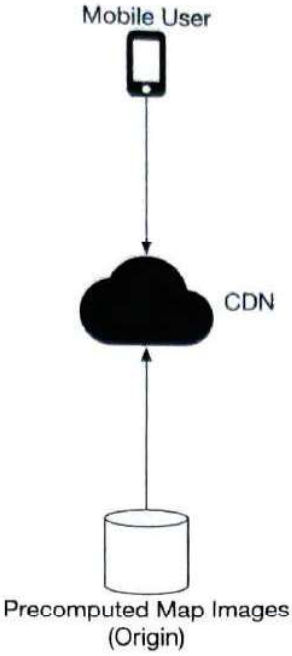


Figure 3.10: Pre-generated images are served by a CDN

In the diagram above, the mobile user makes an HTTP request to fetch a tile from the CDN. If the CDN has not yet served that specific tile before, it fetches a copy from the origin server, caches it locally, and returns it to the user. On subsequent requests, even if those requests are from different users, the CDN returns a cached copy without contacting the origin server.

This approach is more scalable and performant because the map tiles are served from the nearest point of presence (POP) to the client, as shown in Figure 3.11. The static nature of the map tiles makes them highly cacheable.

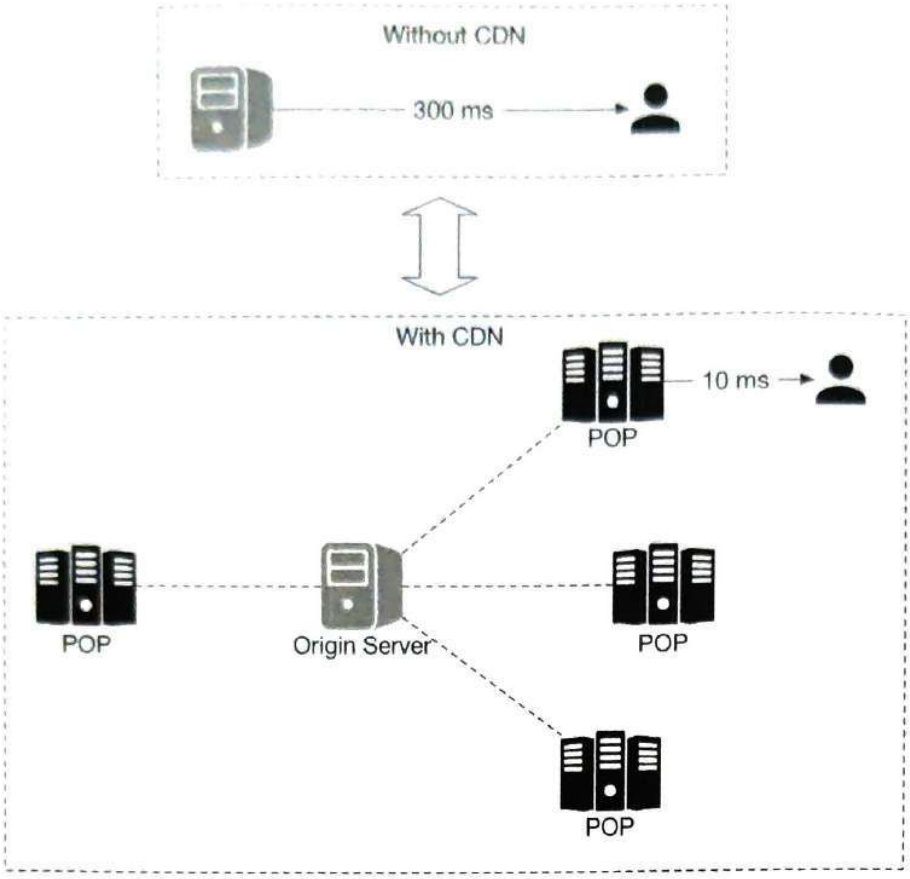


Figure 3.11: Without CDN vs with CDN

It is important to keep mobile data usage low. Let's calculate the amount of data the client needs to load during a typical navigation session. Note the following calculations don't take client-side caching into consideration. Since the routes a user takes could be similar each day, the data usage is likely to be a lot lower with client-side caching.

Data usage
Let's assume a user moves at 30km/h, and at a zoom level where each image covers a block of 200m $\times$ 200m (a block can be represented by a 256-pixel by 256-pixel image and the average image size is 100KB). For an area of 1km $\times$ 1km, we need 25 images or 2.5MB (25 $\times$ 100KB) of data. Therefore, if the speed is 30km/h, we need 75MB (30 $\times$ 2.5MB) of data per hour or 1.25MB of data per minute.

Next, we estimate the CDN data usage. At our scale, the cost is an important factor to consider.

Traffic through CDN

As mentioned earlier, we serve 5 billion minutes of navigation per day. This translates to 5 billion $\times$ 1.25MB = 6.25 billion MB per day. Hence, we serve 62,500MB $\left(\frac{6.25 \text{ billion}}{10^5 \text{ seconds in a day}}\right)$ of map data per second. With a CDN, these map images are going to be served from the POPs all over the world. Let's assume there are 200 POPs. Each POP would only need to serve a few hundred MBs $\left(\frac{62,500}{200}\right)$ per second.

There is one final detail in the map rendering design we have only briefly touched on. How does the client know which URLs to use to fetch the map tiles from the CDN? Keep in mind that we are using option 2 as discussed above. With that option, the map tiles are static and pre-generated based on fixed sets of grids, with each set representing a discrete zoom level.

Since we encode the grids in geohash, and there is one unique geohash per grid, computationally it is very efficient to go from the client's location (in latitude and longitude) and zoom level to the geohash, for the map tile. This calculation can be done on the client and we can fetch any static image tile from the CDN. For example, the URL for the image tile of Google headquarter could look like this: `https://cdn.map-provider.com/tiles/9q9hvu.png`

Refer to Chapter 1 Proximity Service on page 10 for a more detailed discussion of geohash encoding.

Calculating geohash on the client should work well. However, keep in mind that this algorithm is hardcoded in all the clients on all different platforms. Shipping changes to mobile apps is a time-consuming and somewhat risky process. We have to be sure that geohashing is the method we plan to use long-term to encode the collection of map tiles and that it is unlikely to change. If we need to switch to another encoding method for some reason, it will take a lot of effort and the risk is not low.

Here is another option worth considering. Instead of using a hardcoded client-side algorithm to convert a latitude/longitude (lat/lng) pair and zoom level to a tile URL, we could introduce a service as an intermediary whose job is to construct the tile URLs based on the same inputs mentioned above. This is a very simple service. The added operational flexibility might be worth it. This is a very interesting tradeoff discussion we could have with the interviewer. The alternative map rendering flow is shown in Figure 3.12.

When a user moves to a new location or to a new zoom level, the map tile service determines which tiles are needed and translates that information into a set of tile URLs to retrieve.

