- **Document partitioning**: In document partitioning, all the documents collected by the web crawler are partitioned into subsets of documents. Each node then performs indexing on a subset of documents that are assigned to it.
- **Term partitioning**: The dictionary of all terms is partitioned into subsets, with each subset residing at a single node. For example, a subset of documents is processed and indexed by a node containing the term “search.”

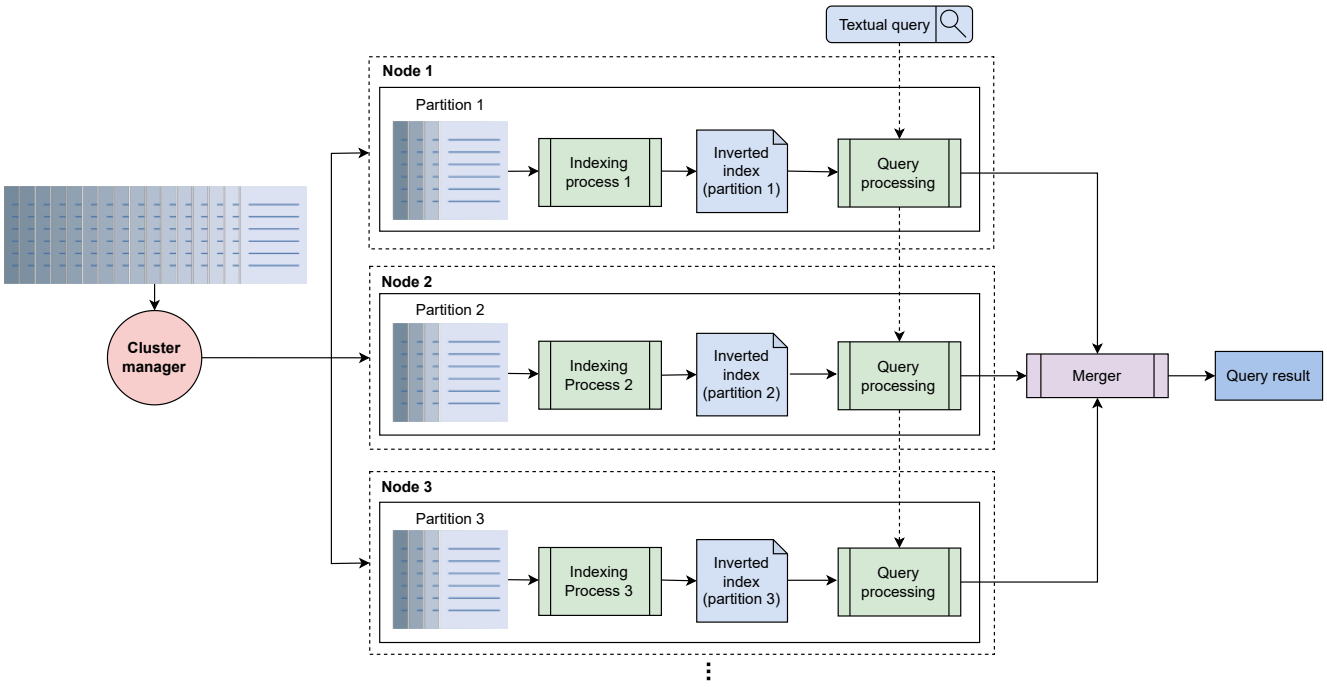


Types of data partitioning in a distributed search

In term partitioning, a search query is sent to the nodes that correspond to the query terms. This provides more concurrency because a stream of search queries with different query terms will be served by different nodes. However, term partitioning turns out to be a difficult task in practice. Multiword queries necessitate sending long mapping lists between groups of nodes for merging, which can be more expensive than the benefits from the increased concurrency.

In document partitioning, each query is distributed across all nodes, and the results from these nodes are merged before being shown to the user. This method of partitioning necessitates less inter-node communication. In our design, we use document partitioning.

Following document partitioning, let's look into a distributed design for index construction and querying, which is shown in the illustration below. We use a cluster that consists of a number of low-cost nodes and a cluster manager. The cluster manager uses a MapReduce programming model to parallelize the index's computation on each partition. MapReduce can work on significantly larger datasets that are difficult to be handled by a single large server.



Distributed indexing and searching in a parallel fashion on multiple nodes in a cluster of commodity machines

The system described above works as follows:

Indexing

- We have a document set already collected by the crawler.

- The **cluster manager** splits the input document set into N number of partitions, where N is equal to three in the illustration above. The size of each partition is decided by the cluster manager given the size of the data, the computation, memory limits, and the number of nodes in the cluster. All the nodes may not be available for various reasons. The cluster manager monitors the health of each node through **periodic heartbeats**. To assign a document to one of the N partitions, a **hashing function** can be utilized.
- After making partitions, the cluster manager runs indexing algorithms for all the N partitions simultaneously on the N number of nodes in a cluster. Each indexing process produces a tiny inverted index, which is stored on the node's local storage. In this way, we produce N tiny inverted indices rather than one large inverted index.

Searching

- In the search phase, when a user query comes in, we run parallel searches on each tiny inverted index stored on the nodes' local storage generating N queries.
- The search result from each inverted tiny index is a mapping list against the queried term (we assume a single word/term user query). The **merger** aggregates these mapping lists.
- After aggregating the mapping lists, the merger sorts the list of documents from the aggregated mapping list based on the frequency of the term in each document.
- The sorted list of documents is returned to the user as a search result. The documents are shown in sorted (ascending) order to the user.

> **Note:** We've designed a search system where we utilized a distributed system and parallelized the indexing and searching process. This helped us handle large datasets by working on the

> smaller partitions of documents. It should be noted that both searching and indexing are performed on the same node. We refer to this idea as **colocation**.

The proposed design works, and we can replicate it across the globe in various data centers to facilitate all users. Thus, we can achieve the following advantages:

- Our design will not be subject to a single point of failure (SPOF).
- Latency for all users will remain small.
- Maintenance and upgrades in individual data centers will be possible.
- Scalability (serving more users per second) of our system will be improved.

Replication

We make replicas of the indexing nodes that produce inverted indices for the assigned partitions. We can answer a query from several sets of nodes with replicas. The overall concept is simple. We continue to use the same architecture as before, but instead of having only one group of nodes, we have R groups of nodes to answer user queries. R is the number of replicas. The number of replicas can expand or shrink based on the number of requests, and each group of nodes has all the partitions required to answer each query.

Each group of nodes is hosted on different availability zones for better performance and availability of the system in case a data center fails.

> **Note:** A load balancer component is necessary to spread the queries across different groups of nodes and retry in case of any error.

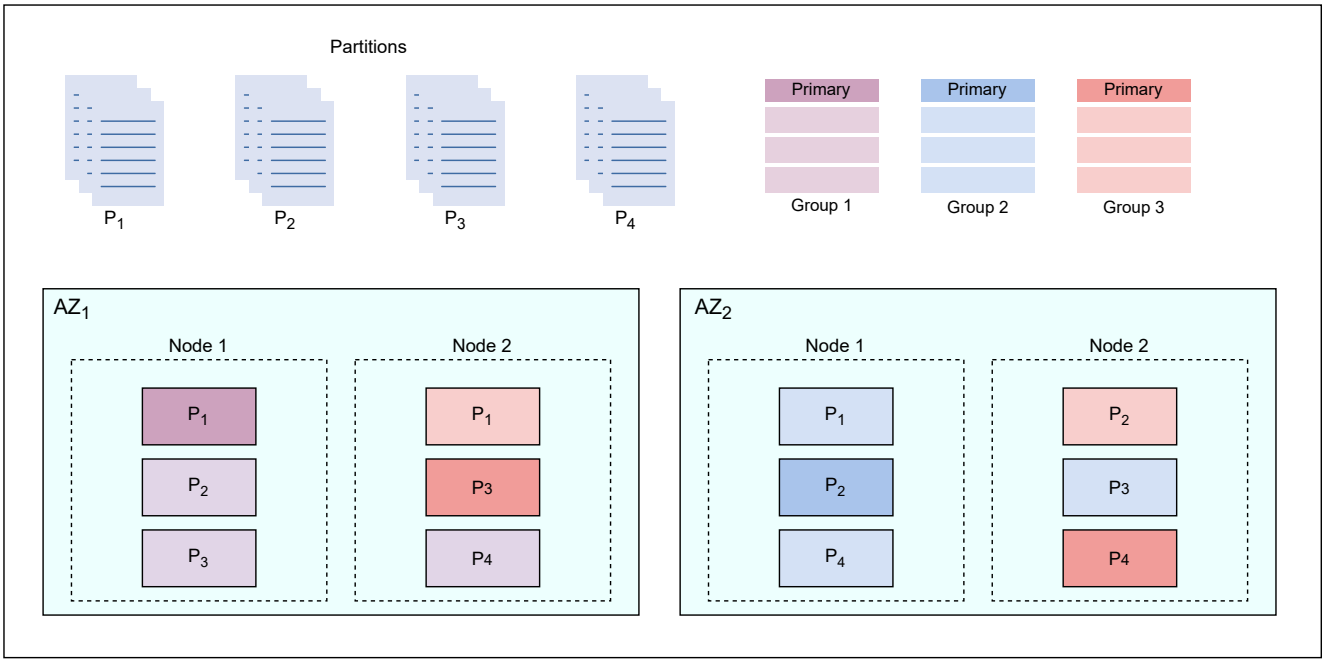
Replication factor and replica distribution

Generally, a replication factor of three is enough. A replication factor of three means three nodes host the same partition and produce the index. One of the three nodes becomes the primary node, while the other two are replicas. Each of these nodes produces indexes in the same order to converge on the same state.

To illustrate, let's divide the data, a document set, into four partitions. Since the replication factor is three, one partition will be hosted by three nodes. We'll assume that there are two availability zones (AZ_1 and AZ_2). And in each availability zone, we have two nodes. Each node acts as a primary for only one partition (For example, Node 1 in AZ_1 is the primary node for partition P_1). The three copies (pink, blue, and purple) for a partition are shared between the two AZ instances so that two copies are in one zone and the third copy is in another zone. Three colors represent three replicas of each partition. For example, the following is true for partition P_4:

- The first replica, represented by the color pink, is placed in Node 2 of AZ_2
- The second replica, represented by the color blue, is placed in Node 1 of AZ_2
- The third replica, represented by the color purple, is placed in Node 2 of AZ_1

Each group in the illustration below consists of one replica from all of the four partitions (P_1, P_2, P_3, P_4)



The replica distribution: Each node contains one primary partition and two replicas

In the above illustration, the primary replica for P_1 is indicated by the dark purple color, the primary replica for P_2 is represented by the dark blue color, and the primary replica for P_3 and P_4 is represented by the dark pink color.

Now that we have completed replication, let's see how indexing and searching are performed in these replicas.

Indexing with replicas

From the diagram above, we assume that each partition is forwarded to each replica for index computation. Let's look at the example where we want to index partition P_1. This means that the same partition will be forwarded to all three replicas in both availability zones. Therefore, each node will compute the index simultaneously and reach the same state.

The advantage of this strategy is that the indexing operation will not suffer the primary node fails.

Searching with replicas

We have three copies of each partition's index. The load balancer chooses one of the three copies of each partition to perform the query. An increased number of copies improves the scalability and availability of the system. Now, the system can handle three times more queries in the same amount of time.

Summary

In this lesson, we learned how to handle a large number of data, and a large number of queries with these strategies:

- Parallel indexing and searching, where both of these processes are colocated on the same nodes.
- Replicating each partition, which means that we replicate the indexing and searching process as well.

We successfully designed a system that scales with read (search) and write (indexing) operations colocated on the same node. But, this scaling method brings some drawbacks. We'll look into the drawbacks and their solutions in the next lesson.

← Back

Indexing in a Distributed Search

Next →

Scaling Search and Indexing

Mark as Completed

Design of a Distributed Logging Service

Learn how to design a distributed logging service.

We'll cover the following

We'll design the distributed logging system now. Our logging system should log all activities or messages (we'll not incorporate sampling ability into our design).

Requirements

Let's list the requirements for designing a distributed logging system:

Functional requirements

The functional requirements of our system are as follows:

- **Writing logs**: The services of the distributed system must be able to write into the logging system.
- **Searchable logs**: It should be effortless for a system to find logs. Similarly, the application's flow from end-to-end should also be effortless.
- **Storing logging**: The logs should reside in distributed storage for easy access.
- **Centralized logging visualizer**: The system should provide a unified view of globally separated services.

Non-functional requirements

The non-functional requirements of our system are as follows:

- **Low latency**: Logging is an I/O-intensive operation that is often much slower than CPU operations. We need to design the system so that logging is not on an application's critical path.
- **Scalability:** We want our logging system to be scalable. It should be able to handle the increasing amounts of logs over time and a growing number of concurrent users.
- **Availability:** The logging system should be highly available to log the data.

Building blocks we will use

The design of a distributed logging system will utilize the following building blocks:

- **Pub-sub system**: We'll use a pub-sub- system to handle the huge size of logs.
- **Distributed search**: We'll use distributed search to query the logs efficiently.

Pub-sub system

Distributed search

Building blocks we will use

API design

The API design for this problem is given below:

Write a message

The API call to perform writing should look like this:

```
write(unique_ID, message_to_be_logged)
```

Parameter	Description
`unique_ID`	It is a numeric ID containing `application-id`, `service-id`, and a time stamp.
`message_to_be_logged`	It is the log message stored against a unique key.

Search log

The API call to search data should look like this:

```
searching(keyword)
```

This call returns a list of logs that contain the keyword.

Parameter	Description
`keyword`	It is used for finding the logs containing the `keyword`.

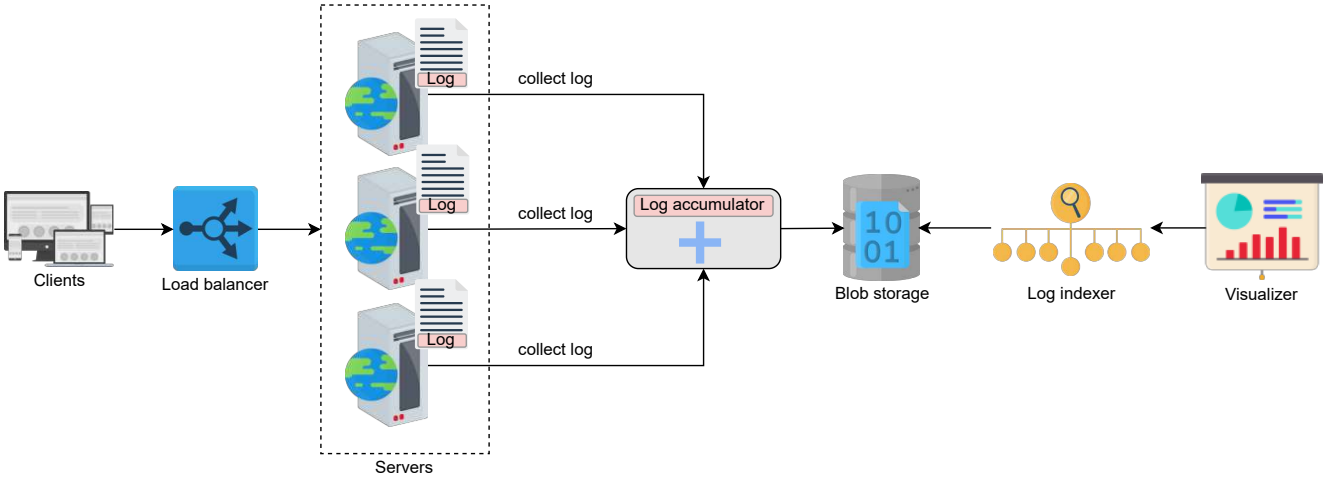
Initial design

In a distributed system, clients across the globe generate events by requesting services from different serving nodes. The nodes generate logs while handling each of the requests. These logs are accumulated on the respective nodes.

In addition to the building blocks, let's list the major components of our system:

- **Log accumulator**: An agent that collects logs from each node and dumps them into storage. So, if we want to know about a particular event, we don't need to visit each node, and we can fetch them from our storage.
- **Storage**: The logs need to be stored somewhere after accumulation. We'll choose blob storage to save our logs.
- **Log indexer**: The growing number of log files affects the searching ability. The log indexer will use the distributed search to search efficiently.
- **Visualizer**: The visualizer is used to provide a unified view of all the logs.

The design for this method looks like this:



Initial design

There are millions of servers in a distributed system, and using a single log accumulator severely affects scalability. Let's learn how we'll scale our system.

Logging at various levels

Let's explore how the logging system works at various levels.

In a server

In this section, we'll learn how various services belonging to different apps will log in to a server.

Let's consider a situation where we have multiple different applications on a

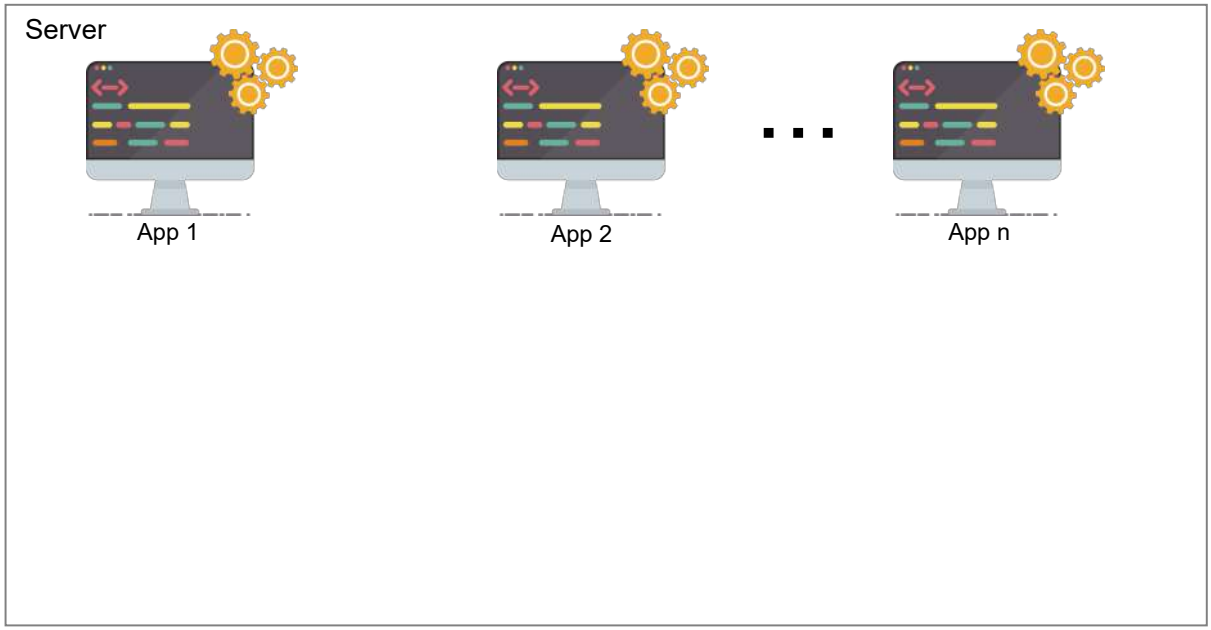
have services like authenticating users, fetching carts, and more running at the same time. Every service produces logs. We use an ID with `application-id`, `service-id`, and its time stamp to uniquely identify various services of multiple applications. Time stamps can help us to determine the causality of events.

Each service will push its data to the **log accumulator** service. It is responsible for these actions:

- Receiving the logs.
- Storing the logs locally.
- Pushing the logs to a pub-sub system.

We use the pub-sub system to cater to our scalability issue. Now, each server has its log accumulator (or multiple accumulators) push the data to pub-sub. The pub-sub system is capable of managing a huge amount of logs.

To fulfill another requirement of low latency, we don't want the logging to affect the performance of other processes, so we send the logs asynchronously via a low-priority thread. By doing this, our system does not interfere with the performance of others and ensures availability.



Multiple applications running on a server, and each application has various microservices

1 of 3

We should be mindful that data can be lost in the process of logging huge amounts of messages. There is a trade-off between user-perceived latency and the guarantee that log data persists. For lower latency, log services often keep data in RAM and persist them asynchronously. Additionally, we can minimize data loss by adding redundant log accumulators to handle growing concurrent users.

Point to Ponder

Question

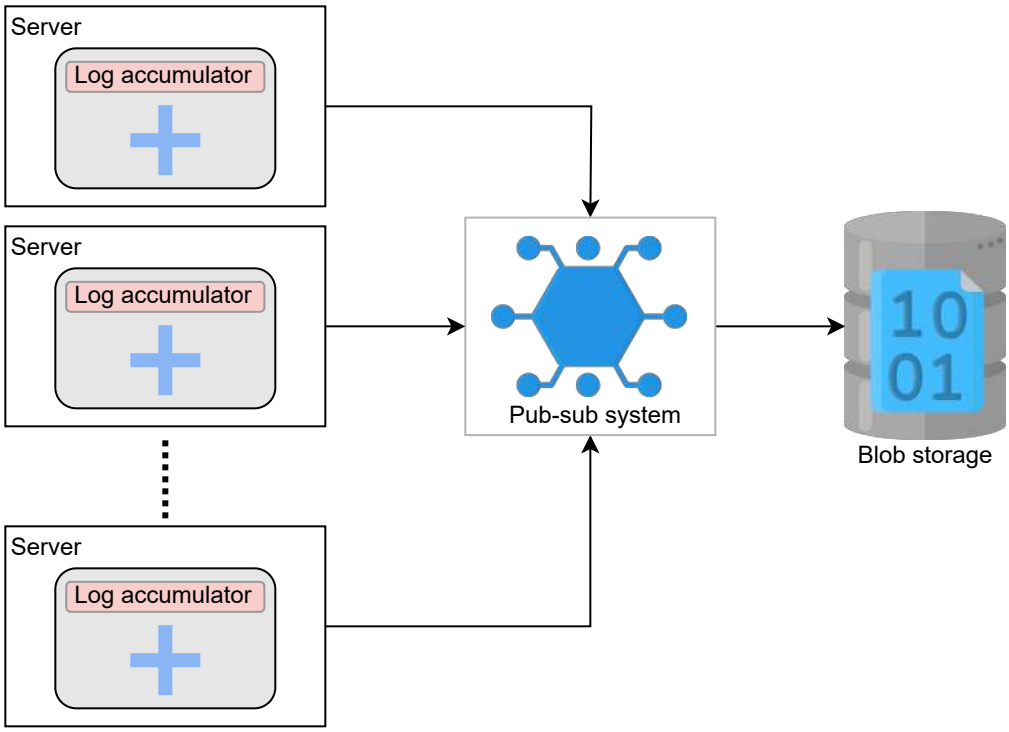
How does logging change when we host our service on a multi-tenant cloud (like AWS) versus when an organization has exclusive control of the infrastructure (like Facebook), specifically in terms of logs?

Show Answer

> **Note:** For applications like banking and financial apps, the logs must be very secure so hackers cannot steal the data. The common practice is to encrypt the data and log. In this way, no one can decrypt the encrypted information using the data from logs.

At datacenter level

All servers in a data center push the logs to a pub-sub system. Since we use a horizontally-scalable pub-sub system, it is possible to manage huge amounts of logs. We may use multiple instances of the pub-sub per data center. It makes our system scalable, and we can avoid bottlenecks. Then, the pub-sub system pushes the data to the blob storage.



Log accumulator sending data to the pub-sub system

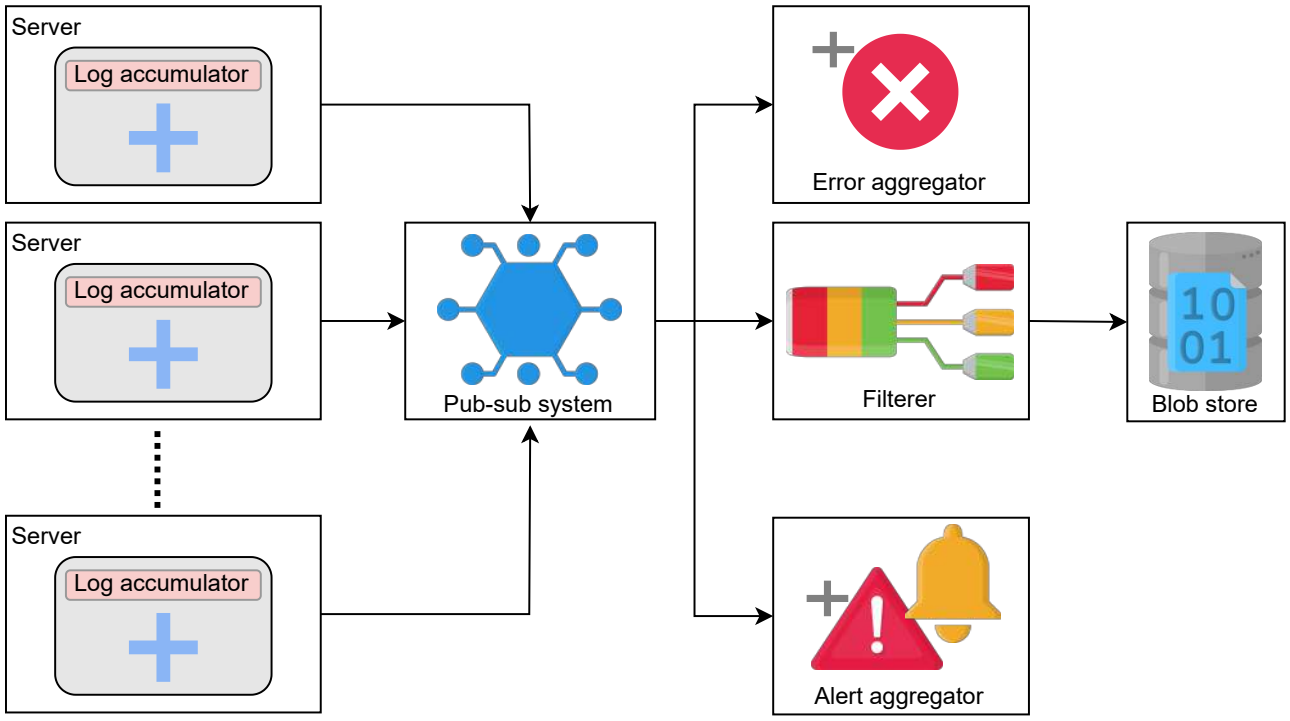
The data does not reside in pub-sub forever and gets deleted after a few days before being stored in archival storage. However, we can utilize the data while it is available in the pub-sub system. The following services will work on the pub-sub data:

- **Filterer**: It identifies the application and stores the logs in the blob storage reserved for that application since we do not want to mix logs two different applications.
- **Error aggregator**: It is critical to identify an error as quickly as possible. We use a service that picks up the error messages from the

pub-sub system and informs the respective client. It saves us the trouble of searching the logs.

- **Alert aggregator**: Alerts are also crucial. So, it is important to be aware of them early. This service identifies the alerts and notifies the appropriate stakeholders if a fatal error is encountered, or sends a message to a monitoring tool.

The updated design is given below:



Adding a filterer, error aggregator, and alert aggregator

Point to Ponder

Question

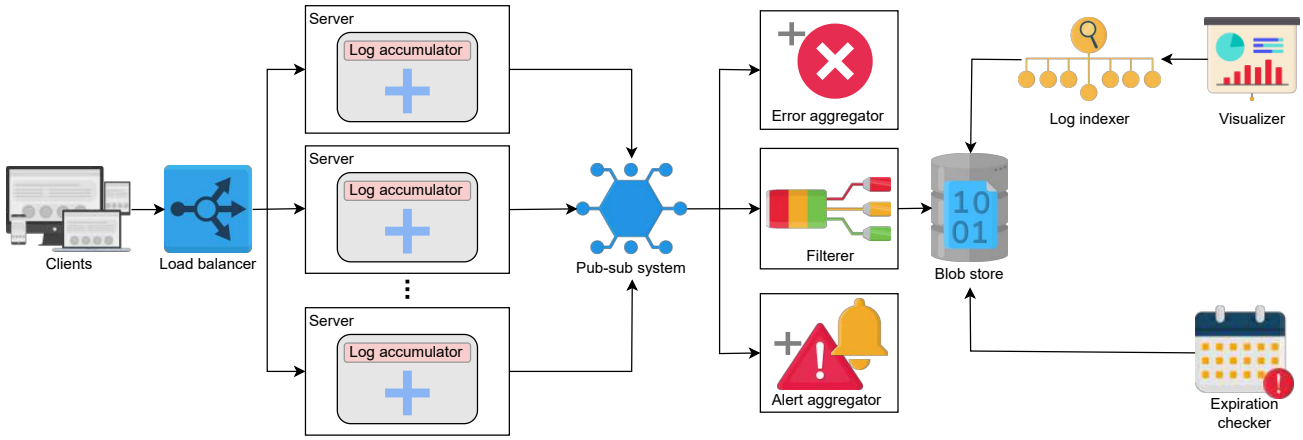
Do we store the logs for a lifetime?

Show Answer

In our design, we have identified another component called the **expiration checker**. It is responsible for these tasks:

- Verifying the logs that have to be deleted. Verifying the logs to store in cold storage.

Moreover, our components log indexer and visualizer work on the blob storage to provide a good searching experience to the end user. We can see the final design of the logging service below:



Logging service design

Point to Ponder

Question

We learned earlier that a simple user-level API call to a large servic might involve hundreds of internal microservices and thousands o nodes. How can we stitch together logs end-to-end for one request with causality intact?