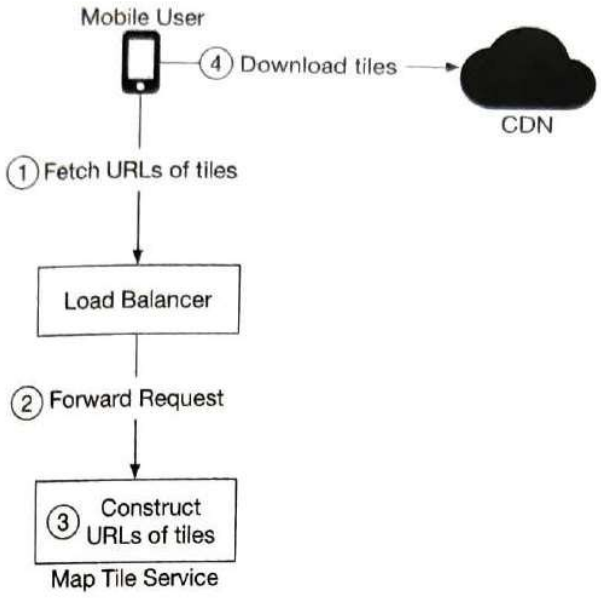


Figure 3.12: Map rendering

1. A mobile user calls the map tile service to fetch the tile URLs. The request is sent to the load balancer.
2. The load balancer forwards the request to the map tile service.
3. The map tile service takes the client's location and zoom level as inputs and returns 9 URLs of the tiles to the client. These tiles include the tile to render and the eight surrounding tiles.
4. The mobile client downloads the tiles from the CDN.

We will go into more detail on the precomputed map tiles in the design deep dive section.

Step 3 - Design Deep Dive

In this section, we first have a discussion about the data model. Then we talk about location service, navigation service, and map rendering in more detail.

Data model

We are dealing with four types of data: routing tiles, user location data, geocoding data, and precomputed map tiles of the world.

Routing tiles

As mentioned previously, the initial road dataset is obtained from different sources and authorities. It contains terabytes of data. The dataset is improved over time by the location data the application continuously collects from the users as they use the application.

This dataset contains a large number of roads and associated metadata such as names, county, longitude, and latitude. This data is not organized as graph data structures and

is not usable by most routing algorithms. We run a periodic offline processing pipeline, called routing tile processing service, to transform this dataset into the routing tiles we introduced. The service runs periodically to capture new changes to the road data.

The output of the routing tile processing service is routing tiles. There are three sets of these tiles at different resolutions, as described in the "Map 101" section on page 60. Each tile contains a list of graph nodes and edges representing the intersections and roads within the area covered by the tile. It also contains references to other tiles its roads connect to. These tiles together form an interconnected network of roads that the routing algorithms can consume incrementally.

Where should the routing tile processing service store these tiles? Most graph data is represented as adjacency lists [12] in memory. There are too many tiles to keep the entire set of adjacency lists in memory. We could store the nodes and edges as rows in a database, but we would only be using the database as storage, and it seems an expensive way to store bits of data. We also don't need any database features for routing tiles.

The more efficient way to store these tiles is in object storage like S3 and cache it aggressively on the routing service that uses those tiles. There are many high-performance software packages we could use to serialize the adjacency lists to a binary file. We could organize these tiles by their geohashes in object storage. This provides a fast lookup mechanism to locate a tile by lat/lng pair.

We discuss how the shortest path service uses these routing tiles shortly.

User location data

User location data is valuable. We use it to update our road data and routing tiles. We also use it to build a database of live and historical traffic data. This location data is also consumed by multiple data stream processing services to update the map data.

For user location data, we need a database that can handle the write-heavy workload well and can be horizontally scaled. Cassandra could be a good candidate.

Here is what a single row could look like:

user_id	timestamp	user_mode	driving_mode	location
101	1635740977	active	driving	(20.0, 30.5)

Table 3.2: Location table

Geocoding database

This database stores places and their corresponding lat/lng pair. We can use a key-value database such as Redis for fast reads, since we have frequent reads and infrequent writes. We use it to convert an origin or destination to a lat/lng pair before passing it to the route planner service.

Precomputed images of the world map

When a device asks for a map of a particular area, we need to get nearby roads and compute an image that represents that area with all the roads and related details. These

computations would be heavy and redundant, so it could be helpful to compute them once and then cache the images. We precompute images at different zoom levels and store them on a CDN, which is backed by cloud storage such as Amazon S3. Here is an example of such an image:

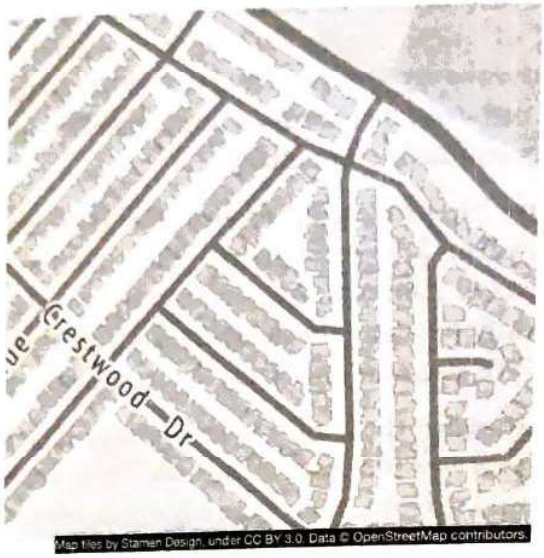


Figure 3.13: Precomputed tiles

Services

Now that we have discussed the data model, let's take a close look at some of the most important services: location service, map rendering service, and navigation service.

Location service

In the high-level design, we discussed how location service works. In this section, we focus on the database design for this service and also how user location is used in detail.

In Figure 3.14, the key-value store is used to store user location data. Let's take a close look.

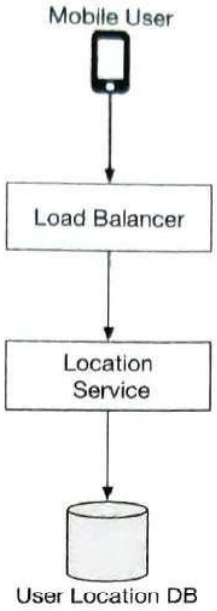


Figure 3.14: User location database

Given the fact we have 1 million location updates every second, we need to have a database that supports fast writes. A NoSQL key-value database or column-oriented database would be a good choice here. In addition, a user's location is continuously changing and becomes stale as soon as a new update arrives. Therefore, we can prioritize availability over consistency. The CAP theorem [13] states that we could choose two attributes among consistency, availability, and partition tolerance. Given our constraints, we would go with availability and partition tolerance. One database that is a good fit is Cassandra. It can handle our scale with a strong availability guarantee.

The key is the combination of `(user_id, timestamp)` and the value is a lat/lng pair. In this setup, `user_id` is the primary key and `timestamp` is the clustering key. The advantage of using `user_id` as the partition key is that we can quickly read the latest position of a specific user. All the data with the same partition key are stored together, sorted by `timestamp`. With this arrangement, the retrieval of the location data for a specific user within a time range is very efficient.

Below is an example of what the table may look like.

key (user_id)	timestamp	lat	long	user_mode	navigation_mode
51	132053000	21.9	89.8	active	driving

Table 3.3: Location data

How do we use the user location data?

User location data is essential. It supports many use cases. We use the data to detect new and recently closed roads. We use it as one of the inputs to improve the accuracy of our map over time. It is also an input for live traffic data.

To support these use cases, in addition to writing current user locations in our data-

base, we log this information into a message queue, such as Kafka. Kafka is a unified low-latency, high-throughput data streaming platform designed for real-time data feeds. Figure 3.15 shows how Kafka is used in the improved design.

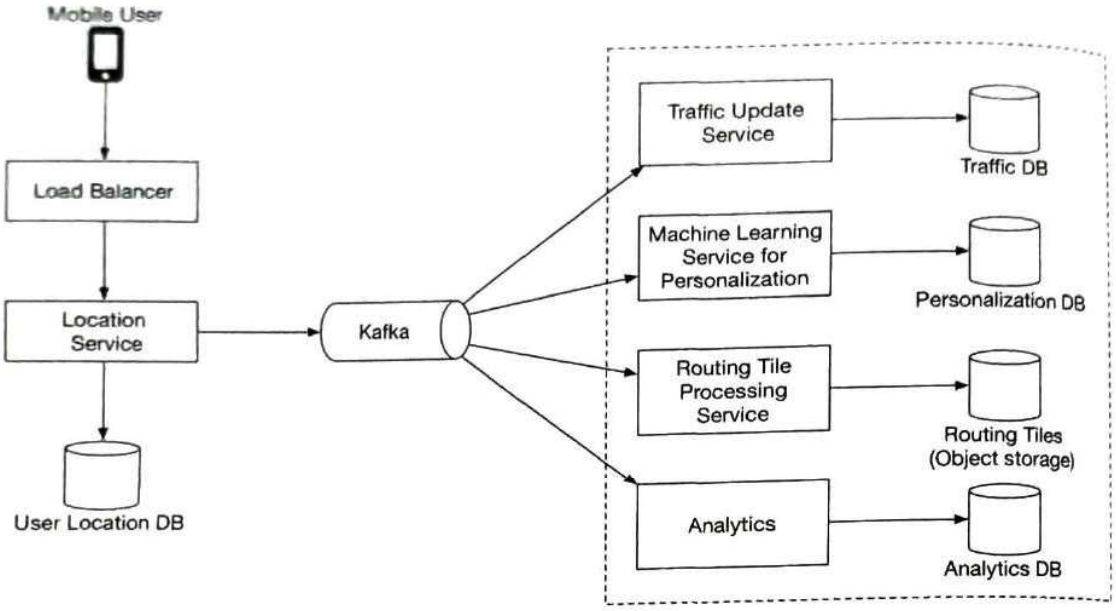


Figure 3.15: Location data is used by other services

Other services consume the location data stream from Kafka for various use cases. For instance, the live traffic service digests the output stream and updates the live traffic database. The routing tile processing service improves the map of the world by detecting new or closed roads and updating the affected routing tiles in object storage. Other services can also tap into the stream for different purposes.

Rendering map

In this section, we dive deep into precomputed map tiles and map rendering optimization. They are primarily inspired by the work of Google Design [3].

Precomputed tiles

As mentioned previously, there are different sets of precomputed map tiles at various distinct zoom levels to provide the appropriate level of map detail to the user, based on the client's viewport size and zoom level. Google Maps uses 21 zoom levels (Table 3.1). This is what we use, as well.

Level 0 is the most zoomed-out level. The entire map is represented by a single tile of size 256×256 pixels.

With each increment of the zoom level, the number of map tiles doubles in both north-south and east-west directions, while each tile stays at 256×256 pixels. As shown in Figure 3.16, at zoom level 1, there are 2×2 tiles, with a total combined resolution of 512×512 pixels. At zoom level 2, there are 4×4 tiles, with a total combined resolution of 1024×1024 pixels. With each increment, the entire set of tiles has 4x as many pixels

as the previous level. The increased pixel count provides an increasing level of detail to the user. This allows the client to render the map at the best granularities depending on the client's zoom level, without consuming excessive bandwidth to download tiles in excessive detail.

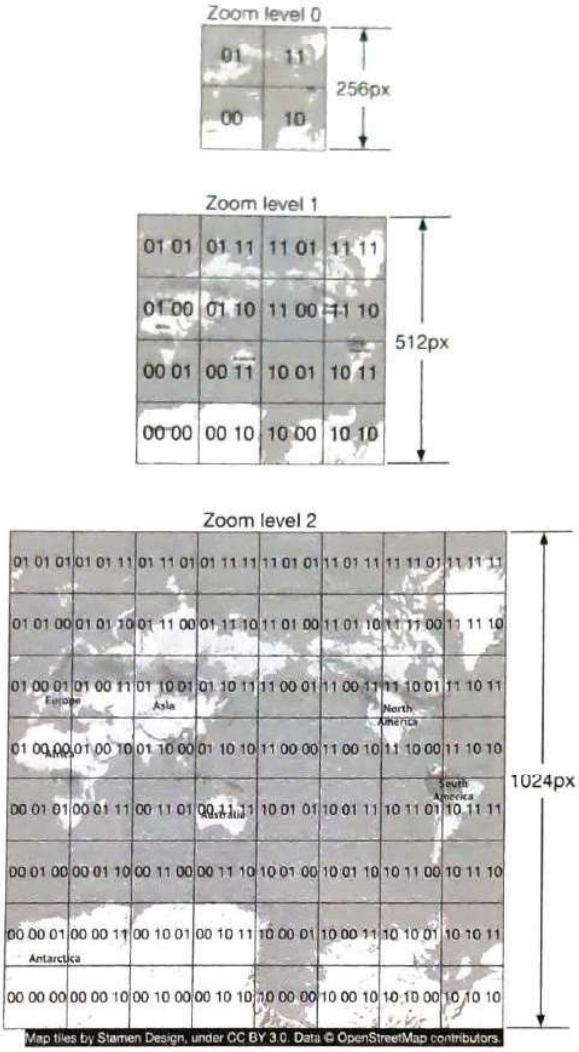


Figure 3.16: Zoom levels

Optimization: use vectors

With the development and implementation of WebGL, one potential improvement is to change the design from sending the images over the network, to sending the vector information (paths and polygons) instead. The client draws the paths and polygons from the vector information.

One obvious advantage of vector tiles is that vector data compresses much better than images do. The bandwidth saving is substantial.

A less obvious benefit is that vector tiles provide a much better zooming experience. With rasterized images, as the client zooms in from one level to another, everything gets

Shortest-path service

The shortest-path service receives the origin and the destination in lat/lng pairs and returns the top-k shortest paths without considering traffic or current conditions. This computation only depends on the structure of the roads. Here, caching the routes could be beneficial because the graph rarely changes.

The shortest-path service runs a variation of A* pathfinding algorithms against the routing tiles in object storage. Here is an overview:

- The algorithm receives the origin and destination in lat/lng pairs. The lat/lng pairs are converted to geohashes which are then used to load the start and end-points of routing tiles.
- The algorithm starts from the origin routing tile, traverses the graph data structure, and hydrates additional neighboring tiles from object storage (or its local cache if it has loaded it before) as it expands the search area. It's worth noting that there are connections from one level of tile to another covering the same area. This is how the algorithm could "enter" the bigger tiles containing only highways, for example. The algorithm continues to expand its search by hydrating more neighboring tiles (or tiles at different resolutions) as needed until a set of best routes is found.

Figure 3.18 (based on [14]) gives a conceptual overview of the tiles used in the graph traversal.

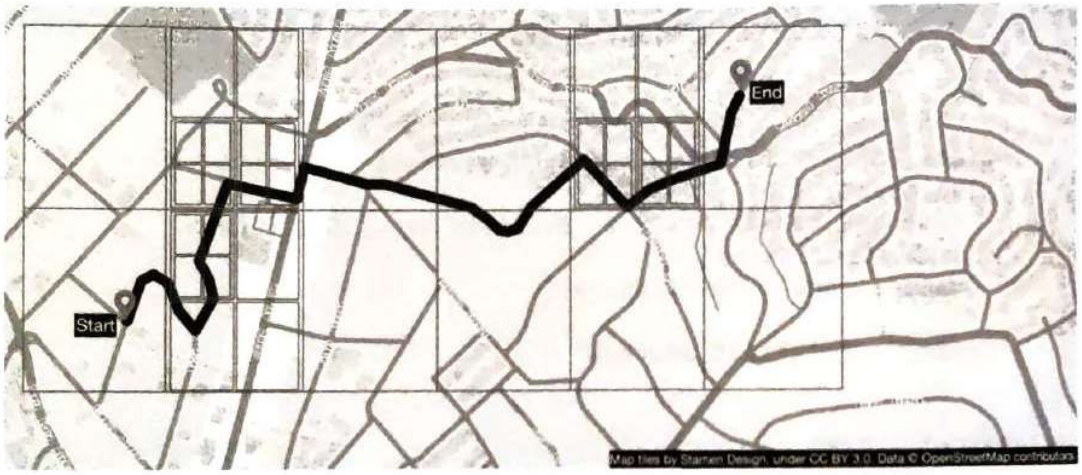


Figure 3.18: Graph traversal

ETA service

Once the route planner receives a list of possible shortest paths, it calls the ETA service for each possible route and gets a time estimate. For this, the ETA service uses machine learning to predict the ETAs based on the current traffic and historical data.