Show Answer

> **Note:** Windows Azure Storage System (WAS) uses an extensive logging infrastructure in its development. It stores the logs in local disks, and given a large number of logs, they do not push the logs to the distributed storage. Instead, they use a grep-like utility that works as a distributed search. This way, they have a unified view of globally distributed logs data.

There can be various ways to design a distributed logging service, but it solely depends on the requirements of our application.

Conclusion

- We learned how logging is crucial in understanding the flow of events in a distributed system. It helps to reduce the mean time to repair (MTTR) by steering us toward the root causes of issues.
- Logging is an I/O-intensive operation that is time-consuming and slow. It is essential to handle it carefully and not affect the critical path of other services' execution.
- Logging is essential for monitoring because the data fetched from logs helps monitor the health of an application. (Alert and error aggregators serve this purpose.)

Back

Introduction to Distributed Logging

Next

System Design: The Distributed Task Sched

Mark as Completed

Requirements of YouTube's Design

Understand the requirements and estimation for YouTube's design.

We'll cover the following

- Requirements
  - Functional requirements
  - Non-functional requirements
- Resource estimation
  - Storage estimation
  - Bandwidth estimation
  - Number of servers estimation
- Building blocks we will use

Requirements

Let’s start with the requirements for designing a system like YouTube.

Functional requirements

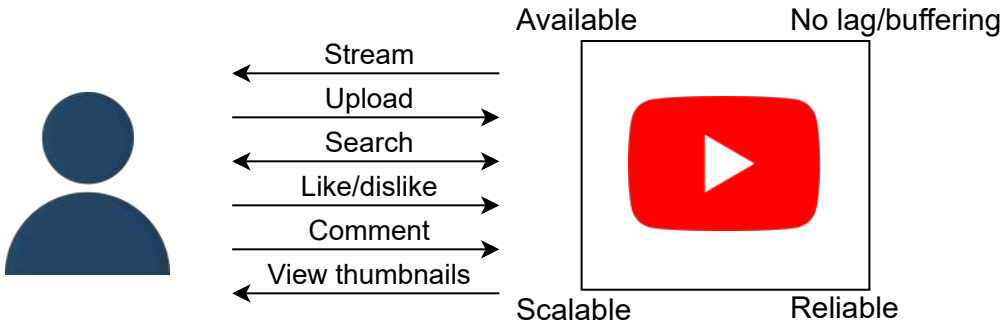
We require that our system is able to perform the following functions:

1. Stream videos
2. Upload videos
3. Search videos according to titles
4. Like and dislike videos
5. Add comments to videos
6. View thumbnails

Non-functional requirements

It's important that our system also meets the following requirements:

- **High availability**: The system should be highly available. High availability requires a good percentage of uptime. Generally, an uptime of 99% and above is considered good.
- **Scalability**: As the number of users grows, these issues should not become bottlenecks: storage for uploading content, the bandwidth required for simultaneous viewing, and the number of concurrent user requests should not overwhelm our application/web server.
- **Good performance**: A smooth streaming experience leads to better performance overall.
- **Reliability**: Content uploaded to the system should not be lost or damaged.



Representation of the functional and non-functional requirements

We don't require strong consistency for YouTube's design. Consider an example where a creator uploads a video. Not all users subscribed to the creator's channel should immediately get the notification for uploaded content.

To summarize, the functional requirements are the features and functionalities that the user will get, whereas the non-functional requirements are the expectations in terms of performance from the syste

Based on the requirements, we'll estimate the required resources and design of our system.

Resource estimation

Estimation requires the identification of important resources that we'll need in the system.

Hundreds of minutes of video content get uploaded to YouTube every minute. Also, a large number of users will be streaming content at the same time, which means that the following resources will be required:

- Storage resources will be needed to store uploaded and processed

- A large number of requests can be handled by doing concurrent processing. This means web/application servers should be in place to serve these users.
- Both upload and download bandwidth will be required to serve millions of users.

To convert the above resources into actual numbers, we assume the following:

- Total number of YouTube users: 1.5 billion.
- Active daily users (who watch or upload videos): 500 million.
- Average length of a video: 5 minutes.
- Size of an average (5 minute-long) video before processing/encoding (compression, format changes, and so on): 600 MB.
- Size of an average video after encoding (using different algorithms for different resolutions like MPEG-4 and VP9): 30 MB.

Storage estimation

To find the storage needs of YouTube, we have to estimate the total number of videos and the length of each video uploaded to YouTube per minute. Let's consider that 500 hours worth of content is uploaded to YouTube in one minute. Since each video of 30 MB is 5 minutes long, we require $\frac{30}{5}$ = 6 MB to store 1 minute of video.

Let's put this in a formula by assuming the following:

$Total_{storage}$: Total storage requirement.

$Total_{upload/min}$: Total content uploaded (in minutes) per minute.

- Example: 500 hours worth of video is uploaded in one minute.

$Storage_{min}$: Storage required for each minute of content

Then, the following formula is used to compute the storage:

$$Total_{storage} = Total_{upload/min} \times Storage_{min}$$

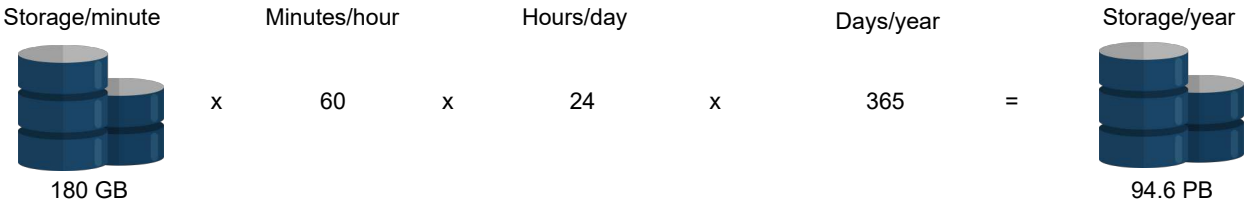
Below is a calculator to help us estimate our required resources. We'll look first at the storage required to persist 500 hours of content uploaded per minute, where each minute of video costs 6 MBs to store:

Storage Required for Storing Content per Minute on YouTube

No. of video hours per minute	Minutes per hour	MB per minute	Storage per minute (GB)
500	60	6	180

Tip

The numbers mentioned above correspond to the compressed version of videos. However, we need to transcode videos into various formats for reasons that we will see in the coming lessons. Therefore, we'll require more storage space than the one estimated above.



Total storage required by YouTube in a year

Quiz

Question

Assuming YouTube stores videos in five different qualities and the average size of a one-minute video is 6 MB, what would the estimated storage requirements per minute be?

Show Answer

Bandwidth estimation

A lot of data transfer will be performed for streaming and uploading videos to YouTube. This is why we need to calculate our bandwidth estimation too

Assume the `upload:view` ratio is `1:300`—that is, for each uploaded video, we have 300 video views per second. We'll also have to keep in mind that when a video is uploaded, it is not in compressed format, while viewed videos can be of different qualities. Let's estimate the bandwidth required for uploading the videos.

We assume:

$Total_{bandwidth}$: Total bandwidth required.

$Total_{content}$: Total content (in minutes) uploaded per minute.

$Size_{minute}$: Transmission required (in MBs) for each minute of content.

Then, the following formula is used to do the computation below:

$$Total_{bandwidth} = Total_{content\_transferred} \times Size_{minute}$$

The Bandwidth Required for Uploading Videos to YouTube

No. of video hours per minute	Minutes per hour	MB per minute	Bandwidth required (Gbps)
500	60	50	200

Show Detailed Calculations

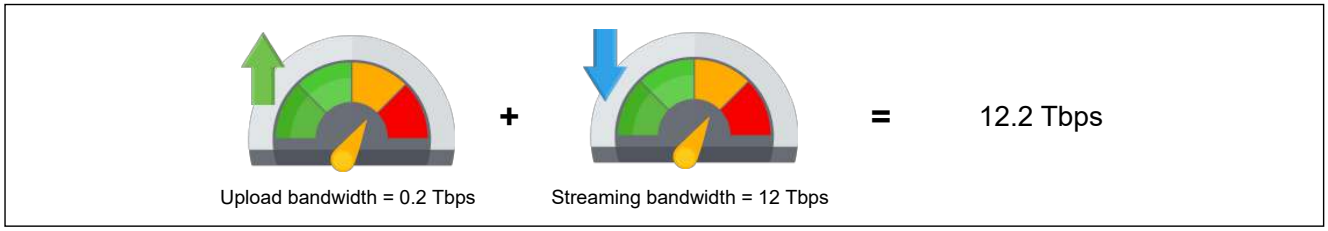
Quiz

Question

If 200 Gbps of bandwidth is required for satisfying uploading needs, how much bandwidth would be required to stream videos? Assume each minute of video requires 10 MB of bandwidth on average.

Hint: The `upload:view` ratio is provided.

Show Answer

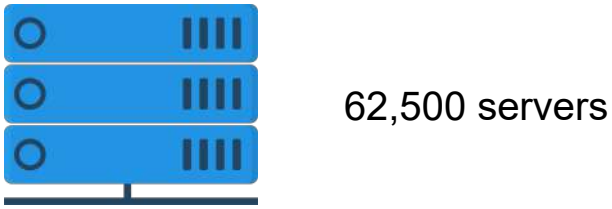


Total bandwidth required by YouTube

Number of servers estimation

We need to handle concurrent requests coming from 500 million daily active users. Let's assume that a typical YouTube server handles 8,000 requests per second.

$$\frac{Number\ of\ active\ users}{Queries\ handled\ per\ server} = 62,500\ servers$$

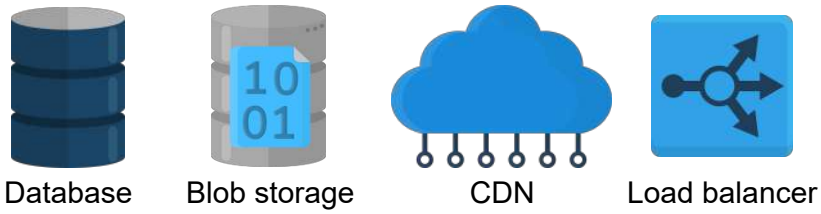


Number of servers required for YouTube

> **Note:** In a real-world scenario, YouTube's design requires storage for thumbnails, users' data, video metadata, users' channel information, and so on. Since the storage requirement for these data sets will not be significant compared to video files, we ignore it for simplicity's sake.

Building blocks we will use

Now that we have completed the resource estimations, let's identify the building blocks that will be an integral part of our design for the YouTube system. The key building blocks are given below:

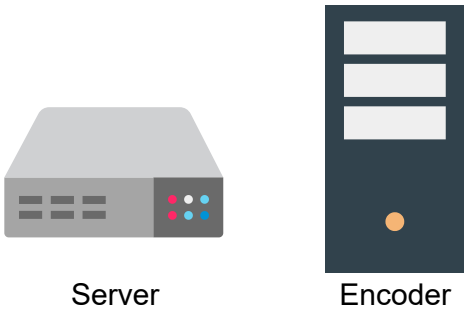


Building blocks in a high-level design

- **Databases** are required to store the metadata of videos, thumbnails, comments, and user-related information.
- **Blob storage** is important for storing all videos on the platform.
- A **CDN** is used to effectively deliver content to end users, reducing delay and burden on end-servers.
- **Load balancers** are a necessity to distribute millions of incoming clients requests among the pool of available servers.

Other than our building blocks, we anticipate the use of the following components in our high-level design:

- Servers are a basic requirement to run application logic and entertain user requests.
- **Encoders** and transcoders compress videos and transform them into different formats and qualities to support varying numbers of devices according to their screen resolution and bandwidth.



Components in YouTube's high-level design

Back

System Design: YouTube

Next

Design of YouTube

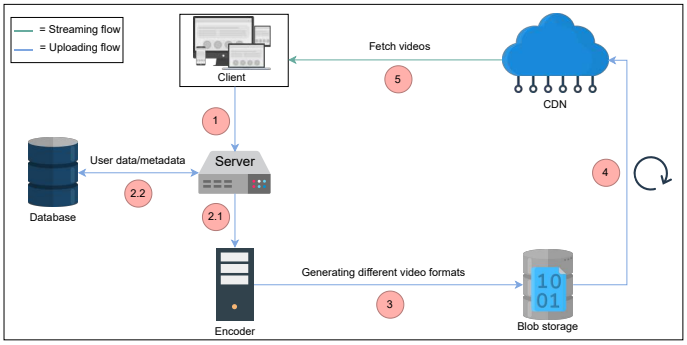
Take a deep dive into YouTube's design.

We'll cover the following

High-level design

The high-level design shows how we'll interconnect the various components we identified in the previous lesson. We have started developing a solution

to support the functional and non-functional requirements with this design.



The high-level design of YouTube

The workflow for the abstract design is provided below:

1. The user uploads a video to the server.
2. The server stores the metadata and the accompanying user data to the database and, at the same time, hands over the video to the encoder for encoding (see 2.1 and 2.2 in the illustration above).
3. The encoder, along with the transcoder, compresses the video and transforms it into multiple resolutions (like 2160p, 1440p, 1080p, and so on). The videos ar stored on blob storage (similar to GFS or S3).
4. Some popular videos may be forwarded to the CDN, which act

as a cache.

5. The CDN, because of its vicinity to the user, lets the user stream the video with low latency. However, CDN is not the only infrastructure for serving videos to the end user, which we will see in the detailed design.

Quiz

Question

Why don't we upload the video directly to the encoder instead of to the server? Doesn't the current strategy introduce an additional delay?

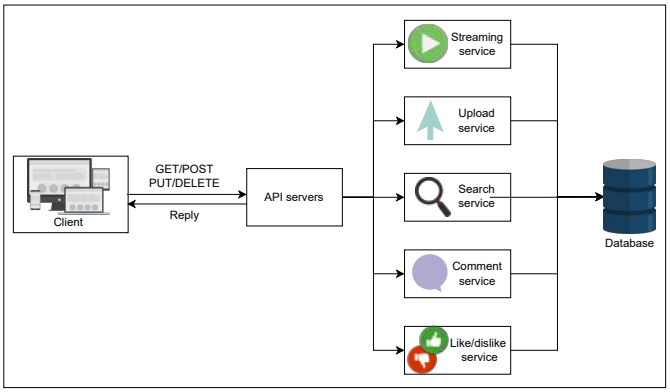
Show Answer

API design

Let's understand the design of APIs in terms of the functionalities we're providing. We'll design APIs to translate our feature set into technica specifications. In this case, REST APIs

can be used for simplicity and speed purposes. Our API design section will help us understand how the client will request services from the back-end application of YouTube. Let's develop APIs for each of the following features:

- Upload videos
- Stream videos
- Search videos
- View thumbnails
- Like or dislike videos
- Comment on videos



API design overview

Upload video

The POST method can upload a video to the `/uploadVideo` API:

```
uploadVideo(user_id, video_file, category_id, title, description, tags, default_language, privacy_settings)
```

Let's take a look at the description of the following parameters here.

Parameter	
`user_id`	This is the user that is upl
`video_file`	This is the video file that
`category_id`	This refers to the category “Entertainment,” “Engine
`title`	This is the title of the vide
`description`	This is the description of
`tags`	This refers to the specific `tags` can improve search
`default_language`	This is the default langua is streamed.
`privacy_settings`	This refers to the privacy asset or private to the upl

> The video file is broken down into smaller packets and store data for a limited time and resume the upload if the user retries. To understand the concept in detail, read more about asynchronous APIs.

Stream video

Back To Course Home

Grokking Modern System Design Interview for Engineers & Managers

16% completed

The GET method is best suited for the `/streamVideo` API:

```
streamVideo(user_id, video_id, screen_resolution, user_bitrate, device_chipset)
```

Some new things introduced in this case are the following parameters:

Parameter	
`screen_resolution`	The server can best optir known.
`user_bitrate`	The transmission capacit quality of video chunks s
`device_chipset`	Many YouTube users wat important to know the h serve the users.

> The server will store different qualities of the same video in its storage and serve users based on their transmission rate.

Search videos

The `/searchVideo` API uses the GET method:

```
searchVideo(user_id, search_string, leng
th, quality, upload_date)
```

Parameter	
`search_string`	This is ahe string use
`length` (optional)	This is used to filter v
`quality` (optional)	This is used to filter v 1080p, and so on.
`upload_date` (optional)	This is used to filter v

View thumbnails

We can use the GET method to access the `/viewThumbnails` API:

```
viewThumbnails(user_id, video_id)
```

Parameter	
`video_id`	This specifies the unique

This API will return the thumbnails o a video in a sequence.

Like and dislike a video

The like and dislike API uses the GET method. As shown below, it's fairly simple.

```
likeDislike(user_id, video_id, like)
```

We can use the same API for the like and dislike functionality. Depending on what is passed as a parameter to the `like` field, we can update the database accordingly—that is, `0` for like and a `1` for dislike.

Comment video

Much like the like and dislike API, we only have to provide the comment string to the API. This API will also use the GET method.

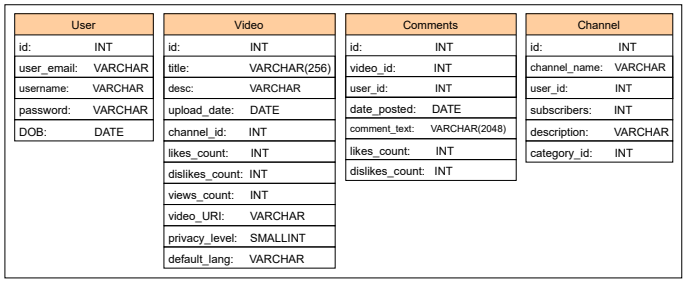
```
commentVideo(user_id, video_id, comment_
text)
```

Parameter	
`comment_text`	This refers to the text tha

Storage schema

Each of the above features in the API design requires support from the

database—we'll need to store the details above in our storage schema to provide services to the API gateway.



Storage schema

> **Note:** Much of the underlying details regarding database tables that can be mapped to services provided by YouTube have been omitted for simplicity. For example, one video can have different qualities and that is not mentioned in the "Video" table.

Detailed design

Now, let's get back to our high-level design and see if we can further explore parts of the design. In particular, the following areas require more discussion:

- **Component integration**: We'll cover some interconnections

between the servers and storage components to better understand how the system will work.

- **Thumbnails**: It's important for users to see some parts of the video through thumbnails. Therefore, we'll add thumbnail generation and storage to the detailed design.
- **Database structure**: Our estimation showed that we require massive storage space. We also require storing varying types of data, such as videos, video metadata, and thumbnails, each of which demands specialized data storage for performance reasons. Understanding the database details will enable us to design a system with the least possible lag.

Let's take a look at the diagram below. We'll explain our design in two steps, where the first looks at what the newly added components are, and the second considers how they coordinate to build the YouTube system.

Detailed design components

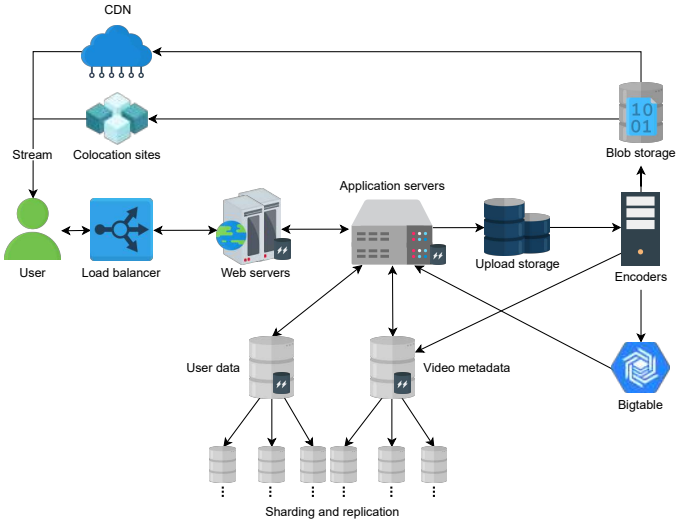
Since we highlighted the requirements of smooth streaming, server-level

details, and thumbnail features, the following design will meet our expectations. Let's explain the purpose of each added component here:

- **Load balancers**: To divide a large number of user requests among the web servers, we require load balancers.
- **Web servers**: Web servers take in user requests and respond to them. These can be considered the interface to our API servers that entertain user requests.
- **Application server**: The application and business logic resides in application servers. They prepare the data needed by the web servers to handle the end users' queries.
- **User and metadata storage**: Since we have a large number of users and videos, the storage required to hold the metadata of videos and the content related to users must be stored in different storage clusters. This is because a large amount of not-so-related data should be decoupled for scalabil purposes.
- **Bigtable**: For each video, we'll require multiple thumbnails.

Bigtable is a good choice for storing thumbnails because of its high throughput and scalability for storing key-value data. Bigtable is optimal for storing a large number of data items each below 10 MB. Therefore, it is the ideal choice for YouTube's thumbnails.

- **Upload storage**: The upload storage is temporary storage that can store user-uploaded videos.
- **Encoders**: Each uploaded video requires compression and transcoding into various formats. Thumbnail generation service is also obtained from the encoders.
- **CDN and colocation sites**: CDNs and colocation sites store popular and moderately popular content that is closer to the user for easy access. Colocation centers are used where it's not possible to invest in a data center facility due to business reasons.



Detailed design of YouTube's components

Design flow and technology usage

Now that we understand the purpose of every component, let's discuss the flow and technology used in different components in the following steps:

1. The user can upload a video by connecting to the web servers. The web server can run Apache or Lighttpd. Lighttpd is preferable because it can serve static pages and videos due to its fast speed.
2. Requests from the web servers are passed onto application servers that can contact various data stores to read or write user, videos, or videos' metadata. The are separate web and application servers because we want to decouple clients' services from the

application and business logic. Different programming languages can be used on this layer to perform different tasks efficiently. For example, the C programming language can be used for encryption. Moreover, this gives us an additional layer of caching, where the most requested objects are stored on the application server while the most frequently requested pages will be stored on the web servers.

3. Multiple storage units are used. Let's go through each of these:
    I. Upload storage is used to store user-uploaded videos before they are temporarily encoded.
    II. User account data is stored in a separate database, whereas videos metadata is stored separately. The idea is to separate the more frequently and less frequently accessed storage clusters from each other for optimal access tim
    We can use MySQL if there are a limited number of concurrent reads and writes. However, as the number of users—and therefore the

number of concurrent reads and writes—grows, we can move towards NoSQL types of data management systems.

III. Since Bigtable is based on Google File System (GFS), it is designed to store a large number of small files with low retrieval latency. It is a reasonable choice for storing thumbnails.

4. The encoders generate thumbnails and also store additional metadata related to videos in the metadata database. It will also provide popular and moderately popular content to CDNs and colocation servers, respectively.
5. The user can finally stream videos from any available site.

> **Note:** Because YouTube is storage intensive, sharding different storage services will effectively come into play as we scale and do frequent writes on the database. At the same time, Bigtable has multiple cache hierarchies. If we combine that with GFS, web- and application-level caching will further

> reduce the request processing latency.

YouTube search

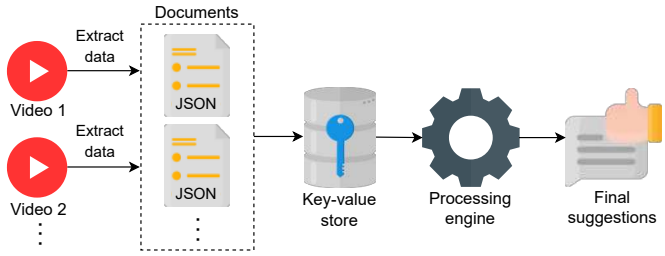
Since YouTube is one of the most visited websites, a large number of users will be using the search feature. Even though we have covered a building block on distributed search, we'll provide a basic overview of how search inside the YouTube system will work.

Each new video uploaded to YouTube will be processed for data extraction. We can use a JSON file to store extracted data, which includes the following:

- Title of the video.
- Channel name.
- Description of the video.
- The content of the video, possibly extracted from the transcripts.
- Video length.
- Categories.

Each of the JSON files can be referred to as a document. Next, keywords wil be extracted from the documents and stored in a key-value store. The *key* in

the key-value store will hold all the keywords searched by the users, while the *value* in the key-value store will contain the occurrence of each key, its frequency, and the location of the occurrence in the different documents. When a user searches for a keyword, the videos with the most relevant keywords will be returned.



An abstraction of how YouTube search works

The approach above is simplistic, and the relevance of keywords is not the only factor affecting search in YouTube. In reality, a number of other factors will matter. The processing engine will improve the search results by filtering and ranking videos. It will make use of other factors like view

Evaluation of YouTube's Design

Let's understand how our design decision fulfills the requirements.

We'll cover the following

Fulfilling requirements

Our proposed design needs to fulfill the requirements we mentioned in the previous lessons. Our main requirements are smooth streaming (low latency), availability, and reliability. Let’s discuss them one by one.

1. **Low latency/Smooth streaming** can be achieved through these strategies:
   - Geographically distributed cache servers at the ISP level to keep t’ most viewed content.
   - Choosing appropriate storage systems for different types of data. For example, we’ll can use Bigtable for thumbnails, blob storage for

videos, and so on.
    - Using caching at various layers via a distributed cache management system.
    - Utilizing content delivery networks (CDNs) that make heavy use of caching and mostly serve videos out of memory. A CDN deploys its services in close vicinity to the end users for low-latency services.
2. **Scalability**: We've taken various steps to ensure scalability in our design as depicted in the table below. The horizontal scalability of web and application servers will not be a problem as the users grow. However, MySQL storage cannot scale beyond a certain point. As we'll see in the coming sections, that may require some restructuring.
3. **Availablity**: The system can be made available through redundancy by replicating data to as many servers as possible to avoid a single point of failure. Replicating data across data centers will ensure high availability, even if an entire data center fails because of power or network issues. Furthermore, local load balancers can exclude any dead servers, and global load balancers can steer traffic to a different region if the need arises.
4. **Reliability**: YouTube's system can be made reliable by using data partitioning and fault-tolerance techniques. Through data partitioning, the non-availability of one type of data will not affect others. We can use redundant hardware and software components for fault tolerance. Furthermore, we can use the heartbeat protocol to monitor the health of servers and omit servers that are faulty and erroneous. We can use a variant of consistent hashing to add or remove servers seamlessly and reduce the burden on specific servers in case of non-uniform load.

Quiz

Question

Isn’t the load balancer a single point of failure (SPOF)?

Show Answer

Requirements	Techniques
Scalability	• Load balancers to multiplex between servers. • Ability to horizontally add (or remove) web servers according to our current needs. • Addition of multiple storage units specific to the required types of data. • Serving from different colocation sites and CDNs. • Separating read/write operations on different servers.
Availability	• Replication of content on different sites. • Important data is persisted on replicated data stores so that we could re-spawn the service in case of major disruption or failures. • Using local and global load balancers.
Performance	• Lighttpd for serving videos/static content. • Caching at each layer (file system, database, cluster, application server, web server). • Addition of multiple storage units specific to the required types of data. • CDNs. • Using an appropriate programming language to perform specific tasks—for example, using C for encryption and Python otherwise.

How YouTube achieves scalability, availability, and good performance

Trade-offs

Let’s discuss some of the trade-offs of our proposed solution.

Consistency

Our solution prefers high availability and low latency. However, strong consistency can take a hit because of high availability (see the CAP theorem). Nonetheless, for a system like YouTube, we can afford to let go of strong consistency. This is because we don’t need to show a consistent feed to all the users. For example, different users subscribed to the same channel may no

see a newly uploaded video at the same time. It's important to mention that we'll maintain strong consistency of user data. This is another reason why we've decoupled user data from video metadata.

Distributed cache

We prefer a distributed cache over a centralized cache in our YouTube design. This is because the factors of scalability, availability, and fault-tolerance, which are needed to run YouTube, require a cache that is not a single point of failure. This is why we use a distributed cache. Since YouTube mostly serves static content (thumbnails and videos), Memcached is a good choice because it is open source and uses the popular Least Recently Used (LRU) algorithm. Since YouTube video access patterns are long-tailed, LRU-like algorithms are suitable for such data sets.

Bigtable versus MySQL

Another interesting aspect of our design is the use of different storage technologies for different data sets. Why did we choose MySQL and Bigtable?

The primary reason for the choice is performance and flexibility. The number of users in YouTube may not scale as much as the number of videos and thumbnails do. Moreover, we require storing the user and metadata in structured form for convenient searching. Therefore, MySQL is a suitable choice for such cases.

However, the number of videos uploaded and the thumbnails for each video would be very large in number. Scalability needs would force us to use a custom or NoSQL type of design for that storage. One could use alternatives to GFS and Bigtable, such as HDFS and Cassandra.

Public versus private CDN

Our design relies on CDNs for low latency serving of the content. However, CDNs can be private or public. YouTube can choose between any one of the two options.

This choice is more of a cost issue than a design issue. However, for areas where there is little traffic, YouTube can use the public CDN because of the following reasons:

1. Setting up a private CDN will require a lot of CAPEX.
2. For rather little viral traffic in certain regions, there will not be enough time to set up a new CDN.
3. There may not be enough users to sustain the business.

However, YouTube can consider building its own CDN if the number of users is too high, since public CDNs can prove to be expensive if the traffic is high. Private CDNs can also be optimized for internal usage to better serve customers.

Duplicate videos

The current YouTube design doesn't handle duplicate videos that have been

leads to a trade-off. As a result, we either waste storage space or face an additional complexity to the upload process for handling duplicate videos.