One of the challenges here is that we not only need to have real-time traffic data but also to predict how the traffic will look like in 10 or 20 minutes. These kinds of challenges need to be addressed at an algorithmic level and will not be discussed in this section. If

you are interested, refer to [15] and [16].

Ranker service

Finally, after the route planner obtains the ETA predictions, it passes this info to the ranker to apply possible filters as defined by the user. Some example filters include options to avoid toll roads or to avoid freeways. The ranker service then ranks the possible routes from fastest to slowest and returns top-k results to the navigation service.

Updater services

These services tap into the Kafka location update stream and asynchronously update some of the important databases to keep them up-to-date. The traffic database and the routing tiles are some examples.

The routing tile processing service is responsible for transforming the road dataset with newly found roads and road closures into a continuously updated set of routing tiles. This helps the shortest path service to be more accurate.

The traffic update service extracts traffic conditions from the streams of location updates sent by the active users. This insight is fed into the live traffic database. This enables the ETA service to provide more accurate estimates.

Improvement: adaptive ETA and rerouting

The current design does not support adaptive ETA and rerouting. To address this, the server needs to keep track of all the active navigating users and update them on ETA continuously, whenever traffic conditions change. Here we need to answer a few important questions:

- How do we track actively navigating users?
- How do we store the data, so that we can efficiently locate the users affected by traffic changes among millions of navigation routes?

Let's start with a naive solution. In Figure 3.19, `user_1`'s navigation route is represented by routing tiles `r_1`, `r_2`, `r_3`, ..., `r_7`.

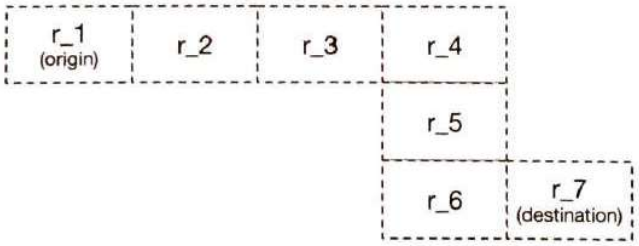


Figure 3.19: Navigation route

The database stores actively navigating users and routes information which might look like this:

```
user_1: r_1, r_2, r_3, ..., r_k
```

```
user_2: r_4, r_6, r_9, ..., r_n
user_3: r_2, r_8, r_9, ..., r_m
...
user_n: r_2, r_10, r_21, ..., r_1
```

Let's say there is a traffic incident in routing tile 2 (`r_2`). To figure out which users are affected, we scan through each row and check if routing tile 2 is in our list of routing tiles (see example below).

```
user_1: r_1, r_2, r_3, ..., r_k
user_2: r_4, r_6, r_9, ..., r_n
user_3: r_2, r_8, r_9, ..., r_m
...
user_n: r_2, r_10, r_21, ..., r_1
```

Assume the number of rows in the table is n and the average length of the navigation route is m. The time complexity to find all users affected by the traffic change is $O(n \times m)$.

Can we make this process faster? Let's explore a different approach. For each actively navigating user, we keep the current routing tile, the routing tile at the next resolution level that contains it, and recursively find the routing tile at the next resolution level until we find the user's destination in the tile as well (Figure 3.20). By doing this, we can get a row of the database table like this.

```
user_1, r_1, super(r_1), super(super(r_1)), ...
```

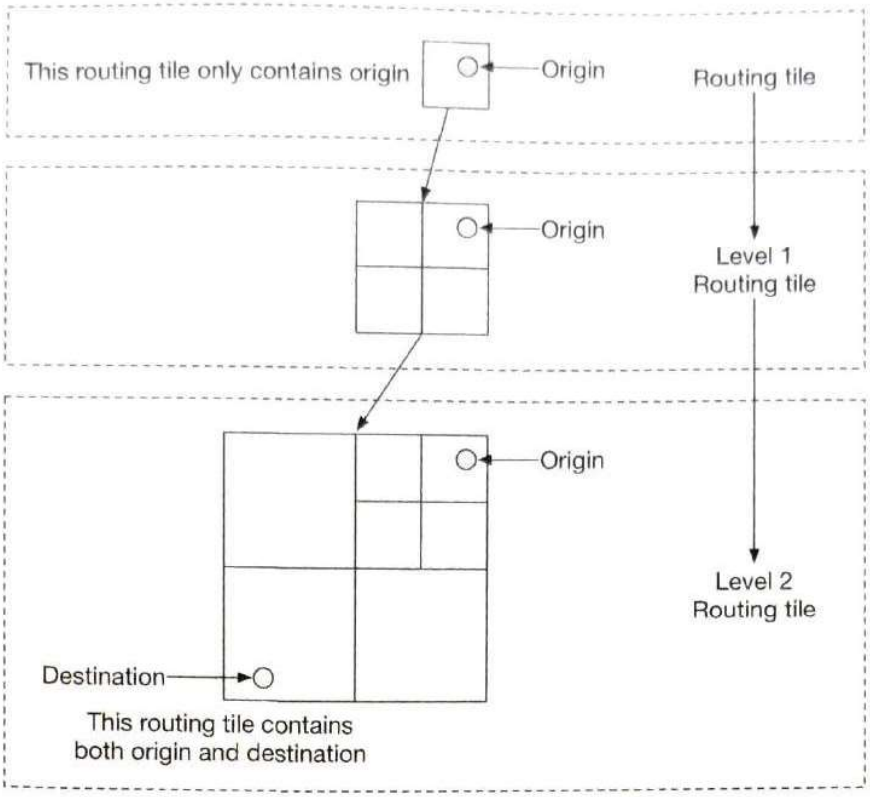


Figure 3.20: Build routing tiles

To find out if a user is affected by the traffic change, we need only check if a routing tile is inside the last routing tile of a row in the database. If not, the user is not impacted. If it is, the user is affected. By doing this, we can quickly filter out many users.

This approach doesn't specify what happens when traffic clears. For example, if routing tile 2 clears and users can go back to the old route, how do users know rerouting is available? One idea is to keep track of all possible routes for a navigating user, recalculate the ETAs regularly and notify the user if a new route with a shorter ETA is found.

Delivery protocols

It is a reality that during navigation, route conditions can change and the server needs a reliable way to push data to mobile clients. For delivery protocol from the server to the client, our options include mobile push notification, long polling, WebSocket, and Server-Sent Events (SSE).

- Mobile push notification is not a great option because the payload size is very limited (4,096 bytes for iOS) and it doesn't support web applications.
- WebSocket is generally considered to be a better option than long polling because it has a very light footprint on servers.
- Since we have ruled out the mobile push notification and long polling, the choice is mainly between WebSocket and SSE. Even though both can work, we lean towards WebSocket because it supports bi-directional communication and features such as

last-mile delivery might require bi-directional real-time communication.

For more details about ETA and rerouting, please refer to [15].

Now we have every piece of the design together. Please see the updated design in Figure 3.21.