Let's perform some calculations to resolve this problem. Assume that 50 out of 500 hours of videos uploaded to YouTube are duplicates. Considering that one minute of video requires 6 MB of storage space, the duplicated content will take up the following storage space:

```
(50 x 60) minutes x 6 MB/min = 18 GB
```

If we avoid video duplication, we can save up to 9.5 petabytes of storage space in a year. The calculations are as follows:

```
18 GB/min x (60 x 24 x 365) total minutes in an year = 9.5 Peta Bytes
```

Storage space being wasted, and other computational costs are not the only issues with duplicate videos. An important aspect of duplicate videos is the copyright issue. No content creator would want their content plagiarized. Therefore, it's plausible to add the complexity of handling duplicate videos to the design of YouTube.

Duplication can be solved with simple techniques like locality-sensitive hashing. However, there can be complex techniques like Block Matching Algorithms (BMAs) and phase correlation to find duplications. Implementing this solution can be quite complex in a huge database of videos. We may have to use technologies like artificial intelligence (AI).

Future scaling

So far, we've focused on the design and analysis of the proposed design for YouTube. In reality, the design of YouTube is quite complex and requires advanced systems. In this section, we'll focus on the pragmatic structure of data stores and the web server.

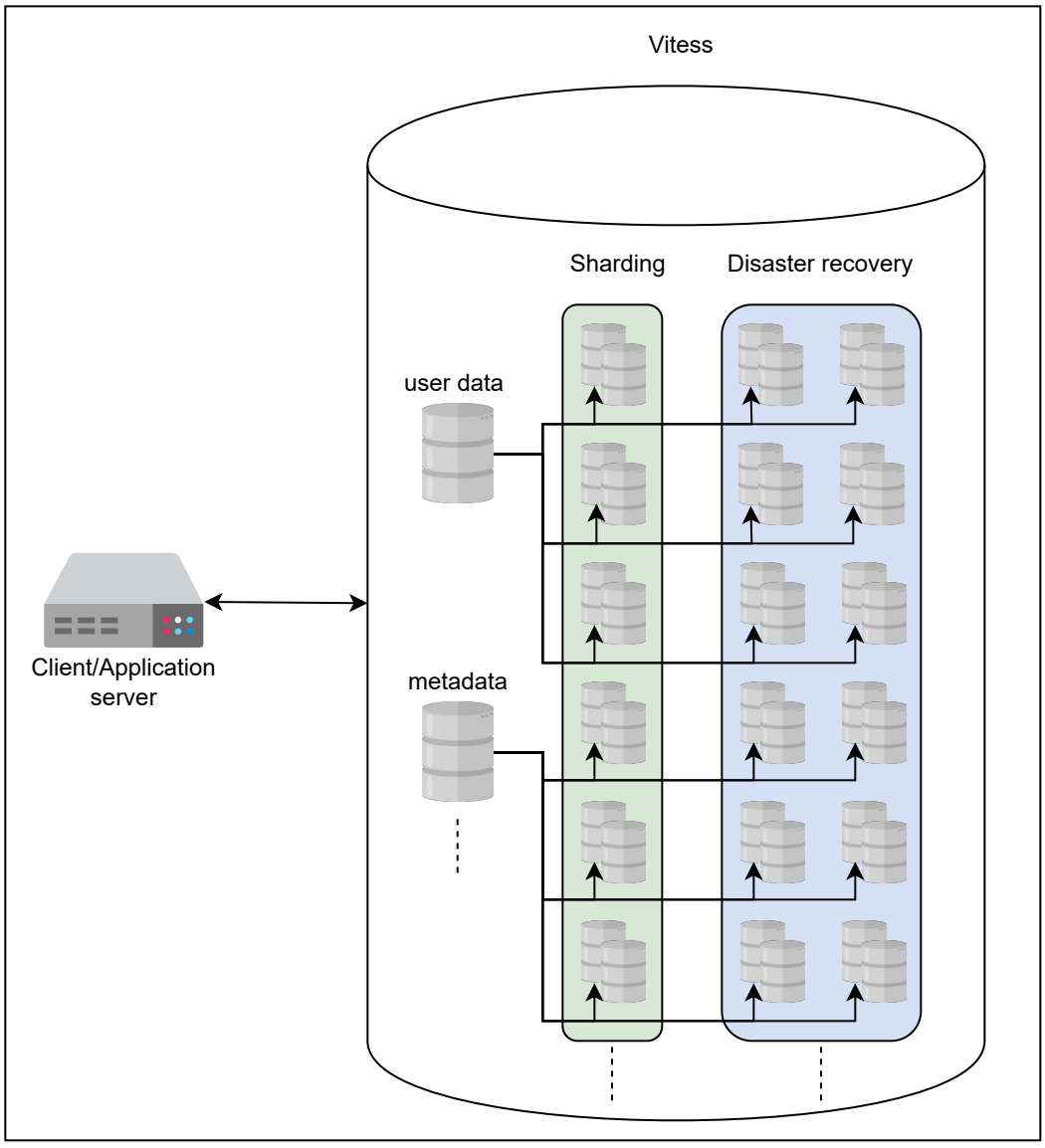
We'll begin our discussion with some limitations in terms of scaling YouTube. In particular, we'll consider what design changes we'll have to make if the traffic load to our service goes up by, say, a few folds.

We already know that we'll have to scale our existing infrastructure, which includes the below elements:

- Web servers
- Application servers
- Datastores
- Placing load balancers among each of the layers above
- Implementing distributed caches

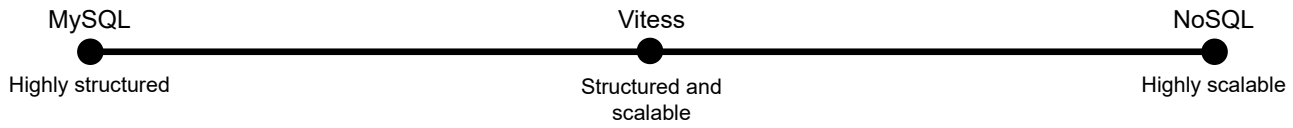
Any infrastructure mentioned above requires some modifications and adaptation to the application-level logic. For example, if we continue to increase our data in MySQL servers, it can become a choke point. To effectively use a sharded database, we might have to make changes to our database client to achieve a good level of performance and maintain the ACID (atomicity, consistency, isolation, durability) properties. However, even if we continue to change to the database client as we scale, its complexity may reach a point where it is no longer manageable. Also note that we haven't incorporated a disaster recovery mechanism into our design yet.

To resolve the problems above, YouTube has developed a solution called Vitess.



Vitess system for scalability

The key idea in Vitess is to put an abstraction on top of all the database layers, giving the database client the illusion that it is talking to a single database server. The single database in this case is the Vitess system. Therefore, all the database-client complexity is migrated to and handled by Vitess. This maintains the ACID properties because the internal database in use is MySQL. However, we can enable scaling through partitioning. Consequently, we'll get a MySQL structured database that gives the performance of a NoSQL storage system. At the same time, we won't have to live with a rich database client (application logic). The following illustration highlights how Vitess is able to achieve both scalability and structure.



Vitess on the scalability versus structure spectrum

One could imagine using techniques like data denormalization instead of the Vitess system. However, data denormalization won't work because it comes at the cost of reduced writing performance. Even if our work is read-intensive, as the system scales, writing performance will degrade to an unbearable limit.

Web server

A **web server** is an extremely important component and, with scale, a custom web server can be a viable solution. This is because most commercial or open-source solutions are general purpose and are develop with a wide range of users in mind. Therefore, a custom solution for such a successful service is desirable.

Let's try an interesting exercise to see which server YouTube currently use Click on the terminal below and execute the following command:

```
lynx -head -dump http://www.youtube.com | grep ^Server
```

Terminal 1

Terminal

Click to Connect...

> **Note:** ESF is a custom web server developed by Google, and as of early 2022, it is widely in use in the Google ecosystem because out-of-the-box solutions were not enough for YouTube's needs.

Back

Design of YouTube

Next

The Reality Is More Complica

Completed

Requirements of Quora's Design

Learn about the requirements for designing Quora.

We'll cover the following

Requirements

Let’s understand the functional and non-functional requirements below:

Functional requirements

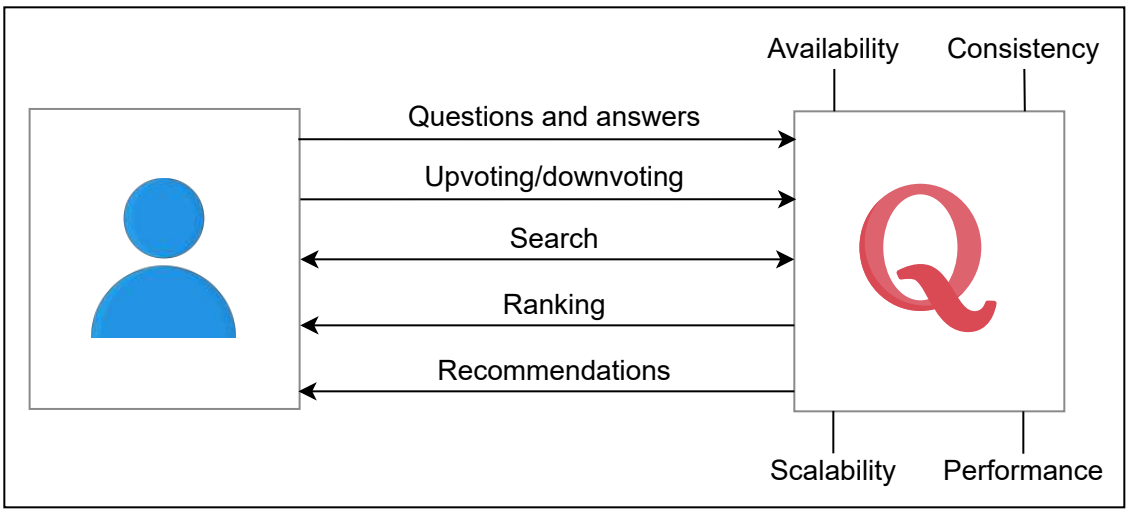
A user should be able to perform the following functionalities:

- **Questions and answers**: Users can ask questions and give answers. Questions and answers can include images and videos.
- **Upvote/downvote and comment**: It is possible for users to upvote, downvote, and comment on answers.
- **Search**: Users should have a search feature to find questions already asked on the platform by other users.

- **Recommendation system**: A user can view their feed, which includes topics they're interested in. The feed can also include questions that need answers or answers that interest the reader. The system should facilitate user discovery with a recommender system.
- **Ranking answers**: We enhance user experience by ranking answers according to their usefulness. The most helpful answer will be ranked highest and listed at the top.

Non-functional requirements

- **Scalability**: The system should scale well as the number of features and users grow with time. It means that the performance and usability should not be impacted by an increasing number of users.
- **Consistency**: The design should ensure that different users' views of the same content should be consistent. In particular, critical content like questions and answers should be the same for any collection of viewers. However, it is not necessary that all users of Quora see a newly posted question, answer, or comment right away.
- **Availability**: The system should have high availability. This applies to cases where servers receive a large number of concurrent requests.
- **Performance**: The system should provide a smooth experience to the user without a noticeable delay.



Functional and non-functional requirements of Quora

Resource estimation

In this section, we'll make an estimate about the resource requirements for Quora service. We'll make assumptions to get a practical and tractable estimate. We'll estimate the number of servers, the storage, and the bandwidth required to facilitate a large number of users.

Assumptions: It is important to base our estimation on some underlying assumptions. We, therefore, assume the following:

- There are a total of 1 billion users, out of which 300 million are daily active users.
- Assume 15% of questions have an image, and 5% of questions have a video embedded in them. A question cannot have both at the same time.
- We'll assume an image is estimated to be 250 KBs, and a video is considered 5 MBs.

Number of servers estimation

Let's estimate our requests per second (RPS) for our design. If there are an average of 300 million daily active users and each user can generate 20 requests per day, then the total number of requests in a day will be:

$$300 \times 10^6 \times 20 = 6 \times 10^9$$

Therefore, the RPS = $\frac{6 \times 10^9}{86400} \approx 69500$ approximately requests per second.

Estimating RPS

Daily active users	300	million
Requests per day per user	20	

Requests Per Second (RPS)	f 69444

We already established in the back-of-the-envelope calculations chapter that we'll use the following formula to estimate a pragmatic number of servers:

$$\frac{Number\ of\ daily\ active\ users}{RPS\ of\ a\ server} = \frac{300 \times 10^6}{8000} = 37500$$

The estimated number of servers required for Quora

> Therefore, the total number of servers required to facilitate 300 million users generating an average of 69,500 requests per second will be 37,500.

Storage estimation

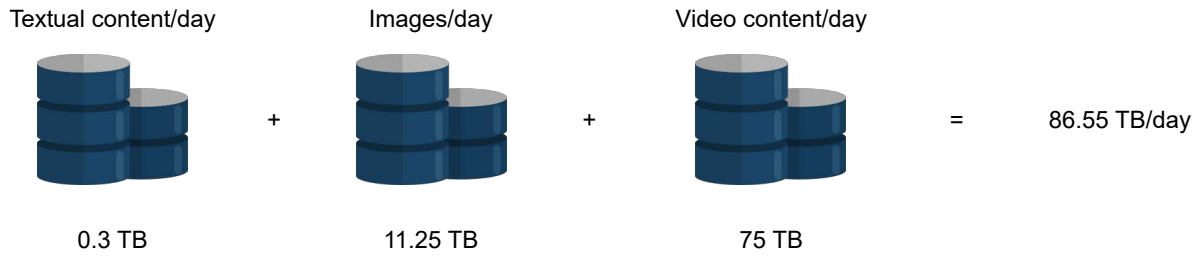
Let's keep in mind our assumption that 15% of questions have images and 5% have videos. So, we'll make the following assumptions to estimate the storage requirements for our design:

- Each of the 300 million active users posts 1 question in a day, and each question has 2 responses on average, 10 upvotes, and 5 comments in total.
- The collective storage required for the textual content of one question equals 1 KB.

Storage Requirements Estimation Calculator

Questions per user	1	per day
Total questions per day	*f* 300	millions
Size of textual content per question	1	KB
Image size	250	KB
Video size	5	MB
Questions containing images	15	percent
Questions containing videos	5	percent
Storage for textual content	*f* 0.3	TB
Storage for image content	*f* 11.25	TB
Storage for video content	*f* 75	TB

See Detailed Calculations



Summarizing storage requirements of Quora

> Total storage required for one day =
> $0.3TB + 11.25TB + 75TB \ = 86.55TB$ per day

The daily storage requirements of Quora seem very high. But for service with 300 million DAU, a yearly requirement of $86.55TB \times 365 = 31.6PB$ is feasible. The practical requirement will be much higher because we have disregarded the storage required for a number of things. For example, non-active (out of $1B$) users' data will require storage.

Bandwidth estimation

The bandwidth estimate requires the calculation of incoming and outgoing data through the network.

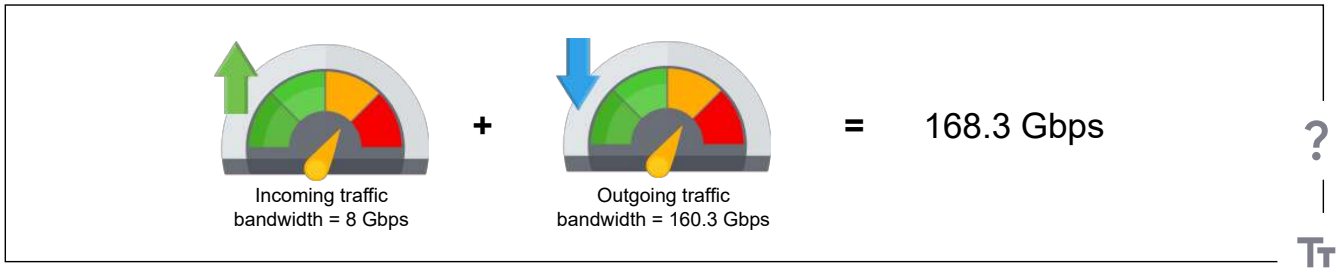
- **Incoming traffic**: The incoming traffic bandwidth required per day will be equal to $\frac{86.55TB}{86400} \times 8 = 8Gbps$
- **Outgoing traffic**: We have assumed that 300 million active users views 20 questions per day, so the total bandwidth requirements can be found in the below calculator:

Bandwidth Requirements Estimation Calculator

Total storage required per day	86.55	TB

Incoming traffic band width	f 8	Gbps
Questions viewed per user	20	per day
Total questions viewed	f 69444	per second
Bandwidth for text of all questions	f 0.56	Gbps
Bandwidth for 15% of image content	f 20.83	Gbps
Bandwidth for 5% of video content	f 138.89	Gbps
Outgoing traffic band width	f 160.3	Gbps

Detailed Calculations



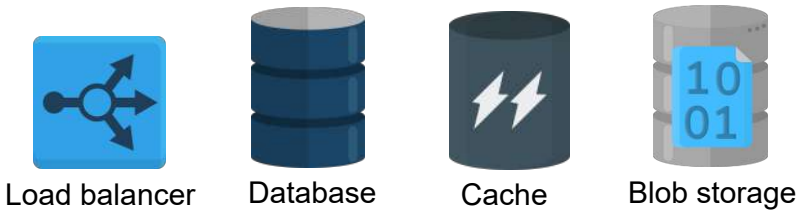
Summarizing the bandwidth requirements of Quora

Total bandwidth requirement of Quora is equal to:

> Incoming + outgoing traffic bandwidth
> $= 8Gbps + 160.3Gbps = 168.3Gbps$

Building blocks we will use

We'll use the following building blocks for the initial design of Quora:



Building blocks required for our design

- **Load balancers** will be used to divide the traffic load among the service hosts.
- **Databases** are essential for storing all sorts of data, such as user questions and answers, comments, and likes and dislikes. Also, user data will be stored in the databases. We may use different types of databases to store different data.
- **A distributed caching system** will be used to store frequently accessed data. We can also use caching to store our view counters for different questions.
- **The blob store** will keep images and video files.

Back
System Design: Quora
Next
Initial Design of Q
Completed

Initial Design of Quora

Transform the Quora requirements into a high-level design.

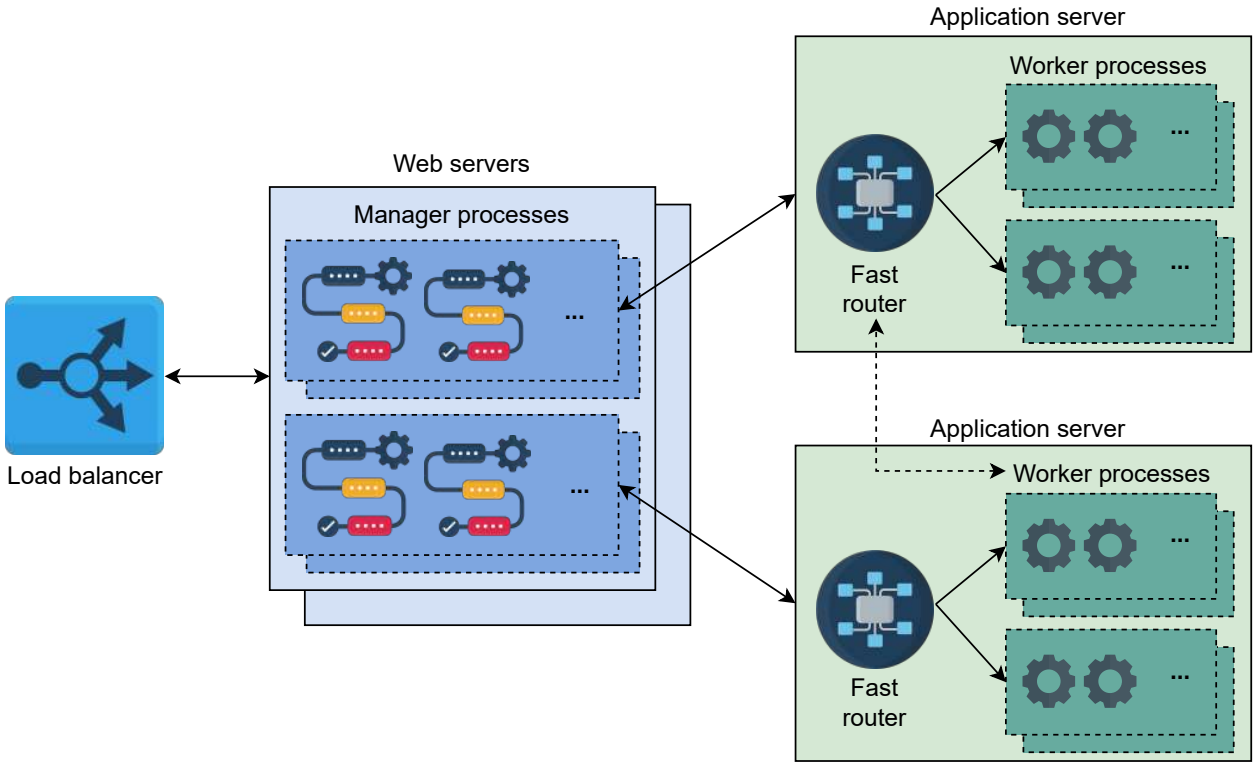
We'll cover the following

Initial design

The initial design of Quora will be composed of the following building blocks and components:

- **Web and application servers**: A typical Quora page is generated by various services. The web and application servers maintain various processes to generate a webpage. The web servers have manager processes and the application servers have worker processes for handling various requests. The manager processes distribute work among the worker processes using a router library. The router library enqueued with tasks by the manager processes and dequeued by worker processes. Each application server maintains several in-memo

queues to handle different user requests. The following illustration provides an abstract view of web and application servers:



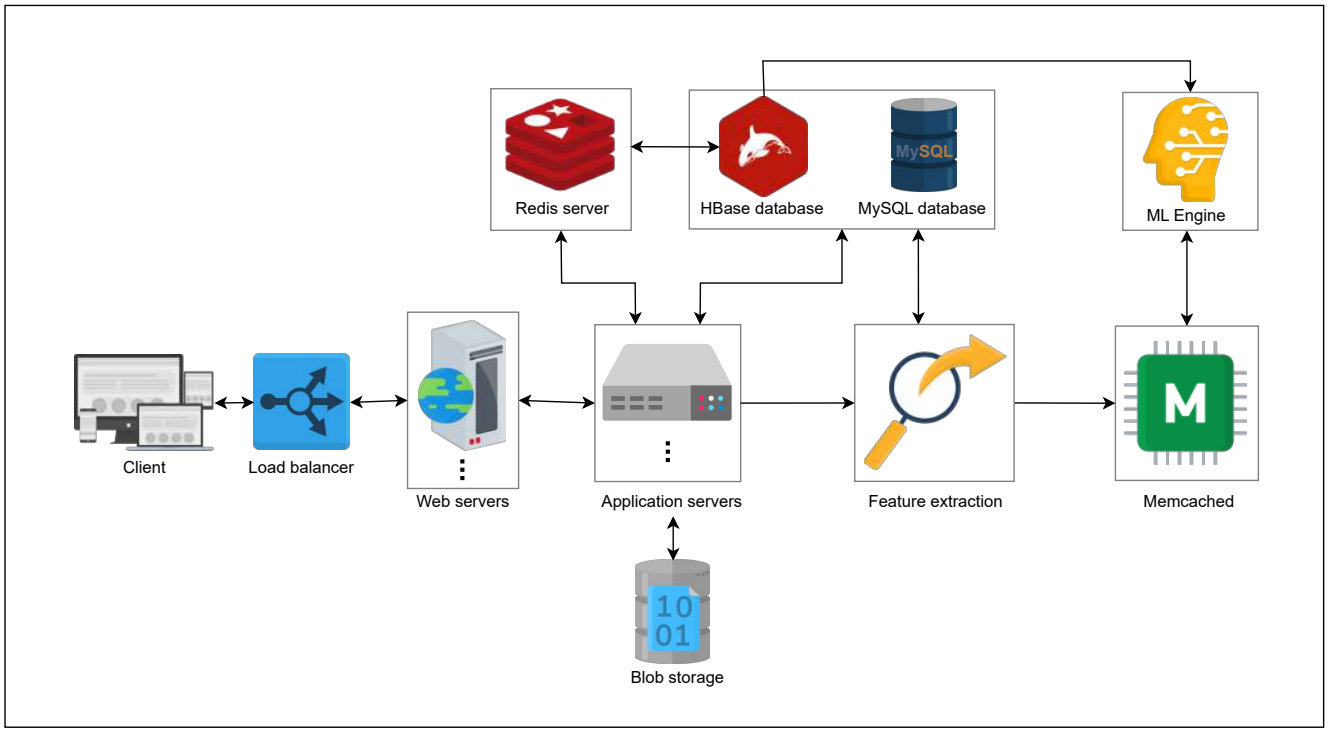
Web and application servers at Quora

- **Data stores**: Different types of data require storage in different data stores. We can use critical data like questions, answers, comments, and upvotes/downvotes in a relational database like MySQL because it offers a higher degree of consistency. NoSQL databases like HBase can be used to store the number of views of a page, scores used to rank answers, and the extracted features from data to be used for recommendations later on. Because recomputing features is an expensive operation, HBase can be a good option to store and retrieve data at high bandwidth. We require high read/write throughput because big data processing systems use high parallelism to efficiently get the required statistics. Also, blob storage is required to store videos and images posted in questions and answers.

Why HBase?

- **Distributed cache**: For performance improvement, two distributed cache systems are used: Memcached and Redis. Memcached is primarily used to store frequently accessed critical data that is otherwise stored in MySQL. On the other hand, Redis is mainly used to store an online view counter of answers because it allows in-store increments. Therefore, two cache systems are employed according to their use case. Apart from these two, CDNs serve frequently accessed videos and images.
- **Compute servers**: A set of compute servers are required to facilitate features like recommendations and ranking based on a set of attributes. These features can be computed in online or offline mode. The compute servers use machine learning (ML) technology to provide effective recommendations. Naturally, these compute servers have a substantially high amount of RAM and processing power.

Of course, other basic building blocks like load balancers, monitoring services, and rate limiters will also be part of the design. A high-level design is provided below:



The high-level design of Quora

Workflow

The design of Quora is complex because we have a large number of functional and non-functional requirements. Therefore, we'll explain the workflow on the basis of each feature:

- **Posting question, answers, comments**: The web servers receive user requests through the load balancer and direct them to the application servers. Meanwhile, the web servers generate part of the web page and let the worker process in the application servers do the rest of the page generation. The questions and answers data is stored in a MySQL database, whereas any videos and images are stored in the blob storage. A similar approach is used to post comments and upvote or downvote answers. Task prioritization is performed by employing different queues for different tasks. We perform prioritization because certain tasks require immediate attention—for example, fetching data from th database for a user request—while others are not so urgent—for

example, sending a weekly email digest. The worker processes will perform tasks by fetching from these queues.

Points to Ponder

Question 1

When would a service like Quora require a notifications feature?

Show Answer

1 of 2

- **Answer ranking system**: Answers to questions can be sorted based on date. Although it is convenient to develop a ranking system on the basis of date (using time stamps), users prefer to see the most appropriate answer at the top. Therefore, Quora uses ML to rank answers. Different features are extracted over time and stored in the HBase for each type of question. These features are forwarded to the ML engine to rank the most useful answer at the top. We cannot use the number of upvotes as the only metric for ranking answers because a good number of answers can be jokes—and such answers also get a lot of upvotes. It is good to implement the ranking system offline because good answers get upvotes and views over time. Also, the offline mode poses a lesser burden on the infrastructure. Implementing the ranking system offlin and the need for special ML hardware makes it suitable to use some public cloud elastic services.

- **Recommendation system**: The recommendation system is responsible for several features. For example, we might need to develop a user feed, find related questions and ads, recommend questions to potential respondents, and even highlight duplicate content and content in violation of the service's terms of use. Unlike the answer ranking system, the recommendation system must provide both online and offline services. This system receives requests from the application server and forwards selected features to the ML engine.
- **Search feature**: Over time, as questions and answers are fed to the Quora system, it is possible to build an index in the HBase. User search queries are matched against the index, and related content is suggested to the user. Frequently accessed indexes can be served from cache for low latency. The index can be constructed from questions, answers, topics labels, and usernames. Tokenization of the search index returns the same results for reordered words also (see Scaling Search and Indexing in **Distributed Search** chapter for more details).

Fun Facts

API design

We'll design the API calls for Quora in this section. We'll define APIs for the following features only:

- Post a question
- Post an answer
- Upvote or downvote a question or answer
- Comment on an answer
- Search

> **Note:** We don't consider APIs for a recommendation system or ranking because they are not placed as an explicit request by the user. Instead, the web server coordinates with other components to ensure the service.

Post a question

The POST method of HTTP is used to call the `/postQuestion` API:

```
postQuestion(user_id, question, description, topic_label, video, image)
```

Let's understand each parameter of the API call:

Parameter	Description
`user_id`	This is the unique identification of the user that posts the question.
`question`	This is the text of the question posed by the user.
`description`	This is the description of a question. This is an optional field.
`topic_label`	This represents a list of domains to which the user's question is related.
`video`	This is a video file embedded in a user question.
`image`	This is an image that is a part of a user question.

The video and image parameters can be `NULL` if no image or video is embedded within the question. Otherwise, it is uploaded as part of the question.