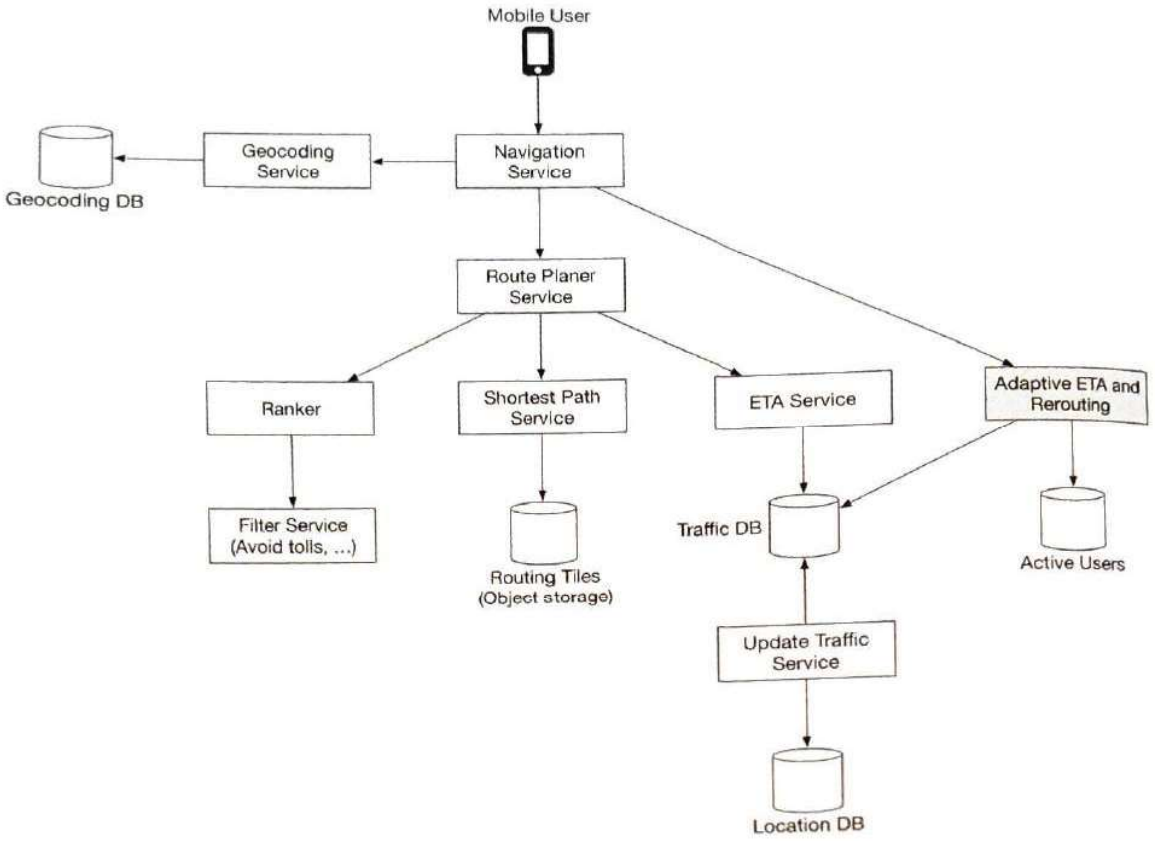


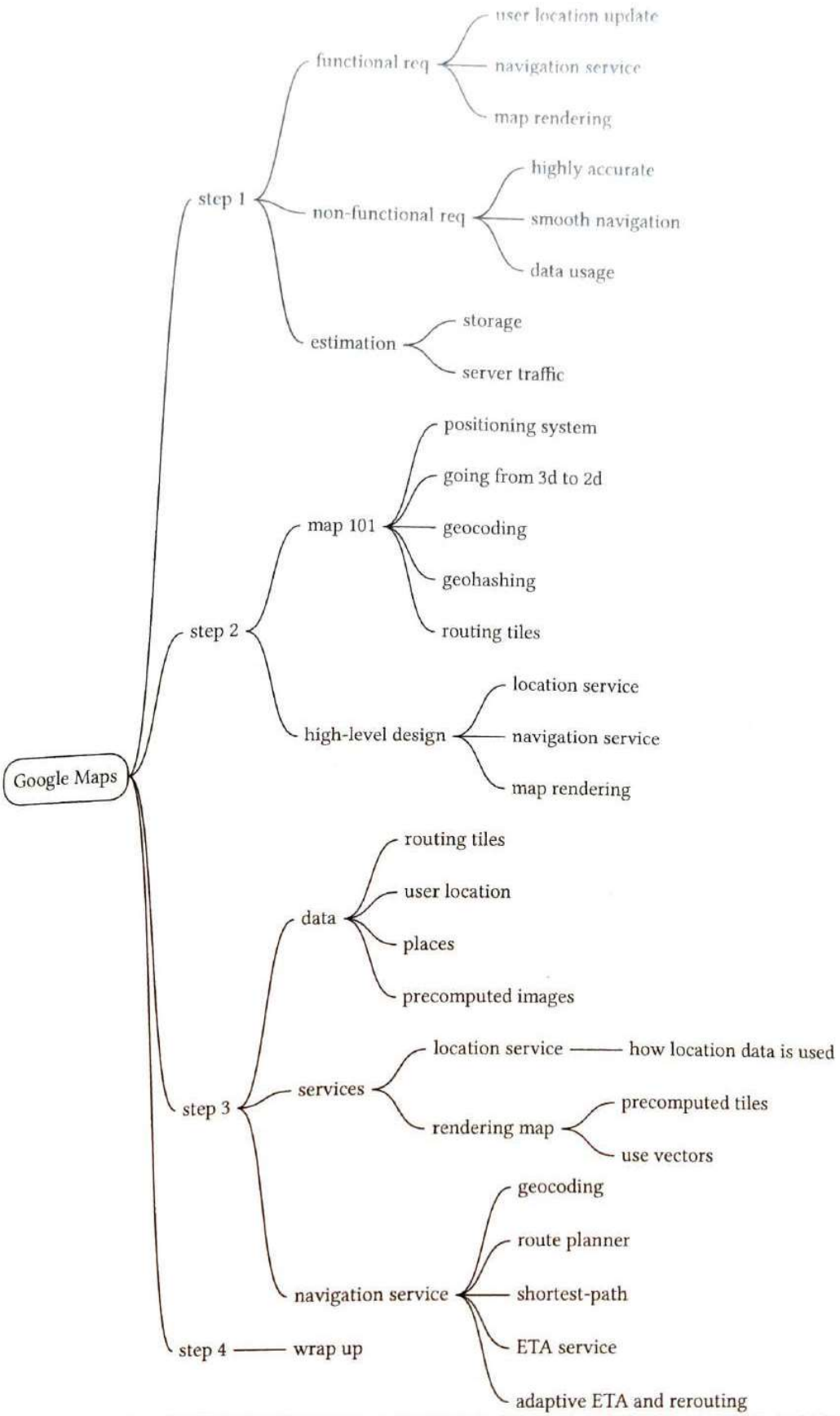
Figure 3.21: Final design

Step 4 - Wrap Up

In this chapter, we designed a simplified Google Maps application with key features such as location update, ETAs, route planning, and map rendering. If you are interested in expanding the system, one potential improvement would be to provide multi-stop navigation capability for enterprise customers. For example, for a given set of destinations, we have to find the optimal order in which to visit them all and provide proper navigation, based on live traffic conditions. This could be helpful for delivery services such as DoorDash, Uber, Lyft, etc.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Google Maps. https://developers.google.com/maps?hl=en_US.

[2] Google Maps Platform. https://cloud.google.com/maps-platform/.

[3] Prototyping a Smoother Map. https://medium.com/google-design/google-maps-cb0326d165f5.

[4] Mercator projection. https://en.wikipedia.org/wiki/Mercator_projection.

[5] Peirce quincuncial projection. https://en.wikipedia.org/wiki/Peirce_quincuncial_projection.

[6] Gall–Peters projection. https://en.wikipedia.org/wiki/Gall–Peters_projection.

[7] Winkel tripel projection. https://en.wikipedia.org/wiki/Winkel_tripel_projection.

[8] Address geocoding. https://en.wikipedia.org/wiki/Address_geocoding.

[9] Geohashing. https://kousiknath.medium.com/system-design-design-a-geo-spatial-index-for-real-time-location-search-10968fe62b9c.

[10] HTTP keep-alive. https://en.wikipedia.org/wiki/HTTP_persistent_connection.

[11] Directions API. https://developers.google.com/maps/documentation/directions/start?hl=en_US.

[12] Adjacency list. https://en.wikipedia.org/wiki/Adjacency_list.

[13] CAP theorem. https://en.wikipedia.org/wiki/CAP_theorem.

[14] Routing Tiles. https://valhalla.readthedocs.io/en/latest/mjolnir/why_tiles/.

[15] ETAs with GNNs. https://deepmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks.

[16] Google Maps 101: How AI helps predict traffic and determine routes. https://blog.google/products/maps/google-maps-101-how-ai-helps-predict-traffic-and-determine-routes/.

4 Distributed Message Queue

In this chapter, we explore a popular question in system design interviews: design a distributed message queue. In modern architecture, systems are broken up into small and independent building blocks with well-defined interfaces between them. Message queues provide communication and coordination for those building blocks. What benefits do message queues bring?

- Decoupling. Message queues eliminate the tight coupling between components so they can be updated independently.
- Improved scalability. We can scale producers and consumers independently based on traffic load. For example, during peak hours, more consumers can be added to handle the increased traffic.
- Increased availability. If one part of the system goes offline, the other components can continue to interact with the queue.
- Better performance. Message queues make asynchronous communication easy. Producers can add messages to a queue without waiting for the response and consumers consume messages whenever they are available. They don't need to wait for each other.

Figure 4.1 shows some of the most popular distributed message queues on the market.

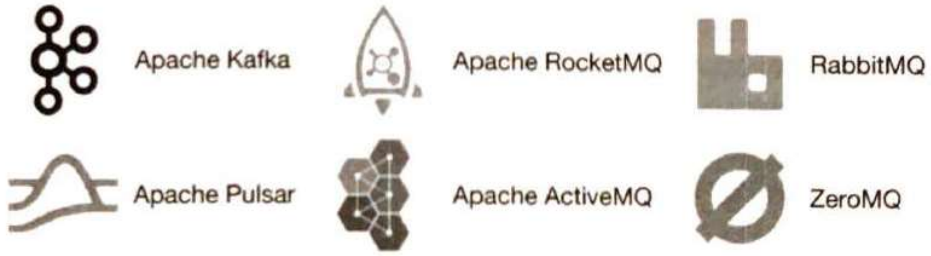


Figure 4.1: Popular distributed message queues

Message queues vs event streaming platforms

Strictly speaking, Apache Kafka and Pulsar are not message queues as they are event streaming platforms. However, there is a convergence of features that starts to blur the distinction between message queues (RocketMQ, ActiveMQ, RabbitMQ, ZeroMQ, etc.) and event streaming platforms (Kafka, Pulsar). For example, RabbitMQ, which is a typical message queue, added an optional streams feature to allow repeated message consumption and long message retention, and its implementation uses an append-only log, much like an event streaming platform would. Apache Pulsar is primarily a Kafka competitor, but it is also flexible and performant enough to be used as a typical distributed message queue.

In this chapter, we will design a distributed message queue with **additional features, such as long data retention, repeated consumption of messages, etc.**, that are typically only available on event streaming platforms. These additional features make the design more complicated. Throughout the chapter, we will highlight places where the design could be simplified if the focus of your interview centers around the more traditional distributed message queues.

Step 1 - Understand the Problem and Establish Design Scope

In a nutshell, the basic functionality of a message queue is straightforward: producers send messages to a queue, and consumers consume messages from it. Beyond this basic functionality, there are other considerations including performance, message delivery semantics, data detention, etc. The following set of questions will help clarify requirements and narrow down the scope.

Candidate: What's the format and average size of messages? Is it text only? Is multimedia allowed?
Interviewer: Text messages only. Messages are generally measured in the range of kilobytes (KBs).

Candidate: Can messages be repeatedly consumed?
Interviewer: Yes, messages can be repeatedly consumed by different consumers. Note that this is an added feature. A traditional distributed message queue does not retain a message once it has been successfully delivered to a consumer. Therefore, a message cannot be repeatedly consumed in a traditional message queue.

Candidate: Are messages consumed in the same order they were produced?
Interviewer: Yes, messages should be consumed in the same order they were produced. Note that this is an added feature. A traditional distributed message queue does not usually guarantee delivery orders.

Candidate: Does data need to be persisted and what is the data retention?
Interviewer: Yes, let's assume data retention is two weeks. This is an added feature. A traditional distributed message queue does not retain messages.

Candidate: How many producers and consumers are we going to support?

Interviewer: The more the better.

Candidate: What's the data delivery semantic we need to support? For example, at-most-once, at-least-once, and exactly once.
Interviewer: We definitely want to support at-least-once. Ideally, we should support all of them and make them configurable.

Candidate: What's the target throughput and end-to-end latency?
Interviewer: It should support high throughput for use cases like log aggregation. It should also support low latency delivery for more traditional message queue use cases.

With the above conversation, let's assume we have the following functional requirements:

- Producers send messages to a message queue.
- Consumers consume messages from a message queue.
- Messages can be consumed repeatedly or only once.
- Historical data can be truncated.
- Message size is in the kilobyte range.
- Ability to deliver messages to consumers in the order they were added to the queue.
- Data delivery semantics (at-least once, at-most once, or exactly once) can be configured by users.

Non-functional requirements

- High throughput or low latency, configurable based on use cases.
- Scalable. The system should be distributed in nature. It should be able to support a sudden surge in message volume.
- Persistent and durable. Data should be persisted on disk and replicated across multiple nodes.