Post an answer

For posting an answer, the POST method is a suitable choice for `/postAnswer` API:

```
postAnswer(user_id, question_id, answer_text, video, image)
```

Parameter	Description
`question_id`	This refers to the question the answer is posted against.
`answer_text`	This is the textual answer posted by the responder.

The rest of the parameters are self-explanatory.

Upvote an answer

The `/upvote` API is below:

```
upvote(user_id, question_id, answer_id)
```

Parameter	Description
`user_id`	This represents the user upvoting the answer.
`answer_id`	This represents the identity of the answer that is upvoted for a particular question, which is identified by the `question_id`.

> **Note:** The downvote API is the same as the upvote API because both are similar functionalities.

Comment on an answer

The `/comment` API has the following structure:

```
comment(user_id, answer_id, comment_text)
```

Parameter	Description
`user_id`	It represents the user commenting on the answer.
`comment_text`	It represents the text a user posts against an answer identified by the `answer_id`.

Search

The `/search` API has the following details:

```
search(user_id, search_text)
```

Parameter	Description
`user_id`	This is the `user_id` performing the search query. It is optional in this case because a non-registered user can also search for questions.
`search_text`	This is the search query entered by a user.

> We use a sequencer to generate the different IDs mentioned in the API calls.

Point to Ponder

Question

Final Design of Quora

Learn about the limitations of Quora's design and improve the design.

We'll cover the following

- Limitations of the proposed design
- Detailed design of Quora
  - Service hosts
  - Vertical sharding of MySQL
  - MyRocks
  - Kafka
  - Technology usage

Limitations of the proposed design

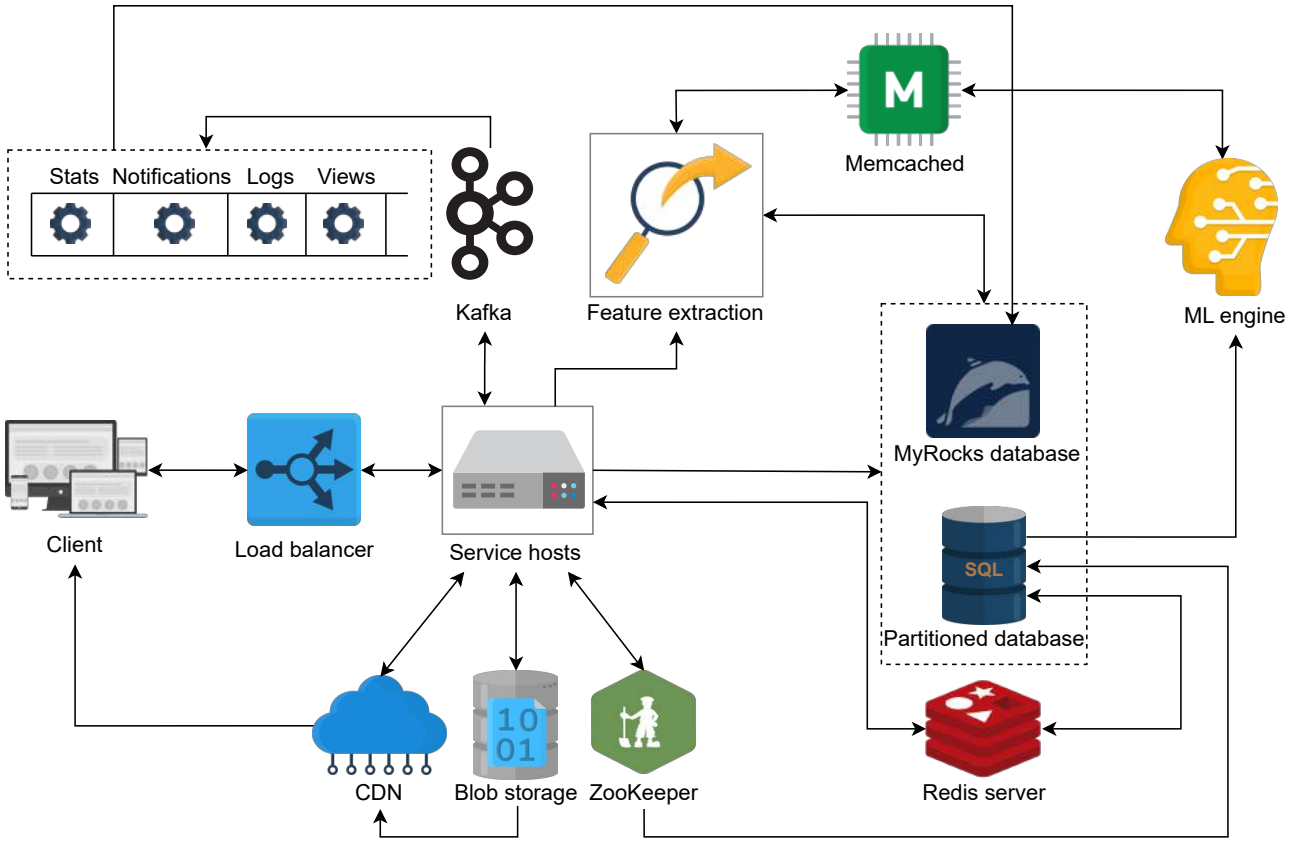
The proposed design serves all the functional requirements. However, it has a number of serious drawbacks that emerge as we scale. This means that we are unable to fulfill the non-functional requirements. Let's explore the main shortcomings below:

- **Limitations of web and application servers**: To entertain the user's request, payloads are transferred between web and application servers. [illegible] the web from application servers (that is, the manager and worker processes), the added latency due to an additional network link erodes user's experience. Apart from data transfer, control communication

between the router library with manager and worker processes also imposes additional performance penalties.

- **In-memory queue failure**: The internal architecture of application servers log tasks and forward them to the in-memory queues, which serve them to the workers. These in-memory queues of different priorities can be subject to failures. For instance, if a queue gets lost, all the tasks in that queue are lost as well, and manual engineering is required to recover those tasks. This greatly reduces the performance of the system. On the other hand, replicating these queues requires increasing RAM size. Also, with the number of features (functional requirements) that our system offers, many tasks can get assembled, which results in insufficient memory. At the same time, it is not desirable to choke application servers with not-so-urgent tasks. For example, application servers should not be burdened with tasks like storing view counts for answers, adding statistics to the database for later analysis, and so on.
- **Increasing QPS on MySQL**: Because we have a higher number of features offered by our system, few MySQL tables receive a lot of user queries. This results in a higher number of QPS on certain MySQL servers, which can result in higher latency. Furthermore, there is no scheme defined for disaster recovery management in our design.
- **Latency of HBase:** Even though HBase allows high real-time throughput, its P99 latency is not among the best. A number of Quora features require the ML engine that has a latency of its own. Due to the addition of the higher latency of HBase, the overall performance of the system degrades over time.

The issues highlighted above require changes to the earlier proposed desig
Therefore, we'll make the following adjustments and update our design:



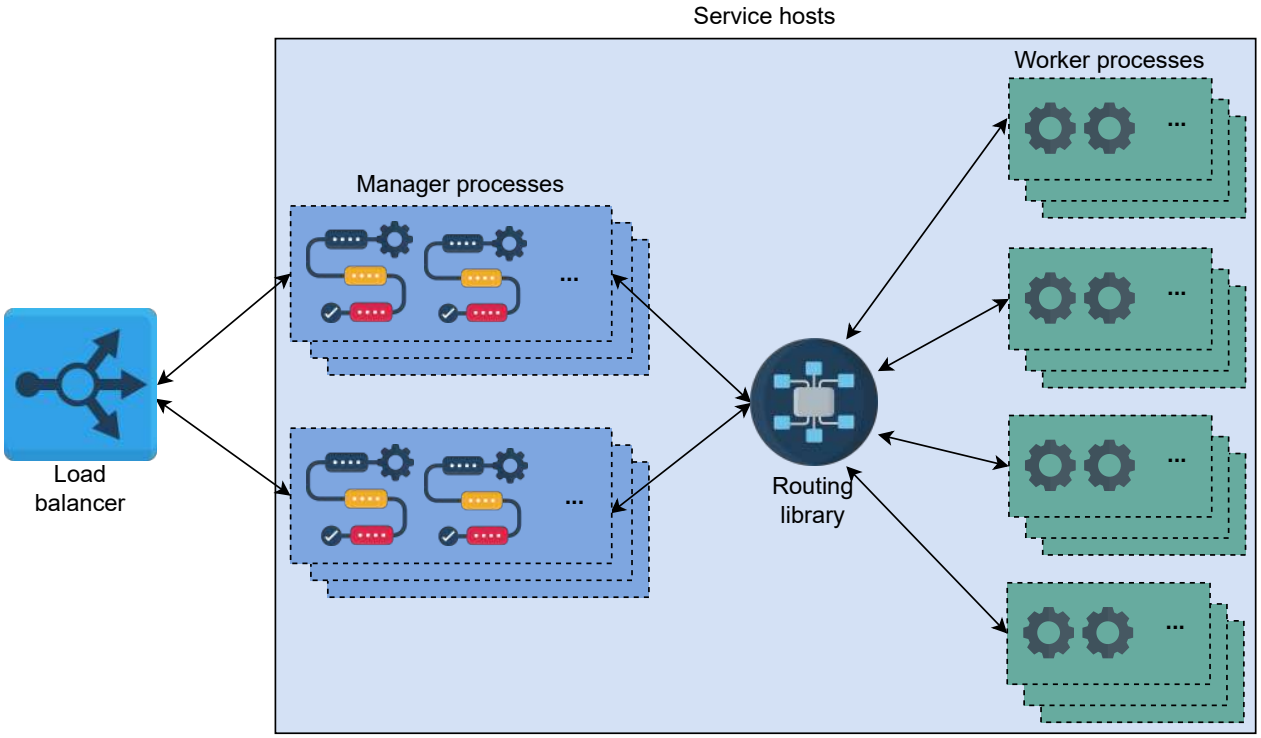
Detailed design of Quora

Detailed design of Quora

Let’s understand the improvements in our design:

Service hosts

We combine the web and application servers within a single powerful machine that can handle all the processes at once. This technique eliminates the network I/O and the latency introduced due to the network hops required between the manager, worker, and routing library processes. The illustration below provides an abstract view of the updated web server architecture:



The updated design, where web and application servers are combined in the service host

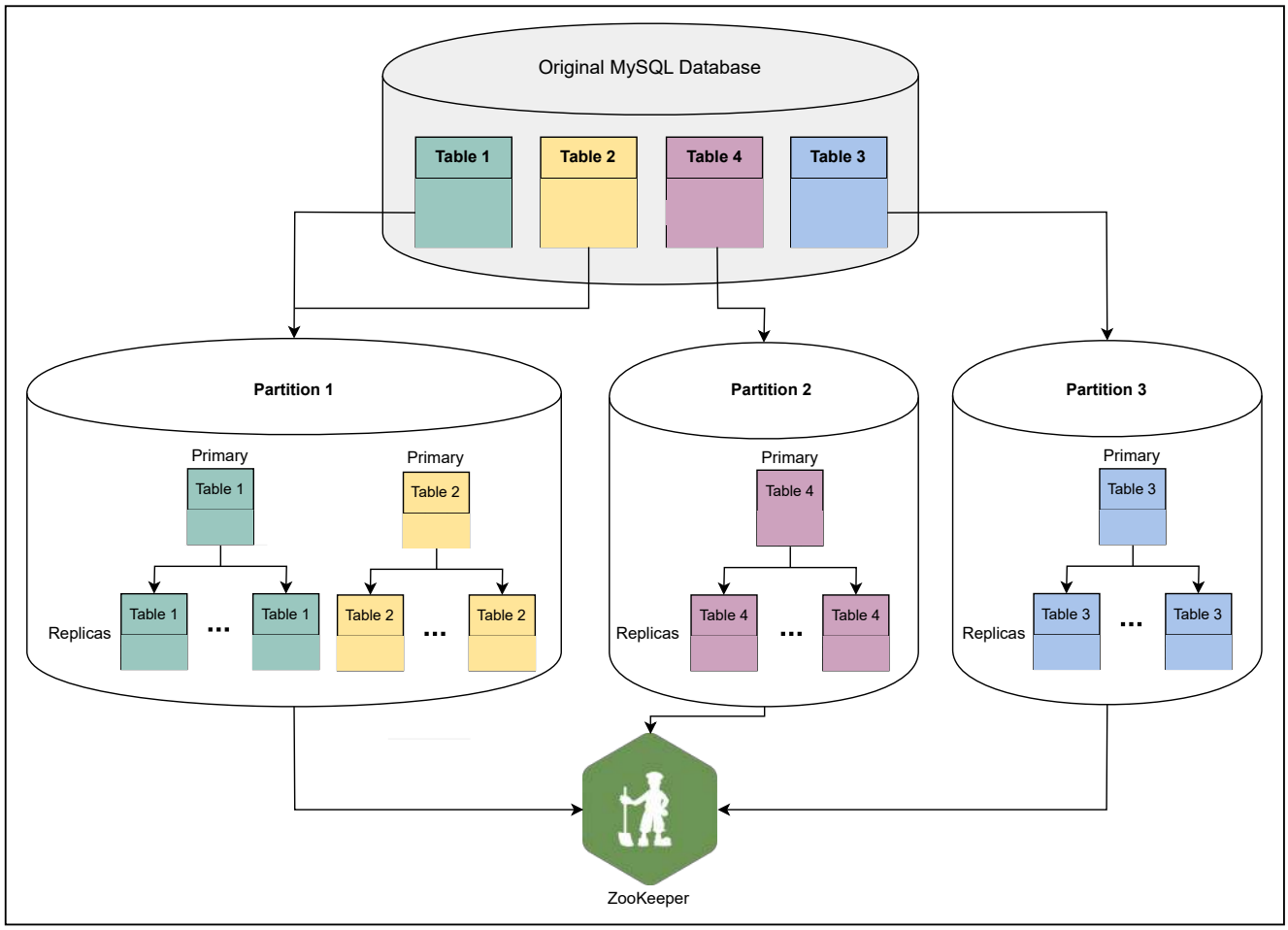
Vertical sharding of MySQL

Tables in the MySQL server are converted to separate shards that we refer to as **partitions**. A partition has a single primary server and multiple replica servers.

The goal is to improve performance and reduce the load due to an increasing number of queries on a single database table. To achieve that, we do vertical sharding in two ways:

1. We split tables of a single database into multiple partitions. The concept is depicted in Partitions 2 and 3, which embed Tables 4 and 3, respectively.
2. We combine multiple tables into a single partition, where `join` operations are anticipated. The concept is depicted in Partition 1, whi embeds Tables 1 and 2.

Therefore, we are able to co-locate related data and reduce traffic on hot data. The illustration below depicts vertical sharding at Quora.



The architecture of vertical sharding at Quora

After we complete the partitioning, we require two types of mappings or metadata to complete our scaling process:

1. Which partitions contain which tables and columns?
2. Which hosts are primary and replicas of a particular partition?

Both of these mappings are maintained by a service like ZooKeeper.

The sharded design above ensures scalability because we are able to locate related data in a single partition, and therefore it eliminates the need for querying data from multiple shards. Also, the number of read-replicas can be increased for hot shards, or further sharding may be performed. For edg

cases where joining may be needed, we can perform it at the application level.

> **Note:** Vertical sharding is of particular interest in Quora's design because horizontal sharding is more common in the database community. The main idea behind vertical sharding is to achieve scalability by carefully dividing or re-locating tables and eliminating join operations across different shards. Nevertheless, a vertically sharded partition or table can grow horizontally to an extent where horizontal sharding will be necessary to retain acceptable performance.

MyRocks

The new design embeds MyRocks as the key-value store instead of HBase. We use the MyRocks version of RocksDB for two main reasons:

1. MyRocks has a lower p99 latency instead of HBase. Quora claims to have reduced P99 latency from 80 ms to 4 ms using MyRocks.
2. There are operational tools that can transfer data between MyRocks and MySQL.

> **Note:** Quora serves the ML compute engine by extracting features from questions and answers stored in MySQL. In this case, the operational tools come in handy to transfer data between MyRocks and MySQL.

Kafka

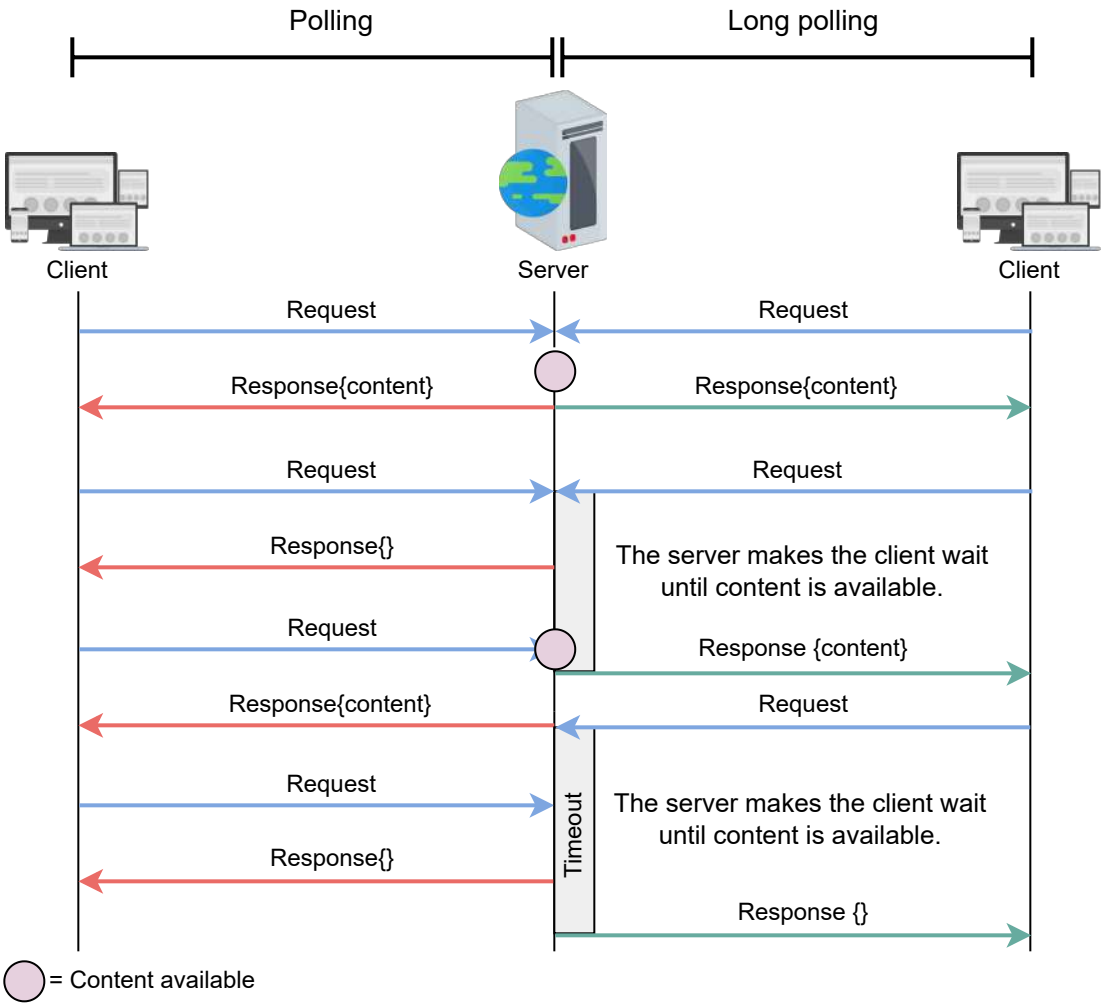
Our updated design reduces the request load on service hosts by separating not-so-urgent tasks from the regular API calls. For this purpose, we use Kafka, which can disseminate jobs among various queues for tasks such as the view counter (see Sharded Counters), notification system, analytics, and highlight topics to the user. Each of these jobs is executed through cron jobs.

Technology usage

Services that scale quickly have little time to develop new features and handle an increasing number of requests from users. Such services employ cloud infrastructure to handle spikes in traffic. Also, the choice of programming language is important. Just like we mentioned that YouTube chose Python for faster programming, we can apply the same logic to Quora. In fact, Quora uses the Python Paste web framework.

It is desirable to use a faster programming language like C++ to develop the feature extraction service. For online recommendation services through a ML engine, feature extraction service should be quick, to enable the ML engine to accomplish accurate recommendations. Not only that, but reducing the latency burden on the ML engine allows it to provide a larger set of services. We can employ the Thrift service to support interoperability between programming languages within different components.

Features like comments, upvotes, and downvotes require frequent page updates from the client side. **Polling** is a technique where the client (browser) frequently requests the server for new updates. The server may or may not have any updates but still responds to the client. Therefore, the server may get uselessly overburdened. To resolve this issue, Quora uses a technique called **long polling**, where if a client requests for an update, the server may not respond for as long as 60 seconds if there are no updates. However, if there is an update, the server will reply immediately and allow the client to make new requests.



Polling vs. long polling

Lastly, Memcached can employ `multiget()` to obtain multiple keys from the cache shards to reduce the retrieval latency of multiple keys.

> **Note:** Quora has employed AWS to set up a good number of its infrastructure elements, including S3 (see the Blob Storage chapter) and Redshift storage.

Quiz

Question 1

What would be considered a good approach for communication between different manager and worker processes within the service hosts?

Show Answer

Requirements of Google Maps' Design

Understand the requirements to design a maps application like Google Maps.

We'll cover the following

Requirements

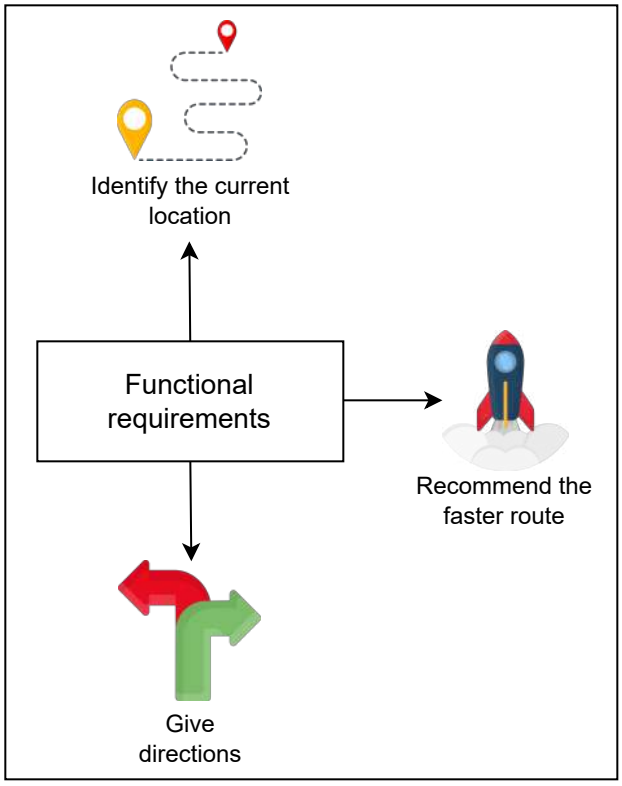
Before we start requirements, let’s clarify that we will design a system like Google Maps by picking a few key features because actual Google Maps is feature-rich and complex.

Let’s list the functional and non-functional requirements of the system under design.

Functional requirements

The functional requirements of our system are as follows.

- **Identify the current location**: Users should be able to approximate their current location (latitude and longitude in decimal values) on the world map.
- **Recommend the fastest route**: Given the source and destination (place names in text), the system should recommend the optimal route by distance and time, depending on the type of transportation.
- **Give directions**: Once the user has chosen the route, the system should list directions in text format, where each item in the list guides the user to turn or continue in a specific direction to reach the destination.



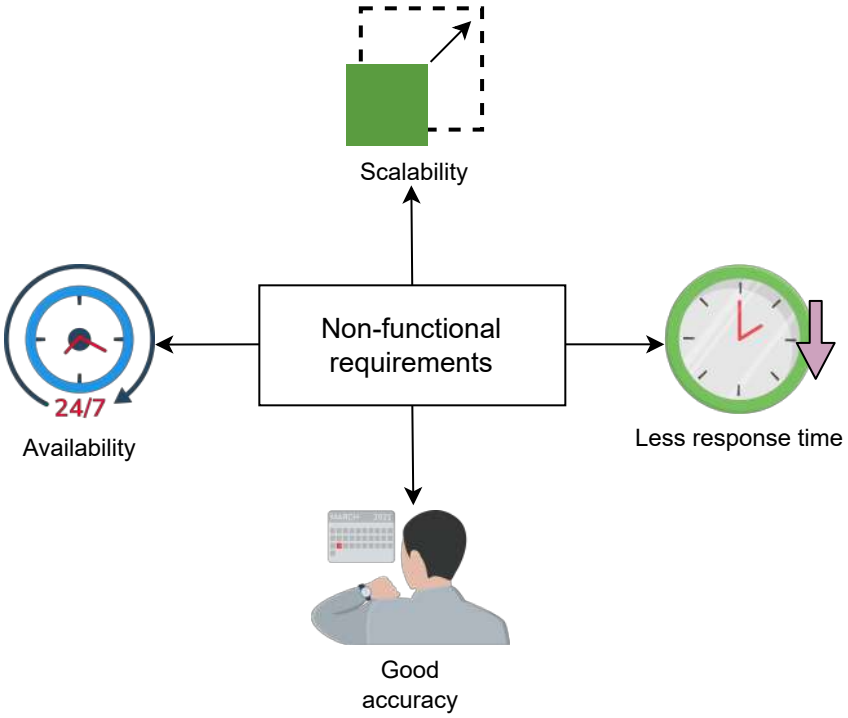
Google Maps functional requirements

Non-functional requirements

The non-functional requirements of our system are as follows.

- **Availability**: The system should be highly available.
- **Scalability**: It should be scalable because both individuals and other enterprise applications like Uber and Lyft use Google Maps to find appropriate routes.

- **Less response time**: It shouldn't take more than two or three seconds to calculate the ETA and the route, given the source and the destination points.
- **Accuracy**: The ETA we predict should not deviate too much from the actual travel time.



Non-functional requirements of Google maps

Note: We're not getting into the details of how we get the data on roads and layout. Government agencies provide maps, and in some places, Google itself drives mapping vehicles to find roads and their intersection points. Road networks are modeled with a graph data structure, where intersection points are the **vertices**, and the roads between intersections are the **weighted edges**.

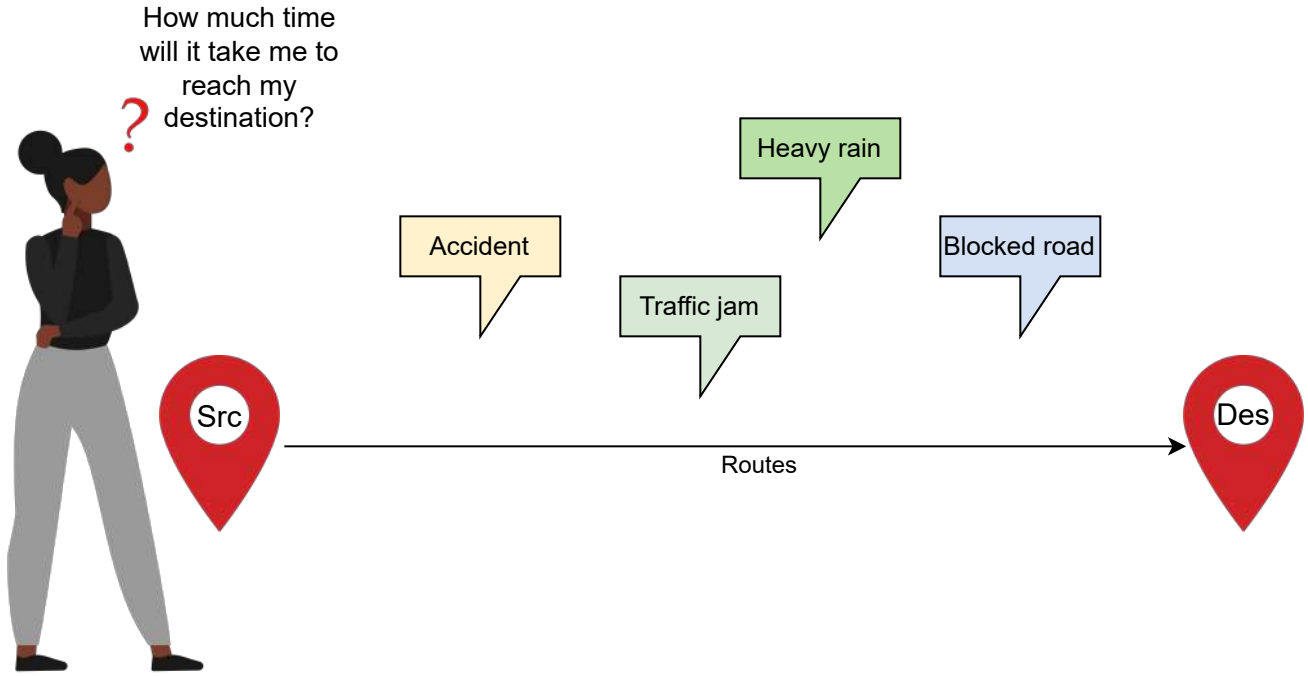
Challenges

Some of the challenges that we need to focus on while designing a system like Google Maps are below:

- **Scalability**: Serving millions of queries for different routes in a second, given a graph with billions of nodes and edges spanning over 194 countries, requires robust scalability measures. A simple approach, given the latitude and longitude of the source and destination, would be to apply an algorithm like Dijkstra to find the shortest path between the source and the destination. However, this approach wouldn't scale well for billions of users sending millions of queries per second. This is because running any path-finding algorithm on a graph with billions of nodes running a million times per second is inefficient in terms of time and cost, ultimately leading to a bad user experience. Therefore, our solution needs to find alternative techniques to scale well.