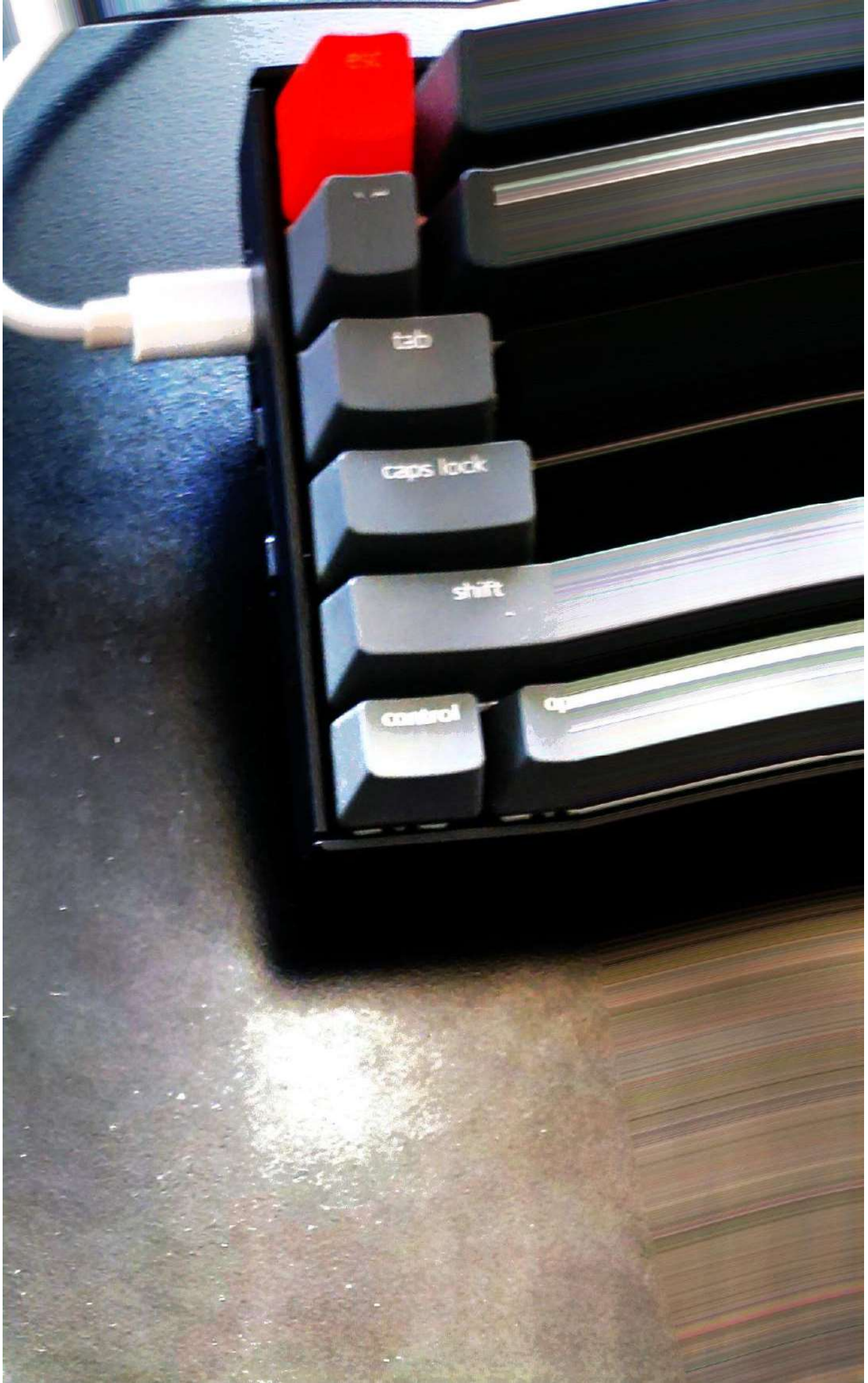
Adjustments for traditional message queues

Traditional message queues like RabbitMQ do not have as strong a retention requirement as event streaming platforms. Traditional queues retain messages in memory just long enough for them to be consumed. They provide on-disk overflow capacity [1] which is several orders of magnitude smaller than the capacity required for event streaming platforms. Traditional message queues do not typically maintain message ordering. The messages can be consumed in a different order than they were produced. These differences greatly simplify the design which we will discuss where appropriate.

Step 2 - Propose High-level Design and Get Buy-in

First, let's discuss the basic functionalities of a message queue.

Figure 4.2 shows the key components of a message queue and the simplified interactions between these components.


tab
caps lock
shift
control

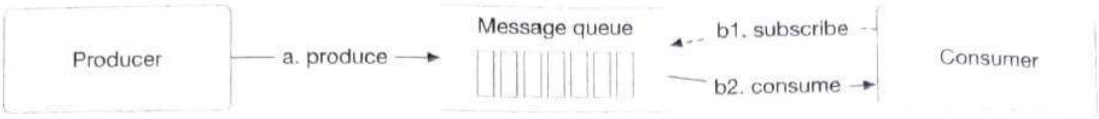


Figure 4.2: Key components in a message queue

- Producer sends messages to a message queue.
- Consumer subscribes to a queue and consumes the subscribed messages.
- Message queue is a service in the middle that decouples the producers from the consumers, allowing each of them to operate and scale independently.
- Both producer and consumer are clients in the client/server model, while the message queue is the server. The clients and servers communicate over the network.

Messaging models

The most popular messaging models are point-to-point and publish-subscribe.

Point-to-point

This model is commonly found in traditional message queues. In a point-to-point model, a message is sent to a queue and consumed by one and only one consumer. There can be multiple consumers waiting to consume messages in the queue, but each message can only be consumed by a single consumer. In Figure 4.3, message A is only consumed by consumer 1.

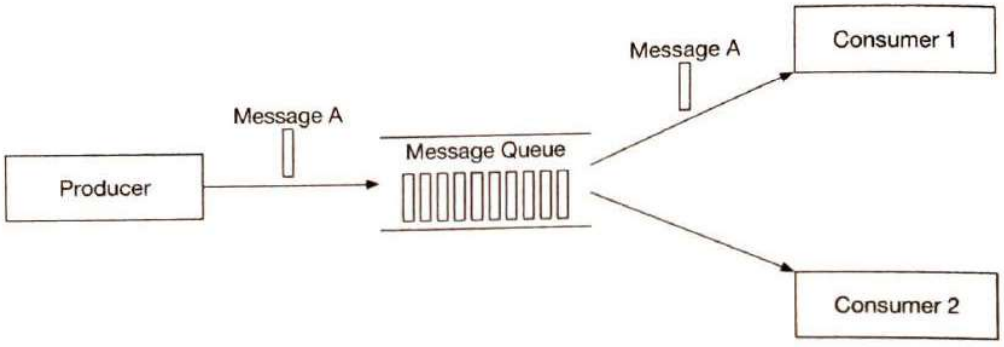


Figure 4.3: Point-to-point model

Once the consumer acknowledges that a message is consumed, it is removed from the queue. There is no data retention in the point-to-point model. In contrast, our design includes a persistence layer that keeps the messages for two weeks, which allows messages to be repeatedly consumed.

While our design could simulate a point-to-point model, its capabilities map more naturally to the publish-subscribe model.

Publish-subscribe

First, let's introduce a new concept, the topic. Topics are the categories used to organize messages. Each topic has a name that is unique across the entire message queue service.

Messages are sent to and read from a specific topic.

In the publish-subscribe model, a message is sent to a topic and received by the consumers subscribing to this topic. As shown in Figure 4.4, message A is consumed by both consumer 1 and consumer 2.

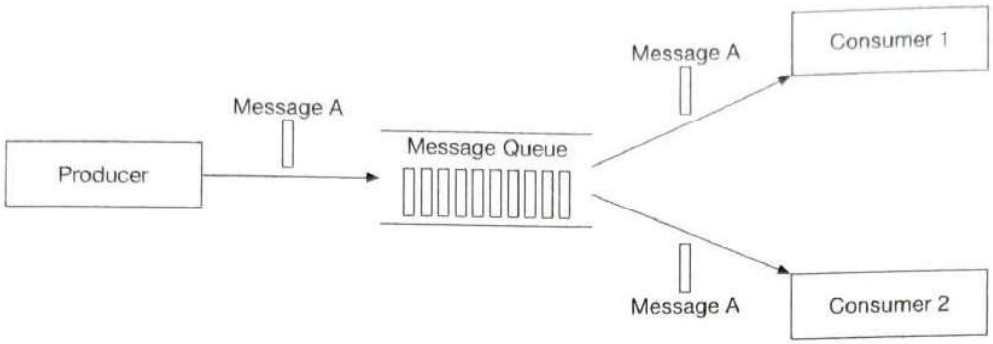


Figure 4.4: Publish-subscribe model

Our distributed message queue supports both models. The publish-subscribe model is implemented by **topics**, and the point-to-point model can be simulated by the concept of the **consumer group**, which will be introduced in the consumer group section.