A graph spanning the whole world network

- **ETA computation**: In an ideal situation with empty roads, it's straightforward to compute ETA using the distance and the speed of the vehicle we want to ride on. However, we cannot ignore factors like the amount of traffic on the roads and road conditions, which affect the ETA directly. For example, a road under construction, collisions, and rush hours all might slow down traffic. Quantifying the factors above to design our system is not trivial. Therefore, we'll, categorize the factors above in terms of traffic load to complete our design.



Factors affecting the ETA computation

Resource estimation

Let’s estimate the total number of servers, storage, and bandwidth required by the system.

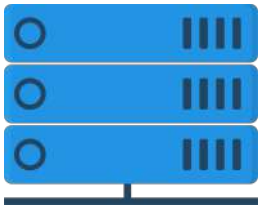
Number of servers estimation

To estimate the number of servers, we need to know how many daily active users are using Google Maps and how many requests per second a single Google Maps server can handle. We assume the following numbers:

- Daily active users who use Google Maps: 32 million (about 1 billion monthly users).
- Number of requests a single server can handle per second: 8,000.

The number of servers required has been calculated using the below formula:

$$\frac{Number\ of\ active\ users}{requests\ handled\ per\ server} = 4K\ servers$$



Number of servers required for Google Maps

Storage estimation

Google Maps is essentially a system with a one-time storage requirement. The road data from many countries has already been added, which is over 20 petabytes as of 2022. Since there are minimal changes on the road networks, the daily storage requirement is going to be negligible for Google Maps. Also, short-term changes in the road network is a small amount of data as compared to the full network data. Therefore, our storage needs don’t change rapidly.

Bandwidth estimation

As a standard practice, we have to estimate the bandwidth required for the incoming and outgoing traffic of our system. Most of the bandwidth requirements for Google Maps are due to requests sent by users. Therefore, we’ve devised the following formula to calculate bandwidth:

$$Total_{bandwidth} = Total_{requests\_second} \times Total_{query\_size}$$

The $Total_{requests\_second}$ represents the number of requests per second, whereas the $Total_{query\_size}$ represents the size of each request.

Incoming traffic

To estimate the incoming query traffic bandwidth, we assume the following numbers:

- Maximum number of requests by a single user per day: 50.

- Request size (source and destination): 200 Bytes.

Using the assumptions above, we can estimate the total number of requests per second on Google Maps using the following formula:

$$Total_{requests\_second} = \frac{Daily\ active\ users \times Requests_{per\_user}}{24 \times 60 \times 60} = 18,518\ requests\ per\ second.$$

We can calculate the incoming query traffic bandwidth required by Google Maps by inserting the request per second calculated above and the size of each request in the aforementioned bandwidth formula.

No. of requests per second	Request size (Bytes)	Bandwidth (Mb/s)
18518	200	29.63

Outgoing traffic

Outgoing application traffic will include the response that the server generates for the user when they make a navigation request. The response consists of visuals and textual content, and typically includes the route shown on the map, estimated time, distance, and more detail about each step in the route. We assume the following numbers to estimate the outgoing traffic bandwidth:

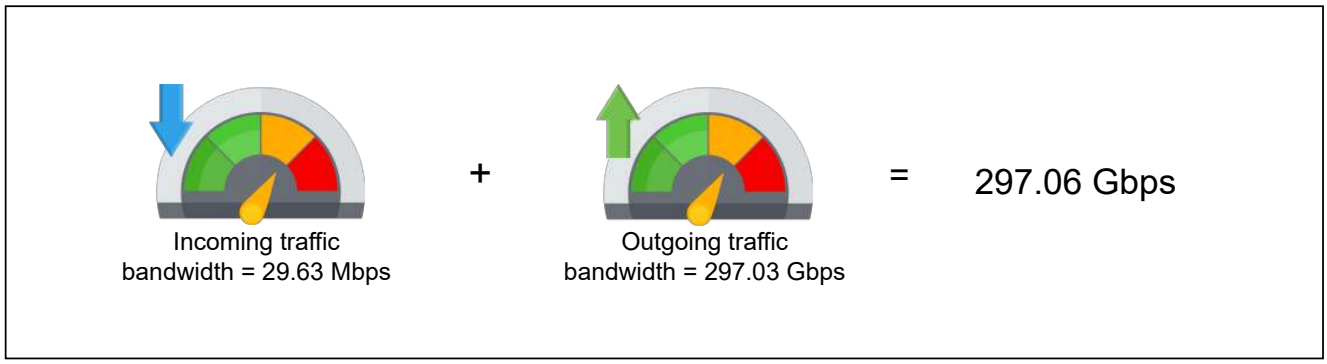
- Total requests per second (calculated above): 18,518.
- Response size: 2 MB + 5 KB = 2005 KB.
  - Size of visual data on average: 2 MB.
  - Size of textual data on average: 5 KB.

We can calculate the bandwidth required for outgoing traffic using the same formula.

Bandwidth Required for the Outgoing Application Traffic

No. of requests per second	Response size (KB)	Bandwidth (Gb/s)
18518	2005	f 297.03

Show Detailed Calculations



Summarizing the bandwidth requirements for Google Maps

Building blocks we will use

Now that we've completed our estimates of resources required, let's identify the building blocks that will be an integral part of our design for the Google Maps system. Below, we have the key building blocks:

Building blocks used in the high-level and detailed design

- **Load balancers** are necessary to distribute user requests among different servers and services.
- **Databases** are required to store data in the form of a graph along with metadata information.
- **Distributed search** is needed to search different places on the map.
- **A pub-sub system** is required for generating and responding to important events during navigation and notifying the corresponding services.
- **A key-value store** is also used to store some metadata information.

Besides the building blocks mentioned above, other components will also be required for designing our maps system. These components will be discussed in the design lessons. We are now ready to explore the system and API design of Google Maps.

Design of Google Maps

Let's build a high-level design of a maps service.

We'll cover the following

High-level design

Let’s start with the high-level design of a map system. We split the discussion into two sections:

1. The components we’ll need in our design.
2. The workflow that interconnects these components.

Components

We’ll use the following components in our design:

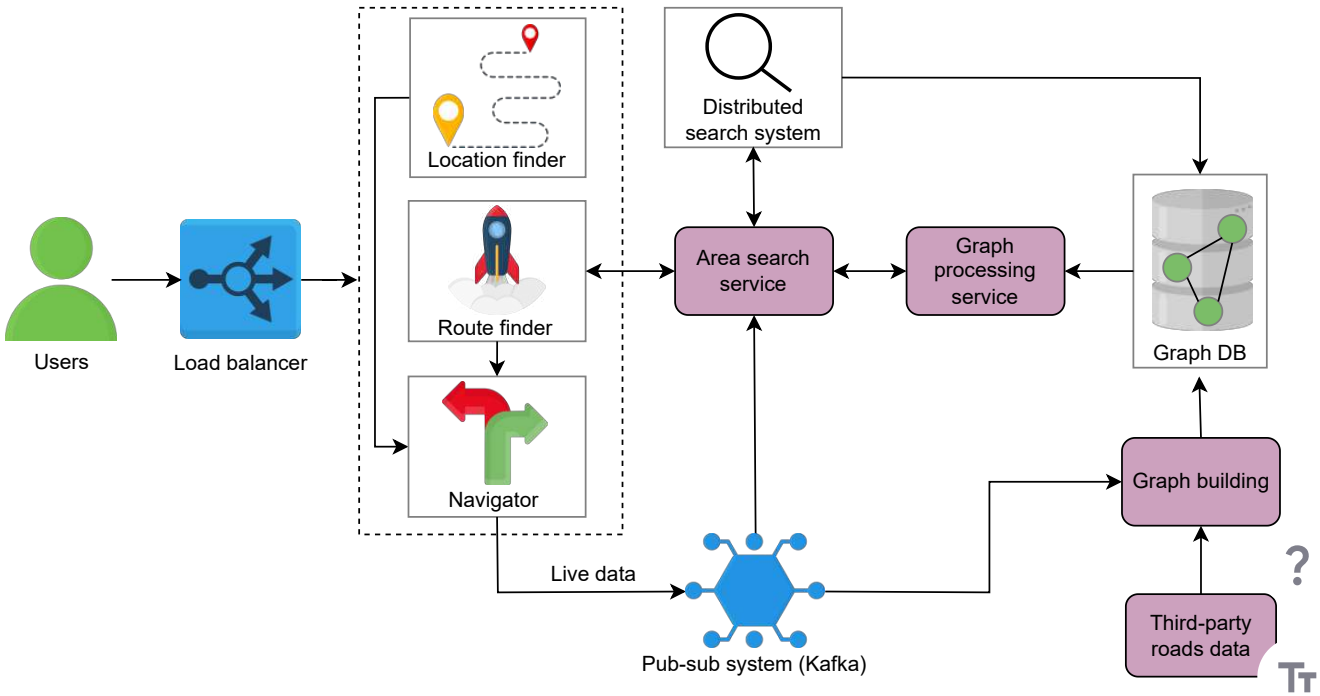
- **Location finder**: The location finder is a service used to find the user’s current location and show it on the map since we can’t possibly personally remember the latitude and longitude for every place in the world.
- **Route finder**: For the people who are new to a place, it’s difficult to travel because they don’t know the correct routes. The route finder is service used to find the paths between two locations, or points.

- **Navigator**: Suggesting a route through the route finder is not enough. A user may deviate from the optimal path. In that case, a navigator service is used. This service keeps track of users' journeys and sends updated directions and notifications to the users as soon as they deviate from the suggested route.
- **GPS/Wi-Fi/Cellular technology**: These are the technologies that we used to find the user's ground position.
- **Distributed search**: For converting place names to latitude/longitude values, we need a conversion system behind the source and destination fields. A distributed search maintains an index consisting of places names and their mapping to latitude and longitude. User's entered keywords are searched using this service to find a location on the map.
- **Area search service**: The area search service coordinates between the distributed search and graph processing service to obtain the shortest path against a user query. The area search service will request the distributed search to obtain the locations of the source and destination on the map. Then, it will use the graph processing service to find the optimal path from the source to the destination.
- **Graph processing service**: There can be multiple paths from one place to another. The graph processing service runs the shortest path algorithm on a shorter graph based on the area spanning the source and destination points and helps us determine which path to follow.
- **Database**: As discussed in the previous lesson, we have the road data from various sources stored in the form of a graph. We'll map this data to a database to develop the road network graph. We're using a graph database like DataStax Graph to store the graph for our design.
- **Pub-sub system**: Users might deviate from the first suggested path. In that case, they'll need information on a new path to their destination. Pub-sub is a system that listens to various events from a service and triggers another service accordingly. For example, when a user deviates from the suggested path, it pings the area search service to find a new route from the user's current location to their destination point. It also

collects the streams of location data for different users from the navigator. This data can be processed later to find traffic patterns on different roads at different times. We'll use Kafka as a pub-sub system in our design.

- **Third-party road data**: How can we build a map system if we don't have the road networks data? We need to collect the road data from third-party resources and preprocess the collected data to bring it into a single format that can be utilized to build the graph.
- **Graph building**: We'll use a service that builds the graph from the given data, either collected from the third-party resources or from the users.
- **User**: This refers to a person or a program that uses the services of the map system.
- **Load balancer**: This is a system that is used to distribute user requests among different servers and services.

The illustration below shows how the above components are connected.



A high-level design of a map system

Workflow

We explain the workflow by assuming that a user has to travel between two points but doesn't know their current location or how to get to their destination. So, let's see how the maps system helps.

> For this exercise, we assume that the data about road networks and maps has already been collected from the third parties and the graph is built and stored in the Graph DB.

- In a maps system, the user has to enter their starting point and their destination to create a path between the two. For the starting source point, the user uses the current location service.
- The location finder determines the current location by maintaining a persistent connection with the user. The user will provide an updated location using GPS, Wi-Fi, and cellular technology. This will be the user's source point.
- For the destination point, the user types an address in text format. It's a good idea to make use of our Typeahead service here to provide useful suggestions and avoid spelling mistakes.
- After entering the source and the destination points, the user requests the optimal path.
- The user's path request is forwarded to the route-finder service.
    - The route finder forwards the requests to an area search service with the source and the destination points.
    - The area search service uses the distributed search to find the latitude/longitude for the source and the destination. It then calculates the area on the map spanning the two (source's and destination's) latitude/longitude points.
    - After finding the area, the area search service asks the graph processing service to process part of the graph, depending on the area to find the optimal path.

    - The graph processing service fetches the edges and nodes within that specified area from the database, finds the shortest path, and returns it to the route-finder service that visualizes the optimal path with the distance and time necessary to comeplete the route. It also displays the steps the user should follow for navigation.
- Now that the user can visualize the shortest path on the map, they also want to get directions towards the destination. The direction request is handled by the navigator.
- The navigator tracks that the user is following the correct path, which it has from the route-finder service. It updates the user's location on the map while the user is moving, and shows where to turn with the distance. If a user deviates from the path, it generates an event that is fed to Kafka.
- Upon receiving the event from the navigator, Kafka updates the subscribed topic of the area search service, which in turn recalculates the optimal path and suggests it to the user. The navigator also provides a stream of live location data to the graph, building it through the pub-sub system. Later, this data can be used to improve route suggestions

API design

Let's look at different APIs for the maps service.

Show the user's current location on the map

The `currLocation` function displays the user's location on the map.

```
currLocation(location)
```

Parameter	Description

`location`	This depicts whether the user location is on or off. This call will establi connection between the client and the server, where the client will pe the server about its current location. If the location is `false`, the servi to turn on their location.

Find the optimal route

The `findRoute` function helps find the optimal route between two points.

```
findRoute(source, destination, transport_type)
```

Parameter	Description
`source`	This is the place (in text format) where the users want to start their jou
`destination`	This is the place (in text format) where the users want to end their jou
`transport_type (optional)`	This can be a bicycle, car, airplane, and so on. The default transport ty the user doesn't provide this parameter.

Get directions

The `directions` function helps us get alerts in the form of texts or sounds that indicate when and where to turn next.

```
directions(curr_location, steps)
```

Parameter	Description
`current_location`	This is the latitude/longitude value of the user's current location on

`steps`	These are the steps the user should follow in order to reach their desti

We described the high-level design by explaining the services we will need. We also discussed API design. Next, we'll discuss how we met the scalability challenge through segments.

Back

Requirements of Google Maps' Design

Next

Challenges of Google Maps' Design

Completed

Challenges of Google Maps' Design

Let's understand and resolve the key challenges in designing a system like Google Maps.

We'll cover the following

Meeting the challenges

We listed two challenges in the Introduction lesson: scalability and ETA computation. Let’s see how we meet these challenges.

Scalability

Scalability is about the ability to efficiently process a huge road network graph. We have a graph with billions of vertices and edges, and the key challenges are inefficient loading, updating, and performing computations. For example, we have to traverse the whole graph to find the shortest path. This results in increased query time for the user. So, what could be the solution to this problem?

The idea is to break down a large graph into smaller subgraphs, or partitions. The subgraphs can be processed and queried in parallel. As a result, the graph construction and query processing time will be greatly

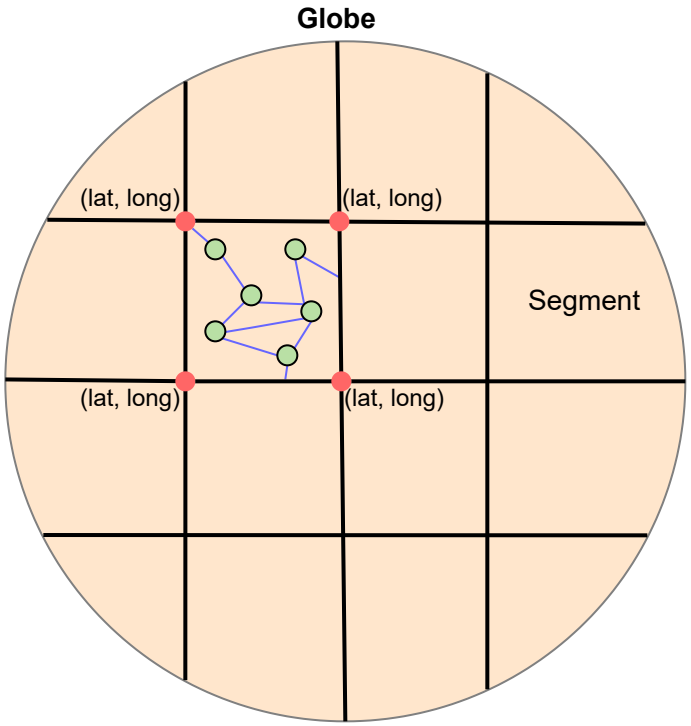
decreased. So, we divide the globe into small parts called segments. Each segment corresponds to a subgraph.

Segment

A **segment** is a small area on which we can work easily. Finding paths within these segments works because the segment's road network graph is small and can be loaded easily in the main memory, updated, and traversed. A city, for example, can be divided into hundreds of segments, each measuring $5 \times 5\ miles$.

> **Note**: A segment is not necessarily a square. It can also be a polygon. However, we are assuming square segments for ease of explanation.

Each segment has four coordinates that help determine which segment the user is in. Each coordinate consists of two values, the latitude and longitude.

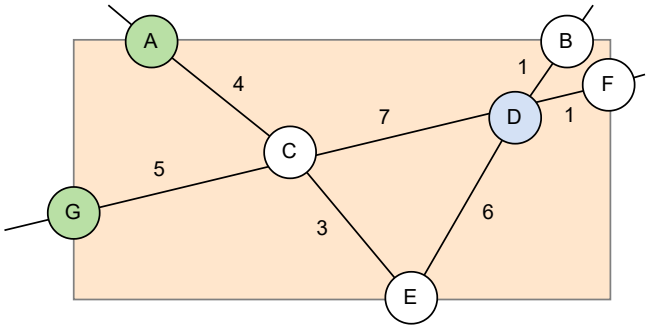


Partitioning the globe into small segments, and each segment has four coordinates with latitude and longitude values

Let's talk about finding paths between two locations within a segment. We have a graph representing the road network in that segment. Each intersection/junction acts as a vertex and each road acts as an edge. The graph is weighted, and there could be multiple weights on each edge—such as distance, time, and traffic—to find the optimal path. For a given source and destination, there can be multiple paths. We can use any of the graph algorithms on that segment's graph to find the shortest paths. The most common shortest path algorithm is the **Dijkstra's algorithm**.

More Shortest Path Algorithms

After running the shortest path algorithm on the segment's graph, we store the algorithm's output in a distributed storage to avoid recalculation and cache the most requested routes. The algorithm's output is the shortest distance in meters or miles between every two vertices in the graph, the time it takes to travel via the shortest path, and the list of vertices along every shortest path. All of the above processing (running the shortest path algorithm on the segment graph) is done offline (not on a user's critical path).



Cached Information: output of the shortest path algorithm

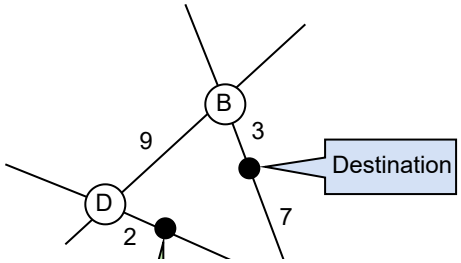
Source	Destination	Distance (m)	ETA (min)	Path
A	D	11	15	A->C->D
G	D	12	16.5	G->C->D
⋮	⋮	⋮	⋮	⋮

Finding a path between vertices within a segment

The illustration above shows how we can find the shortest distance in term of miles between two points. For example, the minimum distance (`m`)

between points A and D is 11 miles by taking the path A->C->D. It has an ETA of 15 minutes.

We've found the shortest path between two vertices. What if we have to find the path between two points that lie on the edges? What we do is find the vertices of the edge on which the points lie, calculate the distance of the point from the identified vertices, and choose the vertices that make the shorter total distance between the source and the destination. The distance from the source (and destination) to the nearest vertices is approximated using latitude/longitude values.



Possible paths	Distance
source->E->Destination	8 + 7 = 15
source->D->B->Destination	2 + 9 + 3 = 14

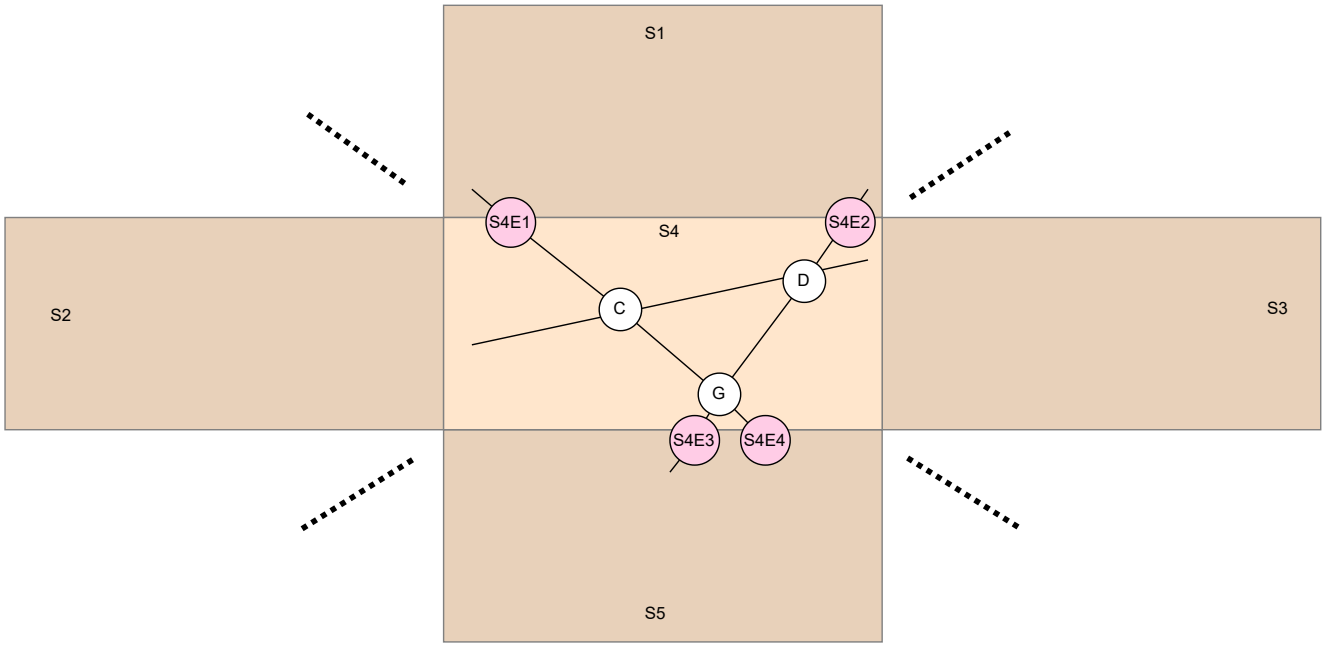
Choosing a path between points that lie on the edge

We're able to find the shortest paths within the segment. Let's see what happens when the source and the destination belong to two different segments, and how we connect the segments.

Connect two segments

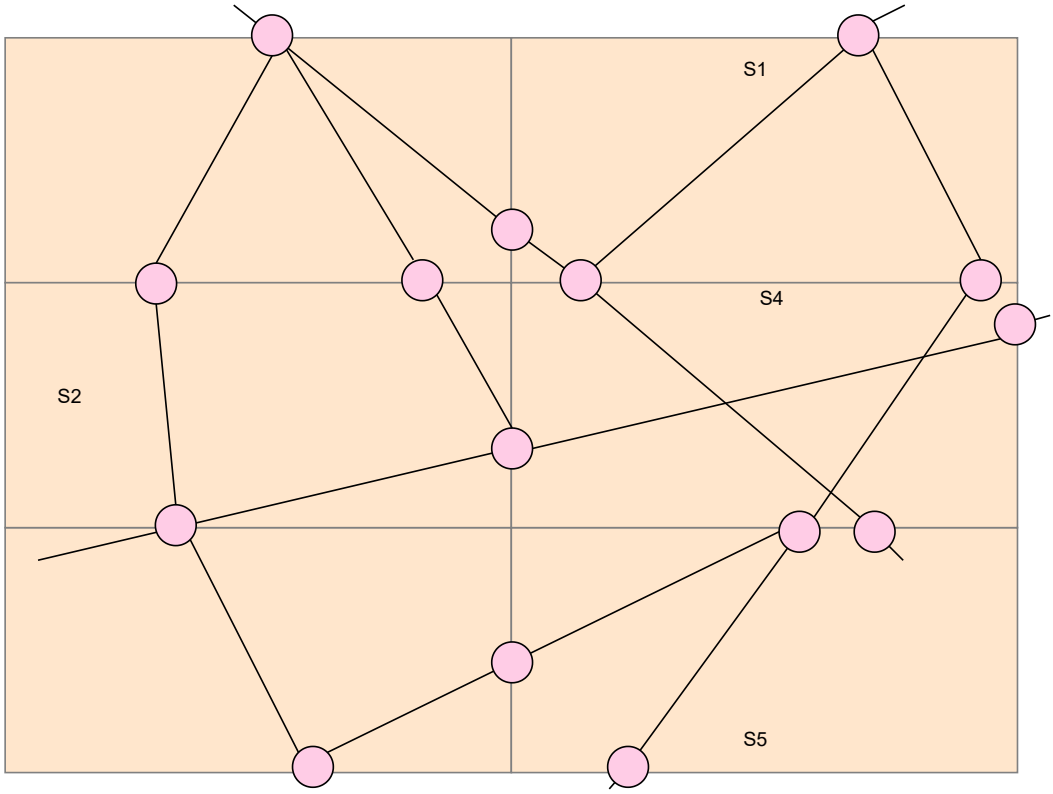
Each segment has a unique name and boundary coordinates. We can easily identify which location (latitude, longitude) lies in which segment. Given the source and the destination, we can find the segments in which they lie. For each segment, there are some boundary edges, which we call **exit points**. In the illustration below, we have four exit points, S4E1, S4E2, S4E3, and S4E4

> **Note**: Besides the vertices we have inside the segment, we also consider the exit points of a segment as vertices and calculate the shortest path for these exit points. So for each segment, we calculate the shortest path between exit points in addition to the shortest path from the exit points to vertices inside the segment. Each vertex's shortest path information from the segment's exit points is cached.



Connecting the segments by considering exit points of a segment as vertices

An exit point connects neighboring segments and is shared between them. In the illustration above, each exit point is connecting two segments and each is shared between two segments. For example, the exit point S4E1 is also an exit point for S1. Having all the exit points for each segment, we can connect the segments and find the shortest distance between two points in different segments. While connecting the segments, we don't care about the inside segment graph. We just need exit points and the cached information about the exit points. We can visualize it as a graph made up of exiting vertices, as shown in the following illustration.



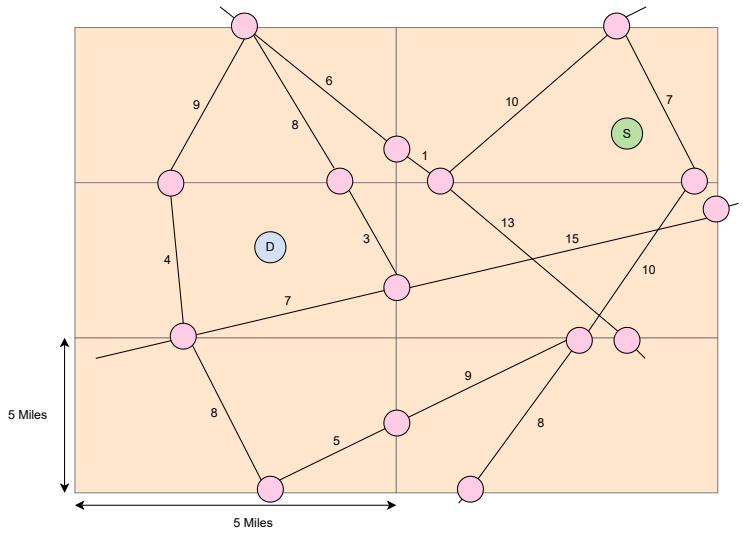
A graph made up of exit points, where the lines connecting the exit points are not actually straight. There could be many vertices in between the two exit points

Since we can't run the shortest path algorithm for all the segments throughout the globe, it's critical to figure out how many segments we need to consider for our algorithm while traveling inter-segment. The aerial distance between the two places is used to limit the number of segments. With the source and destination points, we can find the aerial distance between them using the haversine formula.

Suppose the aerial distance between the source and the destination is 10 kilometers. In that case, we can include segments that are at a distance of 10 kilometers from the source and destination in each direction. This is a significant improvement over the large graph.

Once the number of segments is limited, we can constrain our graph so tha the vertices of the graph are the exit points of each segment, and the calculated paths betweenthe exit points are the graph's edges. All we have

do now is run the shortest path algorithm on this graph to find the route. The following illustration shows how we find the path between two points that lie in different segments.



Source S and Destination D lie in two different segments, and we have to find the shortest route between them

1 of 10

Let’s summarize how we met the challenge of scalability. We divided our problem so that instead of working on a large road network as a whole, we worked on parts (segments) of it. The queries for a specific part of the road network are processed on that part only, and for the queries that require processing more than one part of the network, we connect those parts, as v have shown above.

ETA computation

For computing the ETA with reasonable accuracy, we collect the live location data (`(userID, timestamp,(latitude, longitude))`) from the navigation service through a pub-sub system. With location data streams, we can calculate and predict traffic patterns on different roads. Some of the things that we can calculate are:

- Traffic (high/medium/low) on different routes or roads.
- The average speed of a vehicle on different roads.
- The time intervals during which a similar traffic pattern repeats itself on a route or road. For example, highway X will have high traffic between 8 to 10 AM.

The information above helps us provide a more accurate ETA. For example, if we know that the traffic will be high at a specific time on a particular road, the ETA should also be greater than usual.

In this lesson, we looked at how we meet the scalability challenge through segments and ETA computation using live data. In the next lesson, we'll discuss the design in more detail.

← Back

Design of Google Maps

Next →

Detailed Design of Google Maps

Detailed Design of Google Maps

Let's look into the detailed design of the maps system.

We'll cover the following

In this lesson, we'll discuss our detailed design by answering the following questions:

1. How do user requests benefit from segments?
2. How do we improve the user experience by increasing the accuracy of ETAs?

Segment setup and request handling

This section will describe how the segment data is stored in the database and how user requests are handled using the already stored data.

Starting with the storage schema, we discuss how the segments are added and hosted on the servers and also how the user requests are processed.

Storage schema

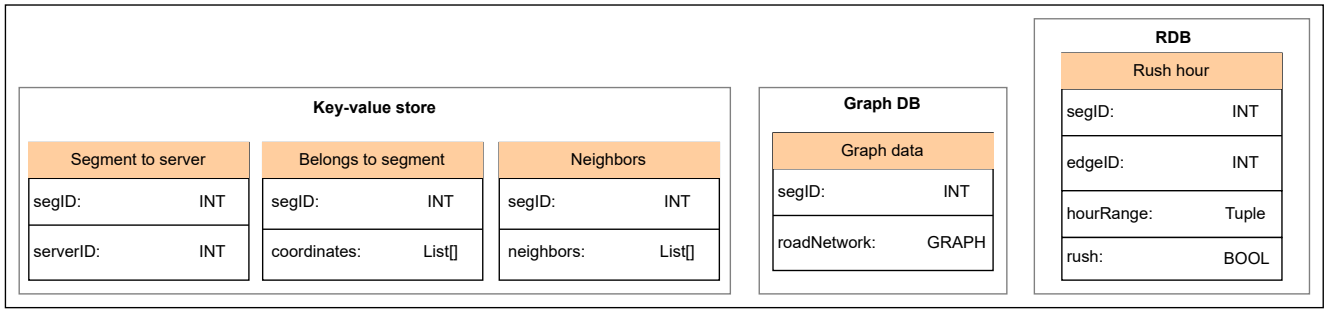
We store the following information for each segment:

Key-value store:

- The segment's `ID`.
- The `serverID` on which the segment is hosted.
- In reality, each segment is a polygon, so we store boundary `coordinates` (latitude/longitude), possibly as a list.
- A list of segment IDs of the `neighbors` segments.

Graph database

- The road network inside the segment in the form of a `graph`.



Storage schema

Relational DB

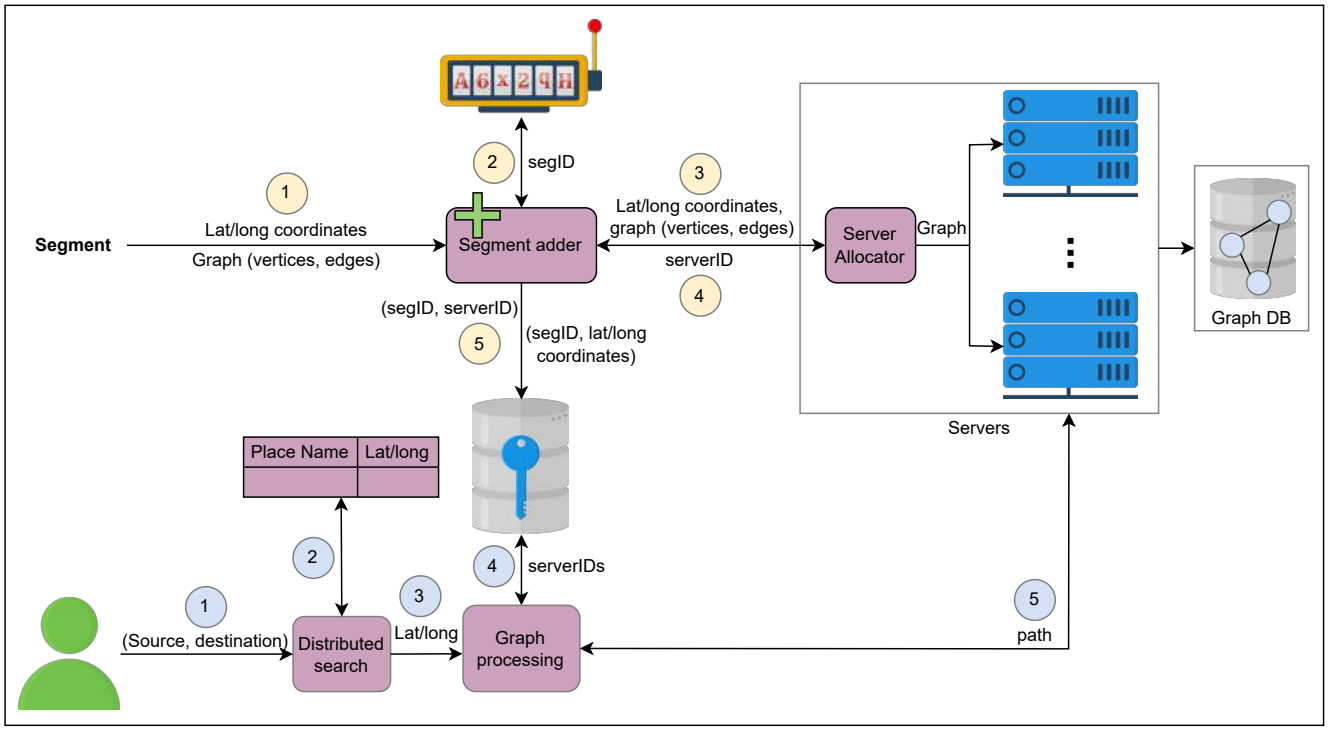
We store the information to determine whether, at a particular hour of the day, the roads are congested. This later helps us decide whether or not to update the graph (weights) based on the live data.

- `edgeID` identifies the edge.
- `hourRange` tells us which hour of the day it is when there are typical road conditions (non-rush hour) on the road.
- `rush` is a Boolean value that depicts whether there is congestion or not on a specific road at a specific time.

> **Note:** `segID`, `serverID`, and `edgeID` are unique IDs generated by a uniqueID generator (see the Sequencer lessons for details).

Design

The following illustration consists of two workflows. One adds segments to the map and hosts them on the severs, while the other shows how the user request to find a path between two points is processed.



Workflow 1) Adding segments (in yellow), and workflow 2) Processing user requests (in blue)

Add segment

1. Each segment has its latitude/longitude boundary coordinates and the graph of its road network.
2. The segment adder processes the request to add the segment along with the segment information. The segment adder assigns a unique ID to each segment using a unique ID generator system.
3. After assigning the ID to the segment, the segment adder forwards the segment information to the server allocator.
4. The server allocator assigns a server to the segment, hosts that segment graph on that server, and returns the `serverID` to the segment adder.

5. After the segment is assigned to the server, the segment adder stores the segment to server mapping in the key-value store. It helps in finding the appropriate servers to process user requests. It also stores each segment's boundary latitude/longitude coordinates in a separate key-value object.

Handle the user's request

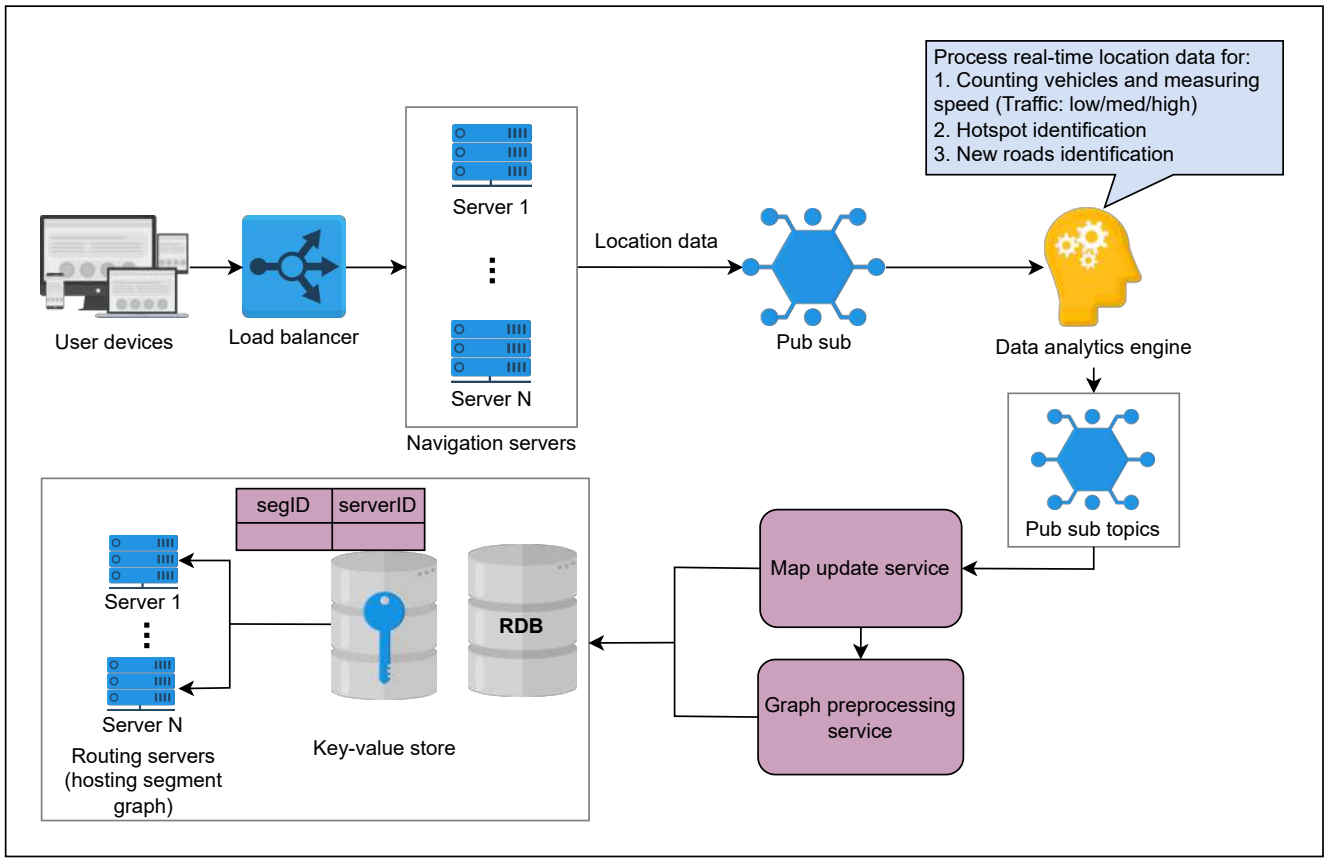
1. The user provides the source and the destination so that our service can find the path between them.
2. The latitude and longitude of the source and the destination are determined through a distributed search.
3. The latitude/longitude for the source and the destination are passed to the graph processing service that finds the segments in which the source and the destination latitude/longitude lie.
4. After finding the segment IDs, the graph processing service finds the
5. The graph processing service connects to the relevant servers to find the shortest path between the source and the destination. If the source and the destination belong to the same segment, the graph processing service returns the shortest path by running the query only on a single segment. Otherwise, it will connect the segments from different servers, as we have seen in the previous lesson.

Improve estimations using live data

This section describes how we can improve the ETA estimation accuracy using live data. If we have a sequence of location data for different devices, we can find movement patterns and perform analytics to predict various factors that may influence a user's trip. Google Maps uses a combination of GPS, Wi-Fi, and cell towers to track users' locations. To collect this data, our maps system servers need to have a persistent connection with all the devices that have their location turned on. Below, we discuss the tools,

techniques, and components involved in the process of improving estimations using live data.

- **WebSocket** is a communication protocol that allows users and servers to have a two-way, interactive communication session. This helps in the real-time transfer of data between user and server.
- The **load balancer** balances the connection load between different servers since there is a limit on the number of WebSocket connections per server. It connects some devices to server 1, some to server 2, and so on.



Performing analytics on the live location data to keep the map up-to-date and also improve ETA estimations

- A **pub-sub** system collects the location data streams (device, time, location) from all servers. The location data from pub-sub is read by a **data analytics engine** like Apache Spark. The data analytics engine uses data science techniques—such as machine learning, clustering, and

so on—to measure and predict traffic on the roads, identify gatherings, hotspots, events, find out new roads, and so on. These analytics help our system improve ETAs.

> **Note:** The amount of traffic, road conditions, and hotspots directly affect average travel speed, which ultimately affects users' ETAs.

- The data analytics engine publishes the analytics data to a new **pub-sub topic**.
- The **map update service** listens to the updates from the pub-sub topic for the analytics. It updates the segment graphs if there is a new road identified or if there is a change in the weight (`average speed (traffic, road condition)`) on the edges of the graph. Depending on the location, we know which segment the update belongs to. We find the **routing server** on which that segment is placed from the key-value store and update the graph on that server.

Point to Ponder

Question

Many traffic conditions are transitory (like stopping at a signal), so updating the graph very often wouldn't scale well because it requires excessive processing. What could be the solution to this problem?

Show Answer

- The **graph preprocessing service** recalculates the new paths on the updated segment graph. We've seen how the paths are updated continuously in the background based on the live data.

We've learned how the segments work, how a user finds the location between two points, and how the ETA's accuracy is improved by utilizing the live location data.

Back

Challenges of Google Maps' Design

Next

Evaluation of Google Maps' Design

Mark as Completed

Requirements of Yelp's Design

Learn about the requirements for a proximity service like Yelp.

We'll cover the following

Requirements

Let's identify the requirements of our system.

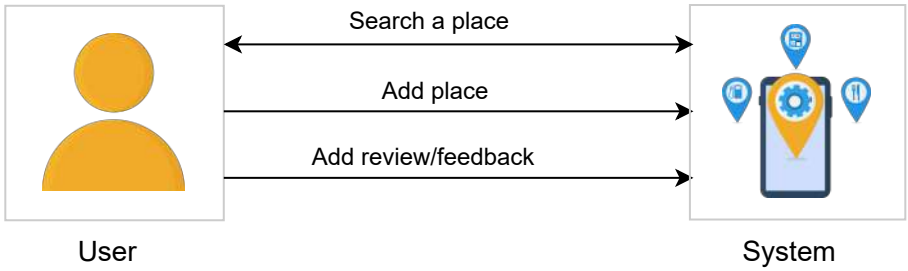
Functional requirements

The functional requirements of our systems are below:

- **User accounts**: Users will have accounts where they're able to perform different functionalities like log in, log out, add, delete, and update places' information.

> **Note:** There can be two types of users: business owners who can add their places on the platform and other users who can search, view, and give a rating to a place.

- **Search**: The users should be able to search for nearby places or places of interest based on their GPS location (longitude, latitude) and/or the name of a place.
- **Feedback**: The users should be able to add a review about a place. The review can consist of images, text, and a rating.



Functional requirements

The non-functional requirements of our systems are:

- **High availability**: The system should be highly available to the users.
- **Scalability**: The system should be able to scale up and down, depending on the number of requests. The number of requests can vary depending on the time and number of days. For example, there are usually more searches made at lunchtime than at midnight. Similarly, during tourist season, our system will receive more requests as compared to in other months of the year.
- **Consistency**: The system should be consistent for the users. All the users should have a consistent view of the data regarding places, reviews, and images.

- **Performance**: Upon searching, the system should respond with suggestions with minimal latency.

Resource estimation

Let's assume that we have:

- A total of 178 million unique users.
- 60 million daily active users.
- 500 million places.

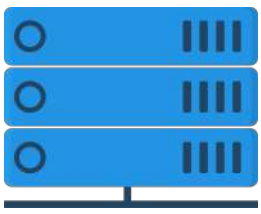
Number of servers estimation

We need to handle concurrent requests coming from 60 million daily active users. As we did in our discussion in the back-of-the-envelope lessons, we assume an RPS of 8,000.

$$\frac{Number\ of\ daily\ active\ users}{RPS\ of\ a\ server} = \frac{60\ million}{8000} = 7500\ servers$$

Estimating the Number of Servers

Number of Daily Active Users (in Millions)	60
RPS of a Server	8000
Number of Servers Required	f 7500



7500 servers

The number of servers required for Yelp

Storage estimation

Let's calculate the storage we need for our data. Let's make the following assumptions:

- We have a total of 500 million places.
- For each place, we need 1,296 Bytes of storage.
- We have one photo attached to each place, so we have 500 million photos.
- For each photo, we need 280 Bytes of storage. Here, we consider the row size of the photo entity in the table, which contains a link to the actual photo in the blob store.
- At least 1 million reviews of different places are added daily.
- For each review, we need 537 Bytes of storage.
- We have a total of 178 million users.
- For each user, we need 264 Bytes of storage.

> **Note:** The Bytes used for each place, photo, review, and user are based on the database schema that we'll discuss in the next lesson.

The following calculater computes the total storage we need:

Estimating Storage Requirements

Type of information	Size Required by an Entity (in Bytes)	Count (in Millions)	Total Size (in GBs)
Place	1296	500	648

Photo	280	500	f 140
Review	537	1	f 0.54
User	264	178	f 46.99
Total Storage Required			f 835.53

The total amount of storage required by Yelp

Bandwidth estimation

To estimate the bandwidth requirements for Yelp, we categorize the bandwidth calculation of incoming and outgoing traffic.

For incoming traffic, let's assume the following:

- On average, five places are added every day.
- For each place, we take up 1,296 Bytes.
- A photo of size 3 MB is also attached with each place. This is the size of the photo that we save in the blob store.
- One million reviews of different places are added every day.
- Each review, takes up 537 Bytes.

We divide the total size of information per day by 86,400 to convert it into per second bandwidth.

Estimating Incoming Bandwidth Requirements

Average Number of Places Added Daily	5
Storage Needed for Each Place (Bytes)	1296
Size of Photo (in MBs)	3
Total Size of Place Information (Bytes)	f 15006480
Average Number of Reviews Added Daily (in Millions)	1
Storage Needed for Each Review (Bytes)	537
Total Size of Reviews (Bytes)	f 537000000
Total Incoming Bandwidth (KBps)	f 6.39
Total Incoming Bandwidth (Kbps)	f 51.12

For outgoing traffic, let's assume the following:

- A single search returns 20 places on average.
- Each place has a single photo attached to it that has an average size of 3 MB.
- Every returned entry contains the place and photo information.

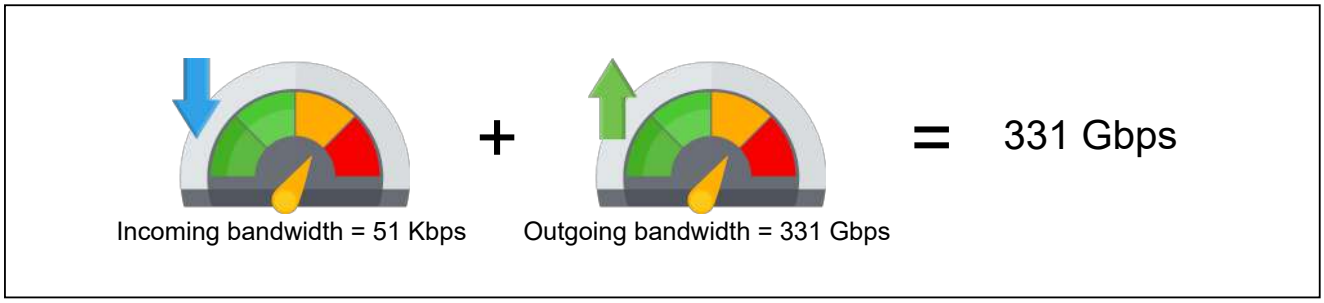
Considering that there are 60 million active daily users, we come to the following estimations:

Estimating Outgoing Bandwidth Requirements

Average Number of Places Returned on Each Search Request	20
Size of Place (in Bytes)	1296
Size of Photo (in MB)	3
Total Size of Place Information (Bytes)	f 60025920
Outgoing Bandwidth Required for a Single Request (Kbps)	f 0.69
Outgoing Bandwidth Required for a Single Request (KBps)	f 5.52
Daily Active Users (in Millions)	60
Total Outgoing Bandwidth Required (Kbps)	f 331200000
Total Outgoing Bandwidth Required (Gbps)	f 331.2

We need a total of approximately 51 $Kbps$ of incoming traffic and approximately 331 $Gbps$ of outgoing, assuming that the uploaded content is not compressed.

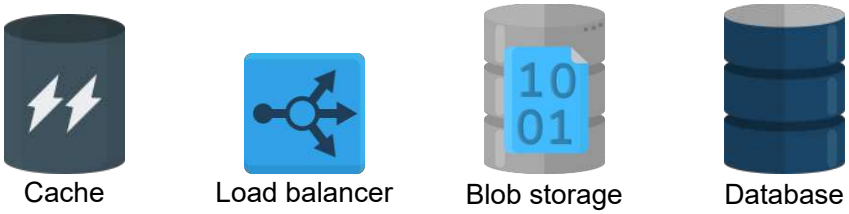
Total bandwidth requirements = 51 $Kbps$ + 331 $Gbps$ ≈ 331 $Gbps$.



The total bandwidth required by Yelp

Building blocks we will use

The design process of Yelp utilizes many building blocks that have already been discussed in the initial chapters of the course. We'll consider the following concepts while designing Yelp:



Building blocks in the high-level design of Yelp

- **Caching**: We'll use the cache to store information about popular places.
- **Load balancer**: We'll use the load balancer to manage the large amount of requests.
- **Blob storage**: We'll store images in the blob storage.
- **Database**: We'll store information about places and users in the database.

We'll also rely on **Google Maps** to understand the feature of searching for places within a particular radius.

Back
System Design: Yelp
Next
Design of

Completed

Design of Yelp

Learn to fulfill the design requirements of the Yelp system.

We'll cover the following

We identified the requirements and calculated the estimations for our Yelp system in the previous lesson. In this lesson, we discuss the API design, go through the storage schema, and then dive into the details of the system's building blocks and additional components.

API design

Let's discuss the API design for Yelp.

Search

We need to implement the `search` function. The API call for searching based on categories like "cafes" will be:

```
search(category, user_location, radius)
```

Parameter	Description
`category`	This is the type of search the user makes—for example, restaurants, cinemas, cafes, and so on.
`user_location`	This contains the location of the user who's searching w
`radius`	This is the specified radius where the user is trying to fi category.

This process returns a JSON object that contains a list of all the possible items in the specified category that also fall within the specified radius. Each entry has a place name, address, category, rating, and thumbnail.

The API call for searching based on the name of a place like "Burger Hut" will be:

```
search(name_of_place, user_location, radius)
```

Parameter	Description
`name_of_place`	This contains the name of the place that the user wants

This process returns a JSON object that contains information of the specifie place.

Add a place

The API call for adding a place is below:

```
add_place(name_of_place, description_of_place, category, latitude, longitude, phot
o}
```

Parameter	Description
`name_of_place`	This contains the name of the place, for example, "Burge
`description_of_place`	This contains a description of the place. For example, "B the yummiest burgers".
`category`	This specifies the category of the place—for example, "c
`latitude`	This tells us the latitude of the place.
`longitude`	This tells us the longitude of the place.
`photo`	This contains photos of the place. There can be a single photos.

This process returns a response saying that a place has been added, or an appropriate error if it fails to add a place.

Add a review

The API call for adding a place is below:

```
add_review(place_ID, user_ID, review_description, rating)
```

Parameter	Description
`place_ID`	This contains the ID of the place whose review is adc
`user_ID`	This contains the ID of the user who adds the review.

`review_description`	This contains the review of the place—for example, "the ambiance were superb".
`rating`	This contains the rating of the place—for example, 4 out

This process returns a response that a review has been added, or an appropriate error if it fails to add a review.

Storage schema

Let's define the storage schema for our system. A few of the tables we might need are "Place," "Photos," "Reviews," and "Users."

Let's define the columns of the "Place" table:

- **Place_ID**: We use the sequencer to generate an 8 Bytes (64 bits) unique ID for a place.

> **Note:** We generate IDs using the unique ID generator.

- **Name_of_Place**: This is a string that contains the name of the place. We use 256 Bytes for it.
- **Description_of_Place**: This holds a description of the place. We use 1,000 Bytes for it.
- **Category**: This specifies the type of place like restaurants, cinemas, bookshops, and so on (8 Bytes).
- **Latitude**: This stores the latitude of the location (8 Bytes).
- **Longitude**: This stores the longitude of the location (8 Bytes).
- **Photos**: This contains the foreign key (8 Bytes) of another table "Photos," which contains all the photos related to a particular place.

- **Rating**: This stores the rating of the place. It shows how many stars a place gets out of five. The rating is calculated based on the reviews it gets from the users.

The columns mentioned above are the most important ones in the table. We can add more columns like “menu,” “address,” “opening and closing hours,” and so on. Therefore, keeping in mind the essential columns, the size of one row of our table will be:

```
Size = 8 + 256 + 1000 + 8 + 8 + 8 + 8 = 1296 bytes
```

Now let’s define the “Photos” table:

- **Photo_ID**: We use the sequencer to generate a unique ID for a photo (8 Bytes or 64 bits).
- **Place_ID**: We use the foreign key (8 Bytes) from the “Place” table to identify which photo belongs to which place.
- **Photo_path**: We store the photos in blob storage and save the photo’s path (256 Bytes) in this column.

```
Size = 8 + 8 + 8 + 256 = 280 bytes
```

We need another table called “Reviews” to store the reviews, ratings, and photos of a place.

- **Review_ID**: We use the sequencer to generate a unique ID of 8 Bytes (64 bits) for a review.
- **Place_ID**: The foreign key (8 Bytes) from the “Place” table to determin which place the rating belongs to.
- **User_ID**: The foreign key (8 Bytes) from the “Users” table to identify which review belongs to which user.

- **Review_description**: This holds a description of the review. We use 512 Bytes for it.
- **Rating**: This stores how many stars a place gets out of five (1 Byte).

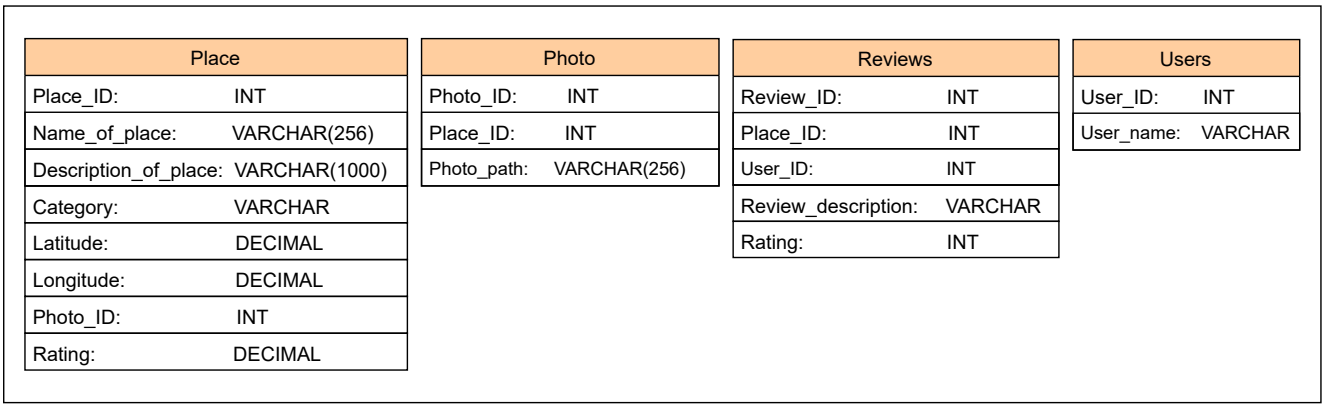
```
Size = 8 + 8 + 8 + 512 + 1 = 537 bytes
```

We use the “Users” table to store user information.

- **User_ID**: We use the sequencer to generate a unique ID for a user (8 Bytes).
- **User_name**: This is a string that contains the user’s name. We use 256 Bytes for it.

```
Size = 8 + 256 = 264 bytes
```

> **Note:** The `INT` in the following schema contains an 8-Byte ID that we generate using the unique ID generator.



Storage schema

Design

Now we’ll discuss the individual building blocks and components used in the design of Yelp and how they work together to complete various functional requirements.

Components

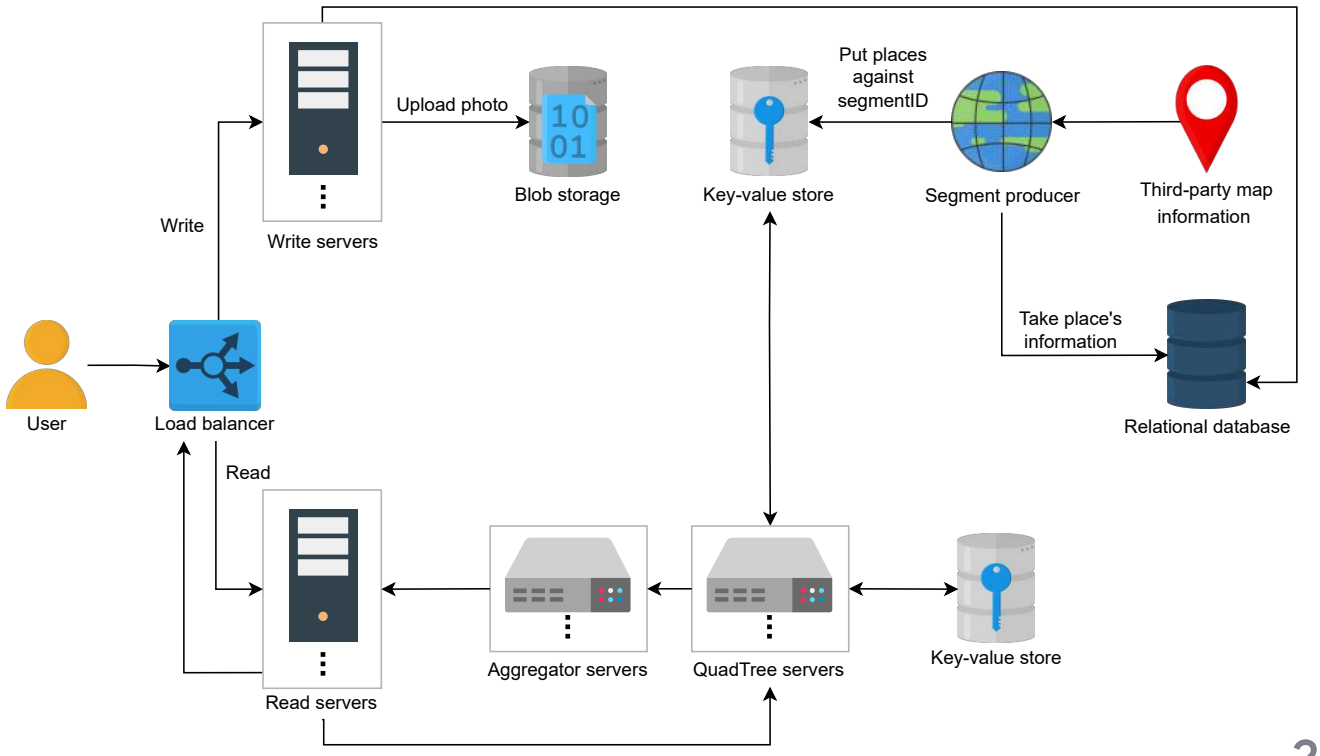
These are the components of our system:

- **Segments producer**: This component is responsible for communicating with the Geospatial database (like Google Maps). It takes up that data and divides the world into smaller regions called segments. The segment producer helps us narrow down the number of places to be searched.
- **QuadTree servers**: These are a set of servers that have trees that contain the places in the segments. A QuadTree server finds a list of places based on the given radius and the user's provided location and returns that list to the user. This component mainly aids the search functionality.
- **Aggregators**: The QuadTrees accumulate all the places and send them to the aggregators. Then, the aggregators aggregate the results and return the search result to the user.
- **Read servers**: We use a set of read servers that we use to handle all the read requests. Since we have more read requests, it's efficient to separate these requests from the write requests. Each read server directs the search requests to the QuadTrees' servers and returns the results to the user.
- **Write server**: We use a set of write servers to handle all the write requests. Each write server handles the write requests of the user and updates the storage accordingly. Examples for write requests include adding a place, writing a comment, rating a place, and so on.
- **Storage**: We'll use two types of storage to fulfill our diverse needs.
    - **SQL database**: Our system will have different tables like "Users," "Place," "Reviews," "Photos," and others as described below. The data in these tables is inherently relational and structured. We need to perform queries like places a user visited, reviews they

added, or view all the reviews of a specific place. It's easy to perform such queries in a SQL-based database. We also want all users to have a consistent view of the data, and SQL-based databases are better suited for such use cases. We'll use reliable and scalable databases, as is discussed in the Database building block.

- **Key-value stores**: We'll need to fetch the places in a segment efficiently. For that, we store the list of places against a segment ID in a key-value store to minimize searching time. We also save the QuadTree information in the key-value store, by storing the QuadTree data against a unique ID.

- **Load balancer**: A load balancer distributes users' incoming requests to all the servers uniformly.



Yelp design

Workflow

The user puts in a search request. We find all the relevant places in the given radius, while considering the user's location (latitude, longitude).

We explain the detailed workflow of our system in terms of the required functionalities below:

Searching a place: The load balancers route read requests to the read servers upon receiving them. The read servers direct them to the QuadTree servers to find all the places that fall within the given radius. The QuadTree servers then send the results to the aggregators to refine them and send them to the user.

Adding a place or feedback: The load balancers route the write requests to the write servers upon receiving them. Depending on the provided content, meaning the place information or review, the write servers add an entry in the relational database and put all the related images in the blob storage.

Making segments: The segment's producer splits the world map taken from the third-party map service into smaller segments. The places inside each segment are stored in a key-value store. Even though this is a one-time job, this process is repeated periodically for newer segments and places. Since the probability of new places being added is low, we update our segments every month.

We've discussed the design of Yelp, its API design, and the relevant storage schema. In the next lesson, we'll talk about the design considerations.

Back

Requirements of Yelp's Design

Next

Design Considerations of

Mark as Completed

Requirements of Uber’s Design

Learn about the requirements to design an Uber service.

We'll cover the following

Requirements

Let’s start with the requirements for designing a system like Uber.

Functional requirements

The functional requirements of our system are as follows:

- **Update driver location**: The driver is a moving entity, so the driver’s location should be automatically updated at regular intervals.

- **Find nearby drivers**: The system should find and show the nearby available drivers to the rider.
- **Request a ride**: A rider should be able to request a ride, after which the nearest driver should be notified about the rider’s requests.
- **Manage payments:** At the start of the trip, the system must initiate the payment process and manage the payments.
- **Show driver estimated time of arrival (ETA):** The rider should be able to see the estimated time of arrival of the driver.
- **Confirm pickup:** Drivers should be able to confirm that they have picked up the rider.
- **Show trip updates**: Once a driver and a rider accept a ride, they should be able to constantly see trip updates like ETA and current location until the trip finishes.
- **End the trip:** The driver marks the journey complete upon reaching the destination, and they then become available for the next ride.

Point to Ponder

Question

What if two drivers are at the same distance from the rider? How will we select the driver to whom we’ll send the request?

Show Answer

Non-functional requirements

The non-functional requirements of our system are as follows:

- **Availability:** The system should be highly available. The downtime of even a fraction of a second can result in a trip failure, in the driver being unable to locate the rider, or in the rider being unable to contact the driver.
- **Scalability:** The system should be scalable to handle an ever-increasing number of drivers and riders with time.
- **Reliability:** The system should provide fast and error-free services. Ride requests and location updates should happen smoothly.
- **Consistency:** The system must be strongly consistent. The drivers and riders in an area should have a consistent view of the system.
- **Fraud detection:** The system should have the ability to detect any fraudulent activity related to payment.

Resource estimation

Now, let’s estimate the resources for our design. Let’s assume it has around 500 million riders and about five million drivers. We’ll assume the following numbers for our estimates:

- We have 20 million daily active riders and three million daily active drivers.
- We have 20 million daily trips.
- All active drivers send a notification of their current location every four seconds.

Storage estimation

Let’s estimate the storage required for our system:

Rider’s metadata

Let’s assume we need around 1,000 Bytes to store each rider’s information, including ID, name, email, and so on, when the rider registers in our application. To store the 500 million riders, we require 500 GB of storage:

$$500 \times 10^6 \times 1000 = 500\ GB$$

Additionally, if we have around 500,000 new riders registered daily, we’ll need a further 500 MB to store them.

Driver’s metadata

Let’s assume we need around 1,000 Bytes to store each driver’s information, including ID, name, email, vehicle type, and so on, when the driver registers in our application. To store the five million drivers, we require 5 GB of storage:

$$5 \times 10^6 \times 1000 = 5\ GB$$

Additionally, if we have around 100,00 new drivers registered daily, we’ll need around 100 MB to store them.

Driver location metadata

Let’s assume we need around 36 Bytes to store the driver’s location updates. If we have five million drivers, we need around 180 MB of storage just for the drivers’ locations.

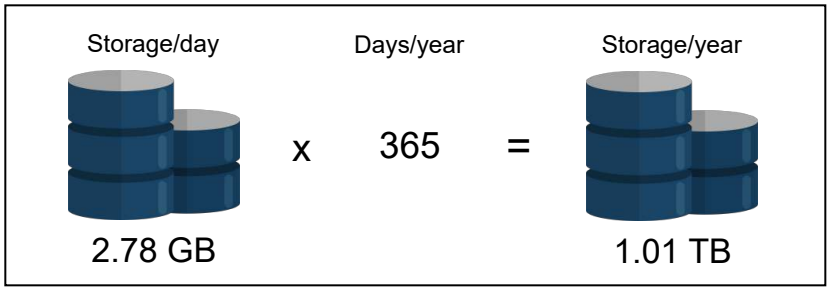
Trip metadata

Let’s assume we need around 100 Bytes to store single trip information, including trip ID, rider ID, driver ID, and so on. If we have 20 million daily rides, we need around 2 GB of storage for the trip data.

Let’s calculate the total storage required for Uber in a single day:

Storage Capacity Estimation

Number of drivers (millions)	5
Storage required to store a driver's location (Bytes)	36
Total storage required to store drivers’ locations (MB per day)	180
Number of trips (millions)	20
Storage required to store a trip (Bytes)	100
Total storage required to store trips (GB per day)	2
Storage required for new riders daily (MB per day)	500
Storage required for new drivers daily (MB per day)	100
Total storage (GB per day)	2.78



Total storage required by Uber in an year

> **Note:** We can adjust the values in the table to see how the estimations change.

Bandwidth estimation

We'll only consider drivers' location updates and trip data for bandwidth calculation since other factors don't require significant bandwidth. We have 20 million daily rides, which means we have approximately 232 trips per second.

$$\frac{20000000}{86400} \approx 232\ trips\ per\ second$$

Now, each trip takes around 100 Bytes. So, it takes around 23 KB per second of bandwidth.

$$232 \times 100\ =\ 23\ KB \times 8\ =\ 185\ kbps$$

As already stated, the driver's location is updated every four seconds. If we receive the driver's ID (3 Bytes) and location (16 Bytes), our application will take the following bandwidth:

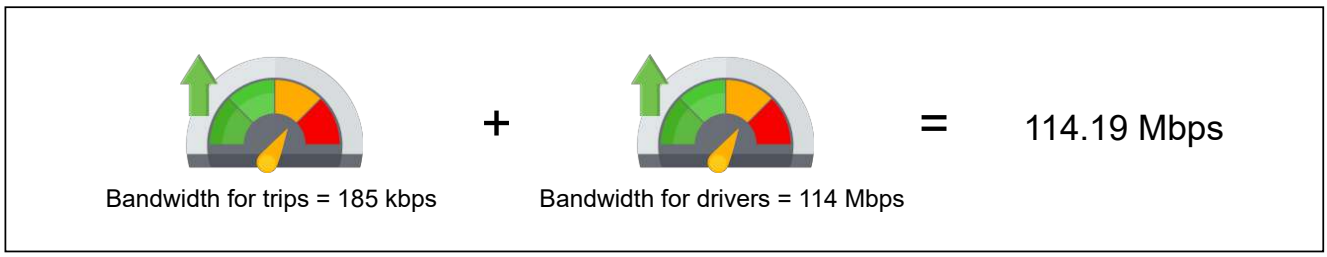
$$3M\ active\ drivers \times (3+16)B = 57MB \times 8\ = \frac{456}{4}\ = 114\ Mbps$$

These bandwidth requirements are modest because we didn't include bandwidth needs for maps and other components that are present in the real Uber service.

Bandwidth Requirements

Trips per second	232
Bandwidth for each trip (Bytes)	100

Total bandwidth for trips (kilo bits per second)	f 185.6
Active drivers (millions)	3
Bandwidth for each driver (Bytes)	19
Bandwidth for drivers (Megabits per second)	f 114
Total bandwidth (Megabits per second)	f 114.19



Total bandwidth required by Uber

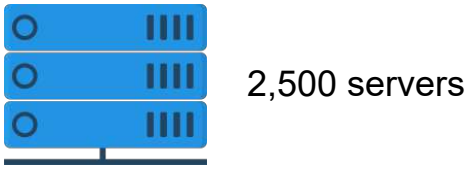
> **Note:** We can adjust the values in the table to see how the requirements change.

We’ve ignored the bandwidth from the Uber service to users because it was very small. More bandwidth will be required for sending map data from the Uber service to users, which we’ve also discussed in the Google Maps chapter.

Number of servers estimation

We need to handle concurrent requests coming from 20 million daily active users. We'll use the following formula to estimate a pragmatic number of servers. We established this formula in the Back-of-the-envelope Calculations chapter:

$$\frac{Number\ of\ daily\ active\ users}{RPS\ of\ a\ server} = \frac{20 \times 10^6}{8000} = 2500$$



Number of servers required for the Uber service

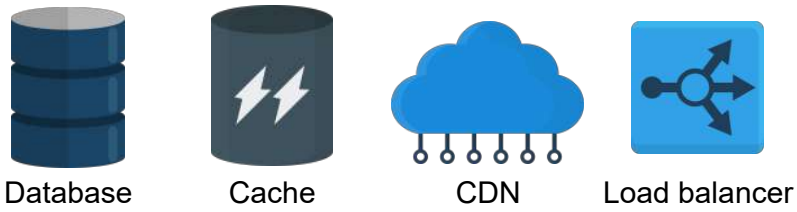
Estimating the Number of Servers

Daily active users (millions)	20
RPS of a server	8000
Number of servers required	f 2500

> **Note:** We can adjust the values in the table to see how the estimations change.

Building blocks we will use

The design of Uber utilizes the following building blocks:



Building blocks in high-level design of Uber

- **Databases** store the metadata of riders, drivers, and trips.
- **A cache** stores the most requested data for quick responses.
- **CDNs** are used to effectively deliver content to end users, reducing delay and the burden on end-servers.
- **A load balancer** distributes the read/write requests among the respective services.

Riders’ and drivers’ devices should have sufficient bandwidth and GPS equipment for smooth navigation with maps.

> **Note:** The information provided in this chapter is inspired by the engineering blog of Uber.

← Back

Next →

High-level Design of Uber

Learn how to design an Uber system.

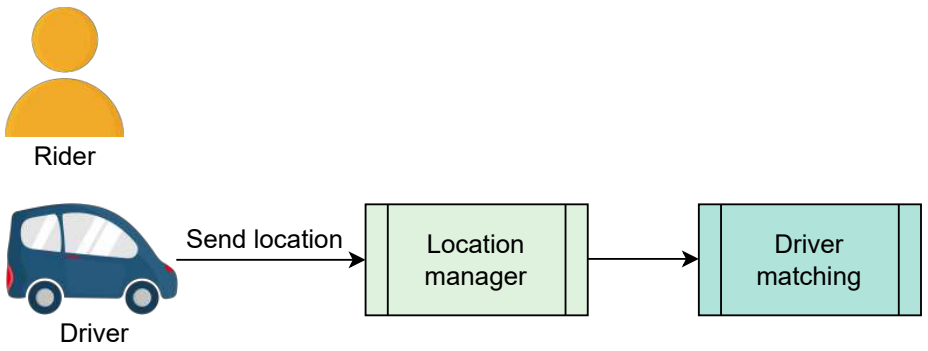
We'll cover the following

Workflow of our application

Before diving deep into the design, let's understand how our application works. The following steps show the workflow of our application:

1. All the nearby drivers except those already serving rides can be seen when the rider starts our application.
2. The rider enters the drop-off location and requests a ride.
3. The application receives the request and finds a suitable driver.

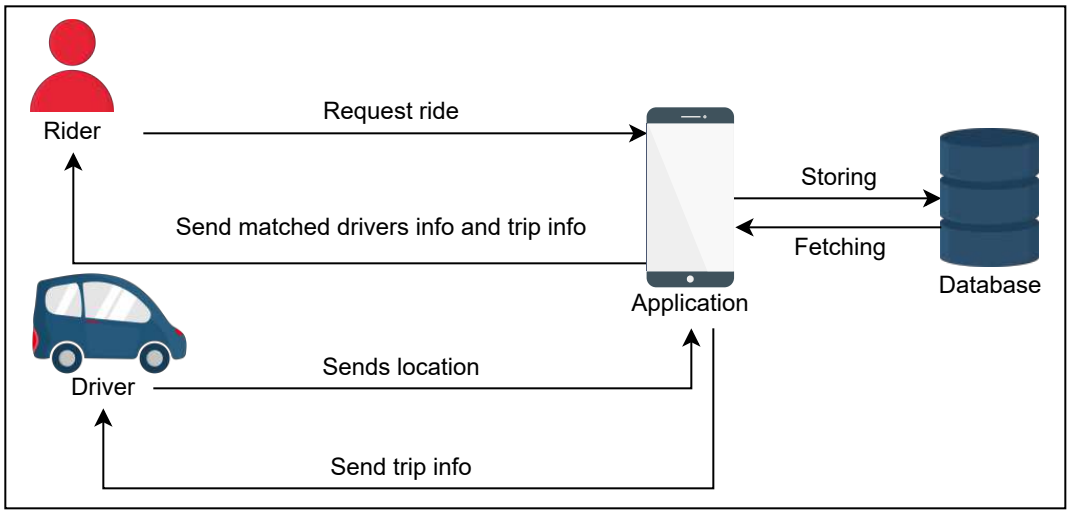
4. Until a matching driver is found, the status will be “Waiting for the driver to respond.”
5. The drivers report their location every four seconds. The application finds the trip information and returns it to the driver.
6. The driver accepts or rejects the request:
    - The driver accepts the request, and status information is modified on both the rider’s and the driver’s applications. The rider finds that they have successfully matched and obtains the driver’s information.
    - The driver refuses the ride request. The rider restarts from step 2 and rematches to another driver.



The rider is yet to open the application. The driver constantly updates their location

High-level design of Uber

At a high level, our system should be able to take requests for a ride from the rider and return the matched driver information and trip information to the rider. It also regularly takes the driver's location. Additionally, it returns the trip and rider information to the driver when the driver is matched to a rider.



High-level design

API design

Let's discuss the design of APIs according to the functionalities we provide. We'll design APIs to translate our feature set into technical specifications.

We won't repeat the description of repeating parameters in the following APIs.

Update driver location

```
updateDriverLocation(driverID, oldlat, oldlong, newlat, newlong )
```

Parameter	Description

`driverID`	The ID of the driver
`oldlat`	The previous latitude of the driver
`oldlong`	The previous longitude of the driver
`newlat`	The new latitude of the driver
`newlong`	The new longitude of the driver

The `updateDriverLocation` API is used to send the driver's coordinates to the driver location servers. This is where the location of the driver is updated and communicated to the riders.

Find nearby drivers

```
findNearbyDrivers(riderID, lat, long)
```

Parameter	Description
`riderID`	The ID of the rider
`lat`	The latitude of the rider
`long`	The longitude of the rider

The `findNearbyDrivers` API is used to send the location of the rider for whom we want to find the nearby drivers.

Request a ride

```
requestRide(riderID, lat, long, dropOfflat,dropOfflong, typeOfVehicle)
```

Parameter	Description
`lat`	The current latitude of the rider
`long`	The current longitude of the rider
`dropOfflat`	The latitude of the rider's drop-off location
`dropOfflong`	The longitude of the rider's drop-off location
`typeOfVehicle`	The type of vehicle required by the rider—for example, economy, and so on.

The `requestRide` API is used to send the location of the rider and the type of vehicle the rider needs.

Show driver ETA

```
showETA(driverID, eta)
```

Parameter	Description
`eta`	The estimated time of arrival of the driver

The `showEta` API is used to show the estimated time of arrival to the rider.

Confirm pickup

```
confirmPickup(driverID, riderID, timestamp)
```

Parameter	Description

`timestamp`	The time at which the driver picked up the rider

The `confirmPickup` API is used to determine when the driver has picked up the rider.

Show trip updates

```
showTripUpdates(tripID, riderID, driverID, driverlat, driverlong, time_elapsed, time_remaining)
```

Parameter	Description
`tripID`	The ID of the trip
`driverlat`	The latitude of the driver
`driverlong`	The longitude of the driver
`time_elapsed`	The total time of the trip
`time_remaining`	The time remaining (extract the current time from the E the destination

The `showTripUpdates` API is used to show the updates of the trip, including the position of the driver and the time remaining to reach the destination.

End the trip

```
endTrip(tripID, riderID, driverID ,time_elapsed, lat, long)
```

The `endTrip` API is used to end the trip.

Back

Requirements of Uber's Design

Next

Detailed Design of Uber

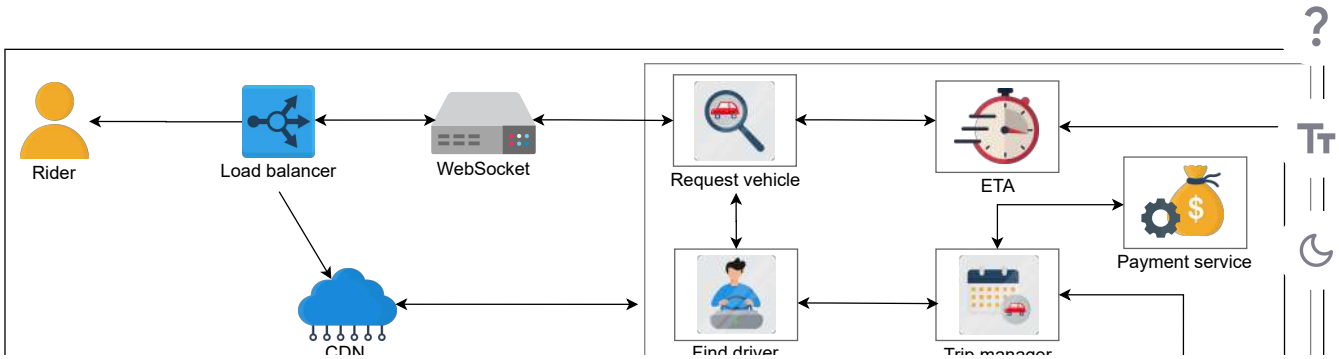
Detailed Design of Uber

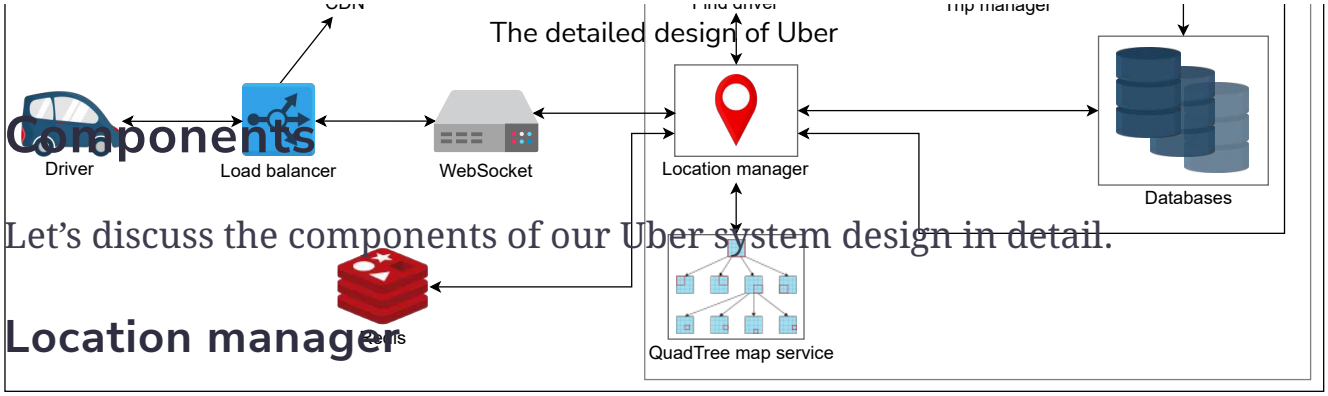
Learn about the detailed design of the Uber system.

We'll cover the following

- Components
  - Location manager
  - QuadTree map service
  - Request vehicle
  - Find driver
  - Trip manager
  - ETA service
    - DeepETA
  - Database
    - Storage schema
  - Fault tolerance
  - Load balancers
  - Cache

Let's look at the detailed design of our Uber system and learn how the various components work together to offer a functioning service:





Let’s discuss the components of our Uber system design in detail.

Location manager

The riders and drivers are connected to the **location manager** service. This service shows the nearby drivers to the riders when they open the application. This service also receives location updates from the drivers every four seconds. The location of drivers is then communicated to the

a map. The location manager saves the last location of all drivers in a database and saves the route followed by the drivers on a trip.

QuadTree map service

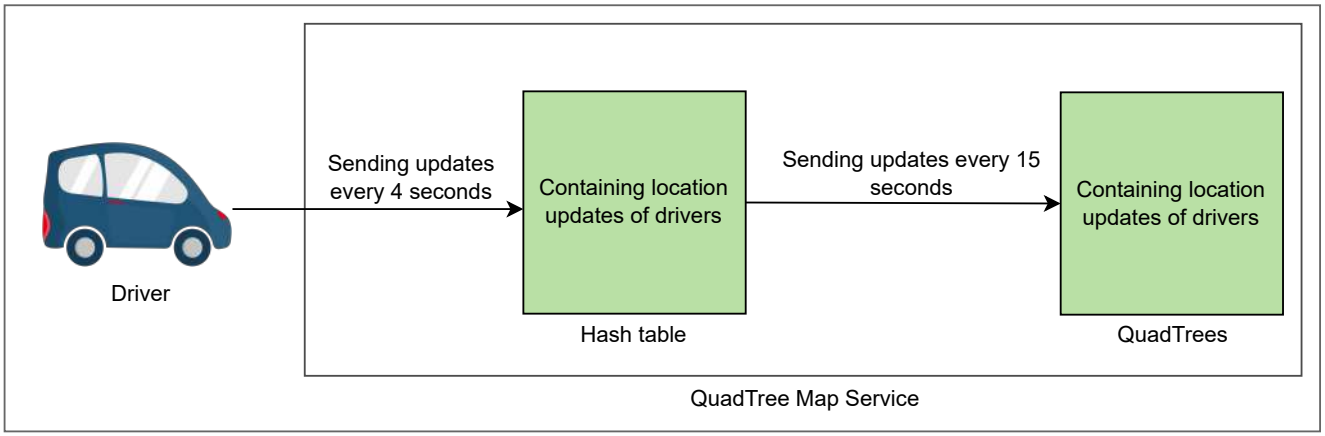
The **QuadTree map service** updates the location of the drivers. The main problem is how we deal with finding nearby drivers efficiently.

We’ll modify the solution discussed in the Yelp chapter according to our requirements. We used QuadTrees on Yelp to find the location. QuadTrees help to divide the map into segments. If the number of drivers exceeds a certain limit, for example, 500, then we split that segment into four more child nodes and divide the drivers into them.

Each leaf node in QuadTrees contains segments that can’t be divided further. We can use the same QuadTrees for finding the drivers. The most significant difference we have now is that our QuadTree wasn’t designed with regular upgrades in consideration. So, we have the following issues with our dynamic segment solution.

We must update our data structures to point out that all active drivers update their location every four seconds. It takes a longer amount of time to modify the QuadTree whenever a driver's position changes. To identify the driver's new location, we must first find a proper grid depending on the driver's previous position. If the new location doesn't match the current grid, we should remove the driver from the current grid and shift it to the correct grid. We have to repartition the new grid if it exceeds the driver limit, which is the number of drivers for each region that we set initially. Furthermore, our platform must tell both the driver and the rider, of the car's current location while the ride is in progress.

To overcome the above problem, we can use a hash table to store the latest position of the drivers and update our QuadTree occasionally, say after 10–15 seconds. We can update the driver's location in the QuadTree around every 15 seconds instead of four seconds, and we use a hash table that updates every four seconds and reflects the drivers' latest location. By doing this, we use fewer resources and time.



Updating driver's location

Request vehicle

The rider contacts the **request vehicle** service to request a ride. The rider adds the drop-off location here. The request vehicle service then

communicates with the find driver service to book a vehicle and get the details of the vehicle using the location manager service.

Find driver

The **find driver** service finds the driver who can complete the trip. It sends the information of the selected driver and the trip information back to the request vehicle service to communicate the details to the rider. The find driver service also contacts the trip manager to manage the trip information.

Trip manager

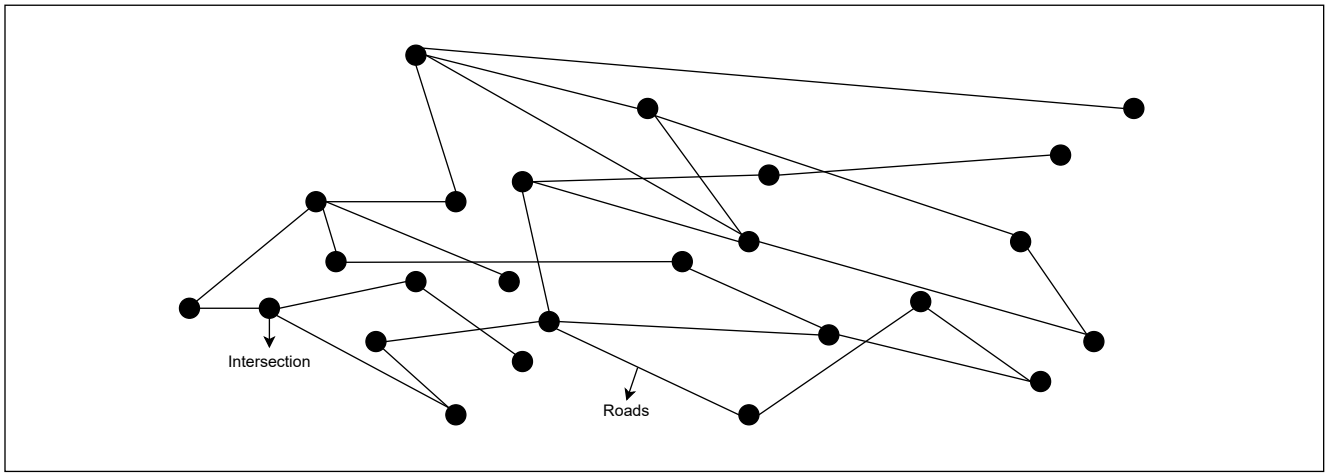
The **trip manager** service manages all the trip-related tasks. It creates a trip in the database and stores all the information of the trip in the database.

ETA service

The **ETA service** deals with the estimated time of arrival. It shows riders the pickup ETA when their trip is scheduled. This service considers factors such as route and traffic. The two basic components of predicting an ETA given an origin and destination on a road network are the following:

- Calculate the shortest route from origin to destination.
- Compute the time required to travel the route.

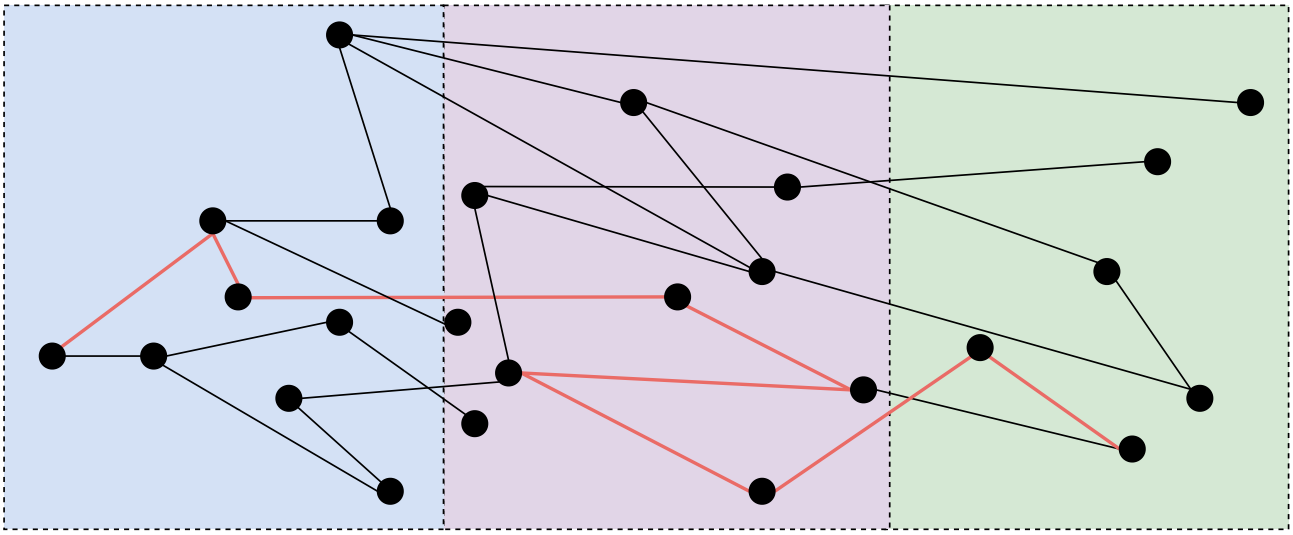
The whole road network is represented as a graph. Intersections are represented by nodes, while edges represent road segments. The graph also depicts one-way streets, turn limitations, and speed limits.



Graph representation

To identify the shortest path between source and destination, we can utilize routing algorithms such as Dijkstra's algorithm. However, Dijkstra, or any other algorithm that operates on top of an unprocessed graph, is quite slow for such a system. Therefore, this method is impractical at the scale at which these ride-hailing platforms operate.

To resolve these issues, we can split the whole graph into partitions. We preprocess the optimum path inside partitions using **contraction hierarchies** and deal with just the partition boundaries. This strategy can considerably reduce the time complexity since it partitions the graph into layers of tiny cells that are largely independent of one another. The preprocessing stage is executed in parallel in the partitions when necessary to increase speed. In the illustration below, all the partitions process the best route in parallel. For example, if each partition takes one second to find the path, we can have the complete path in one second since all partitions work in parallel.



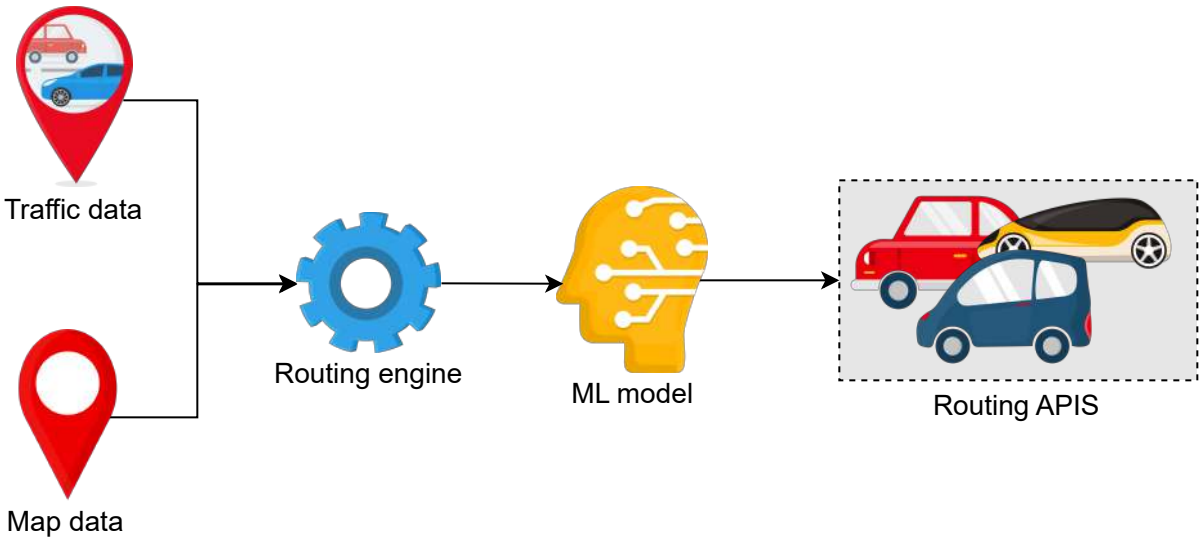
Partitioning the graph

Once we determine the best route, we calculate the expected time to travel the road segment while accounting for traffic. The traffic data will be the edge weights between the nodes.

DeepETA

We use a machine learning component named **DeepETA** to deliver an immediate improvement to metrics in production. It establishes a model foundation that can be reused for multiple consumer use cases.

We also use a routing engine that uses real-time traffic information and map data to predict an ETA to traverse the best path between the source and the destination. We use a post-processing ML model that considers spatial and temporal parameters, such as the source, destination, time of the request, and knowledge about real-time traffic, to forecast the ETA residual.



DeepETA

Database

Let's select the database according to our requirements:

- The database must be able to scale horizontally. There are new customers and drivers on a regular basis, so we should be able to add more storage without any problems.
- The database should handle a large number of reads and writes because the location of drivers is updated every four seconds.
- Our system should never be down.

We've already discussed the various database types and their specifications in the Database chapter. So, according to our understanding and requirements (high availability, high scalability, and fault tolerance), we can use Cassandra to store the driver's last known location and the trip information after the trip has been completed, and there will be no updates to it.

We can use a MySQL database to store trip information while it's in progre
We use MySQL for in-progress trips for frequent updates since the trip information is relational, and it needs to be consistent across tables. We us

Cassandra because the data we store is enormous and it increases continuously.

> **Note:** Recently, Uber migrated their data storage to Google Cloud Spanner. It provides global transactions, strongly consistent reads, and automatic multisite replication and failover features.

Points to Ponder

Question 1

Why haven't we used only the MySQL database?

Show Answer

1 of 2

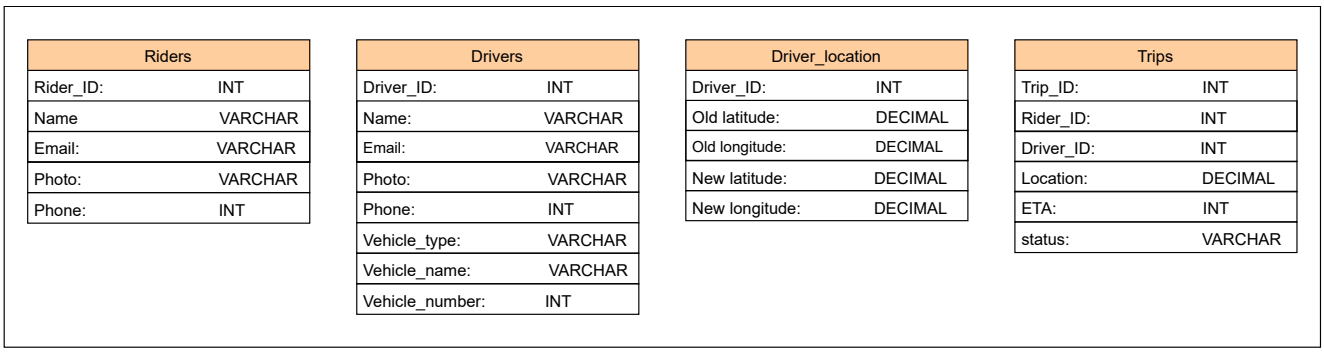
Storage schema

On a basic level in the Uber application, we need the following tables:

- **Riders:** We store the rider's related information, such as ID, name, email, photo, phone number, and so on.
- **Drivers:** We store the driver's related information, such as ID, name, email, photo, phone number, vehicle name, vehicle type, and so on.
- **Driver_location:** We store the driver's last known location.

- **Trips:** We store the trip's related information, such as trip ID, rider ID, driver ID, status, ETA, location of the vehicle, and so on.

The following illustration visualizes the data model:



The storage schema

Fault tolerance

For availability, we need to have replicas of our database. We use the primary-secondary replication model. We have one primary database and a few secondary databases. We synchronously replicate data from primary to secondary databases. Whenever our primary database is down, we can use a secondary database as a primary one.

Point to Ponder

Question

How will we handle a driver's slow and disconnecting network?

Show Answer

Load balancers

We use load balancers between clients (drivers and riders) and the application servers to uniformly distribute the load among the servers. The requests are routed to the specified server that provides the requested service.

Cache

A million drivers need to send the updated location every four seconds. A million requests to the QuadTree service affects how well it works. For this, we first store the updated location in the hash table stored in Redis. Eventually, these values are copied into the persistent storage every 10–15

Payment Service and Fraud Detection in Uber Design

Learn how the payment service of Uber works and how they deal with payment fraud.

We'll cover the following

Our goal in this lesson is to highlight the need to secure the system against malicious activities. We'll discuss fraudulent activities in the context of payment. We'll first briefly go through how the payment system works, and then we'll cover how to guard our system against malicious activity.

The payment service deals with the payments side of the Uber application. It includes collecting payment from the rider to give it to the driver. As the trip completes, it collects the trip information, calculates the cost of the journey and initiates the payment process.

Major functionality

The Uber payments platform includes the following functionality:

- **New payment options:** It adds new payment options for users.
- **Authorization:** It authorizes the payments for transactions.
- **Refund**: It refunds a payment that was authorized before.
- **Charging:** It moves money from a user account to Uber.

What to prevent

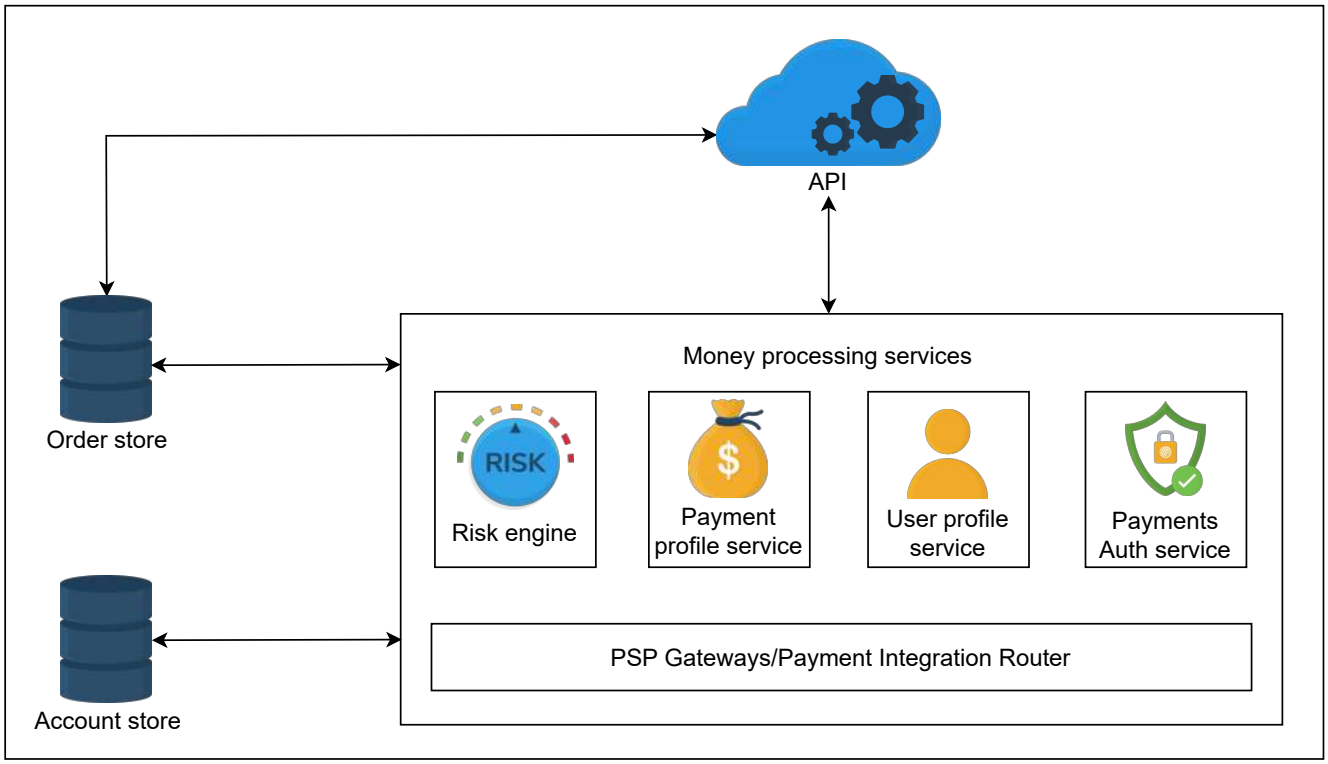
We need to prevent the following scenarios for the success of the payment system:

- Lack of payment
- Duplicate payments
- Incorrect payment
- Incorrect currency conversion
- Dangling authorization

The Uber payment platform is based on the double-entry bookkeeping method.

Design

Let’s look at the design of the payment platform in Uber and learn how different components work together to complete the transactions:



The payment platform in Uber

The key components used in the payment service are as follows:

- **API**: The Uber API is used to access the payment service.
- **Order store**: This stores all the orders. Orders collect payments and contain information regarding money flow between different riders and drivers.
- **Account store**: This stores all the accounts of riders and drivers.
- **Risk engine**: The risk engine analyzes different risks involved in collecting payment from a particular rider. It checks the history of the rider—for example, the rating of the rider, if the rider has sufficient funds in their rider account, any pending dues, if the rider cancels the rides frequently, and so on.
- **Payment profile service**: This provides information on the payment mechanisms, such as credit and debit cards.
- **User profile service**: This provides information regarding users' payments.

- **Payment authorization service**: This offers payment authentication services.
- **PSP gateways**: This connects with payment service providers

Workflow

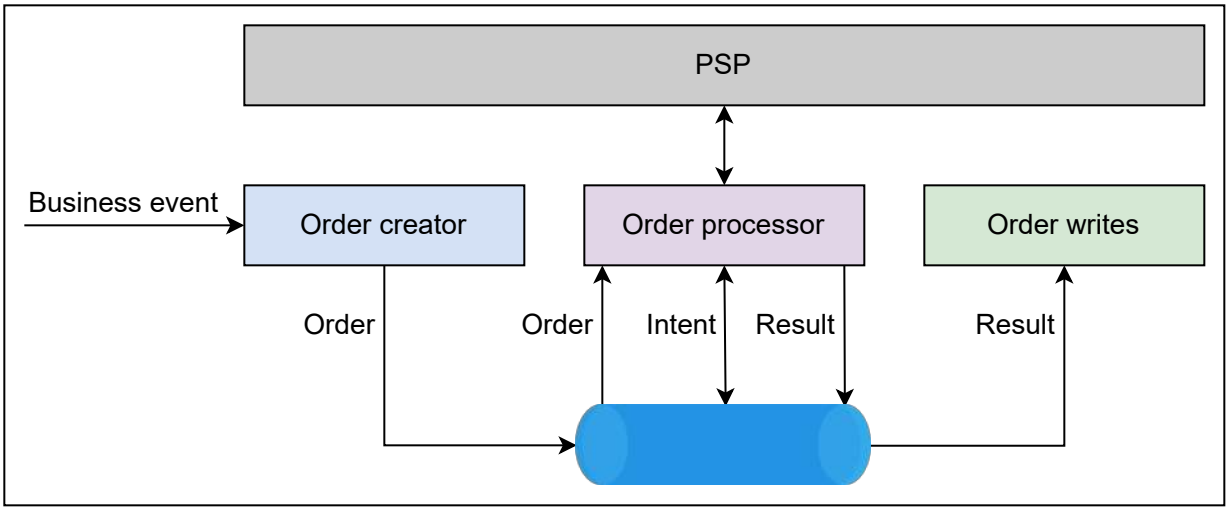
The following steps show the workflow of the payment service:

1. When a rider requests a ride, the Uber application uses the Uber API to request the risk engine to check the risks involved.
2. The risk engine obtains user information from the user profile service and evaluates the risk involved.
3. If the risks are high, the rider's request isn't entertained.
4. If the risks are low, the risk engine creates an authorization token and sends it to the payment profile service (for record-keeping), which fetches that token and sends it to the payment authorization service.
5. The payment authorization service sends that request to the PSP gateway, which contacts the service provider for authorization.
6. The PSP gateway sends the authorization token back to the payment authorization service, which sends the token back to the Uber application that the trip request is approved.
7. After the trip is completed, the Uber application uses the API to send the payment request to the PSP gateway with authorization data.
8. The PSP gateway contacts the service provider and sends the status back to the Uber application through the API.

The Uber payment service is constructed with a set of microservices grouped together as a stream-processing architecture.

Apache Kafka

Kafka is an open-source stream-processing software platform. It's the primary technology used in payment services. Let's see how Kafka helps to process an order:



Payment service with Kafka

The order creator gets a business event—for example, the trip is finished. The order creator creates the money movement information and the metadata. This order creator publishes that information to Kafka. Kafka processes that order and sends it to the order processor. The order processor then takes that information from Kafka, processes it, and sends it as intent to Kafka. From that, the order processor again processes and contacts the PSP. The order processor then takes the answer from PSP and transmits it to Kafka as a result. The result is then saved by the order writer.

The key capabilities of Kafka that the payment service uses are the following:

- It works like message queues to publish and subscribe to message queues.
- It stores the records in a way that is fault tolerant.
- It processes the payment records asynchronously.

Fraud detection

Fraud detection is a critical operational component of our design. Payment fraud losses are calculated as a percentage of gross amounts. Despite fraud activities accounting for a tiny portion of gross bookings, these losses

considerably influence earnings. Moreover, if malicious behavior isn't promptly detected and handled, it may be further leveraged, which results in significant losses for the business. Earlier, we discussed the risk engine that detects fraud before the trip starts. Here, we'll focus on the fraud detection that happens during trips or at the end of the trips.

Fraud detection is challenging because many instances of fraud are like detecting zero-day security bugs. Therefore, our system needs intelligence to detect anomalies, and it also needs accountability for automated decisions in the form of human-led audits.

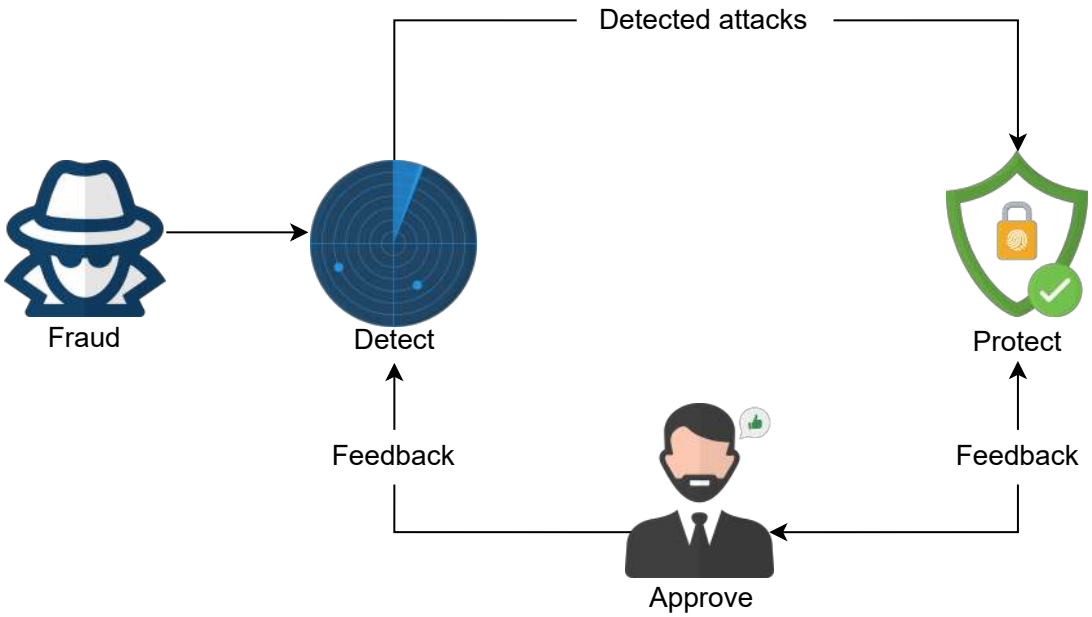
The activities that are considered fraudulent by Uber are as follows:

- A driver deliberately increases the time or distance of a trip in a dishonest manner—for example, if they take a much longer route than necessary to the rider's destination.
- A driver manipulates the GPS data or uses fake location GPS apps.
- A driver confirms trips with no intention of completing them, forcing riders to cancel.
- A driver creates false driver or rider accounts for fraudulent purposes.
- A driver provides incorrect or inaccurate information about themselves when opening their account.
- A driver drives a vehicle that hasn't been approved by Uber.
- A driver claims fraudulent fees or charges, such as unwarranted cleaning fees.
- A driver willfully confirms or completes fraudulent trips.

RADAR

Uber introduced **RADAR**, a human-assisted AI fraud detection and mitigation solution to handle the discussed scenarios of fraud. The RADAR system detects fraud by analyzing the activity time series of the payment

system. It then generates a rule for it and stops it from further processing. This proactive approach can help to detect unseen fraud in real time. This detection model also uses human knowledge for continuous improvements. Here, we'll briefly discuss how RADAR works:



The RADAR fraud detection solution

RADAR recognizes the beginning of a fraud attempt and creates a rule to prevent it. The fraud analyst is involved in the next step. They review the rule and approve or reject it if required. They then send feedback (approved or not approved) to the protection system. The feedback is also sent to the fraud detection system by the fraud analysts to improve detection.

Furthermore, the system examines the data in the following time dimensions:

- It examines the trip time when a trip has been completed. Multiple factors are involved in real-time trip monitoring. For instance, the system can check whether or not the driver and rider locations are the same. The system can also monitor the speed and analyze the real

traffic to check if the driver is intentionally driving slow or if there's traffic congestion.

- Payment settlement time refers to obtaining payment processing data. It might take days or weeks to settle.

We've just discussed the basic details of how guarding against malicious activity at scale is necessary for the success of a business. In the proposed model, the human experts are involved, potentially decreasing the

Requirements of Twitter’s Design

Understand the requirements and estimation for Twitter's design.

We'll cover the following

Requirements

Let’s understand the functional and non-functional requirements below:

Functional requirements

The following are the functional requirements of Twitter:

- **Post Tweets**: Registered users can post one or more Tweets on Twitter.
- **Delete Tweets**: Registered users can delete one or more of their Tweets on Twitter.

- **Like or dislike Tweets**: Registered users can like and dislike public and their own Tweets on Twitter.
- **Reply to Tweets**: Registered users can reply to the public Tweets on Twitter.
- **Search Tweets**: Registered users can search Tweets by typing keywords, hashtags, or usernames in the search bar on Twitter.
- **View user or home timeline**: Registered users can view the user's timeline, which contains their own Tweets. They can also view the home's timeline, which contains followers' Tweets on Twitter.
- **Follow or unfollow the account**: Registered users can follow or unfollow other users on Twitter.
- **Retweet a Tweet**: Registered users


Back To Course Home

Grokking Modern System Design Interview for Engineers & Managers

16% completed

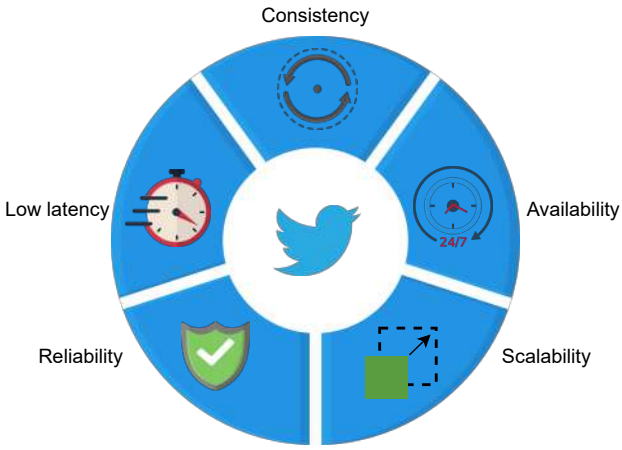
Non-functional requirements

- **Availability**: Many people and organizations use Twitter to communicate time-sensitive information (service outage messages). Therefore, our service

must be highly available and have a good uptime percentage.

- **Latency**: The latency to show the most recent top Tweets in the home timeline could be a little high. However, near real-time services, such as Tweet distribution to followers, must have low latency.
- **Scalability**: The workload on Twitter is read-heavy, where we have many readers and relatively few writers, which eventually requires the scalability of computational resources. Some estimates suggest a 1:1000 write-to-read ratio for Twitter. Although the Tweet is limited to 280 characters, and the video clip is limited to 140s by default, we need high storage capacity to store and deliver Tweets posted by public figures to their millions of followers.
- **Reliability**: All Tweets remain on Twitter. This means that Twitter never deletes its content. So there should be a promising strategy t prevent the loss or damage of the uploaded content.

- **Consistency**: There's a possibility that a user on the east coast of the US does not get an immediate status (like, reply, and so on.) update on the Tweet, which is liked or Retweeted by a user on the west coast of the US. However, the user on the west coast of the US needs an immediate status update on his like or reply. An effective technique is needed to offer rapid feedback to the user (who liked someone's post), then to other specified users in the same region, and finally to all worldwide users linked to the Tweet.



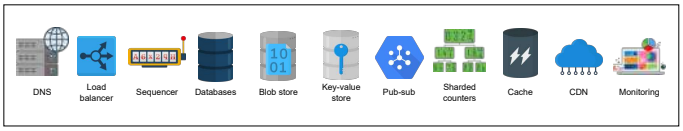
Non-functional requirements of Twitter

Furthermore, we must identify which essential resources must be estimated in the Twitter design. We used Twitte

as an example in our Back-of-the-Envelope chapter, so we won't repeat that exercise here.

Building blocks we will use

Twitter's design utilizes the following building blocks that we discussed in the initial chapters.



Building blocks used in Twitter design

- **DNS** is the service that maps human-friendly Twitter domain names to machine-readable IP addresses.
- **Load balancers** distribute the read/write requests among the respective services.
- **Sequencers** generate the unique IDs for the Tweets.
- **Databases** store the metadata of Tweets and users.
- **Blob stores** store the images and video clips attached with the Tweets.
- **Key-value stores** are used for multiple purposes such as indexing, identifying the specified

counter to update its value, identifying the Tweets of a particular user and many more.

- **Pub-sub** is used for real-time processing such as elimination of redundant data, organizing data, and much more.
- **Sharded counters** help to handle the count of multiple features such as viewing, liking, Retweeting, and so on,. of the accounts with millions of followers.
- A **cache** is used to store the most requested and recent data in RAM to give users a quick response.
- **CDN** helps end users to access the data with low latency.
- **Monitoring** analyses all outgoing and incoming traffic, identifies the redundancy in the storage system, figures out the failed node, and so on.

In the next lesson, we'll discuss the design of the Twitter system.

Back

System Design: Twitter

Nex ter

High-level Design

Completed