Topics, partitions, and brokers

As mentioned earlier, messages are persisted by topics. What if the data volume in a topic is too large for a single server to handle?

One approach to solve this problem is called **partition (sharding)**. As Figure 4.5 shows, we divide a topic into partitions and deliver messages evenly across partitions. Think of a partition as a small subset of the messages for a topic. Partitions are evenly distributed across the servers in the message queue cluster. These servers that hold partitions are called **brokers**. The distribution of partitions among brokers is the key element to support high scalability. We can scale the topic capacity by expanding the number of partitions.

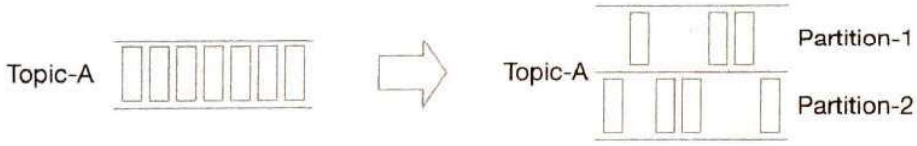


Figure 4.5: Partitions

Each topic partition operates in the form of a queue with the FIFO (first in, first out) mechanism. This means we can keep the order of messages inside a partition. The position of a message in the partition is called an **offset**.

When a message is sent by a producer, it is actually sent to one of the partitions for the topic. Each message has an optional message key (for example, a user's ID), and all messages for the same message key are sent to the same partition. If the message key is absent, the message is randomly sent to one of the partitions.

When a consumer subscribes to a topic, it pulls data from one or more of these partitions. When there are multiple consumers subscribing to a topic, each consumer is responsible for a subset of the partitions for the topic. The consumers form a **consumer group** for a topic.

The message queue cluster with brokers and partitions is represented in Figure 4.6.

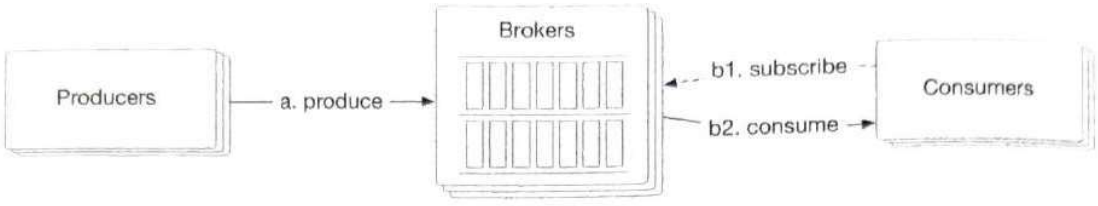


Figure 4.6: Message queue cluster

Consumer group

As mentioned earlier, we need to support both point-to-point and subscribe-publish models. **A consumer group** is a set of consumers, working together to consume messages from topics.

Consumers can be organized into groups. Each consumer group can subscribe to multiple topics and maintain its own consuming offsets. For example, we can group consumers by use cases, one group for billing and the other for accounting.

The instances in the same group can consume traffic in parallel, as in Figure 4.7.

- Consumer group 1 subscribes to topic A.
- Consumer group 2 subscribes to both topics A and B.
- Topic A is subscribed by both consumer groups-1 and group-2, which means the same message is consumed by multiple consumers. This pattern supports the subscribe/publish model.

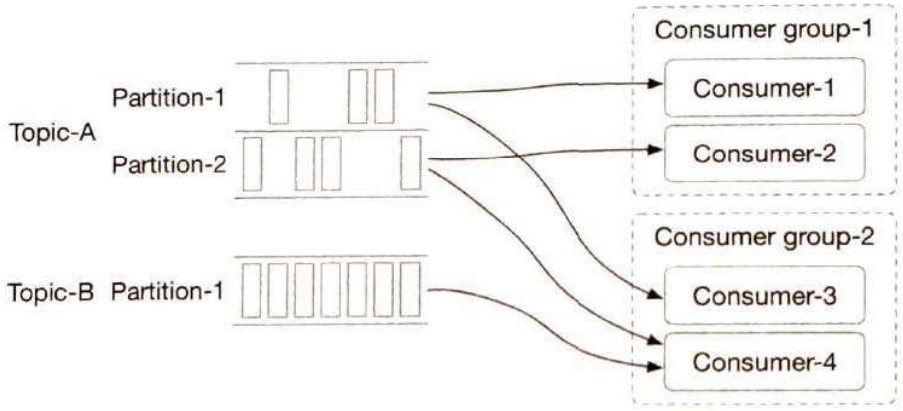


Figure 4.7: Consumer groups

However, there is one problem. Reading data in parallel improves the throughput, but

the consumption order of messages in the same partition cannot be guaranteed. For example, if Consumer-1 and Consumer-2 both read from Partition-1, we will not be able to guarantee the message consumption order in Partition-1.

The good news is we can fix this by adding a constraint, that a single partition can only be consumed by one consumer in the same group. If the number of consumers of a group is larger than the number of partitions of a topic, some consumers will not get data from this topic. For example, in Figure 4.7, Consumer-3 in Consumer group-2 cannot consume messages from topic B because it is consumed by Consumer-4 in the same consumer group, already.

With this constraint, if we put all consumers in the same consumer group, then messages in the same partition are consumed by only one consumer, which is equivalent to the point-to-point model. Since a partition is the smallest storage unit, we can allocate enough partitions in advance to avoid the need to dynamically increase the number of partitions. To handle high scale, we just need to add consumers.

High-level architecture

Figure 4.8 shows the updated high-level design.

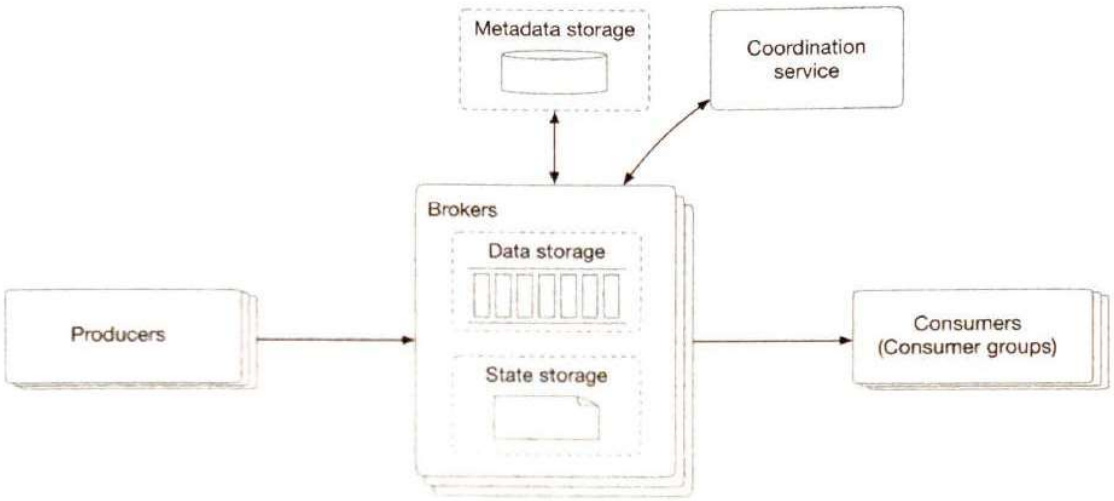


Figure 4.8: High-level design

Clients

- Producer: pushes messages to specific topics.
- Consumer group: subscribes to topics and consumes messages.

Core service and storage

- Broker: holds multiple partitions. A partition holds a subset of messages for a topic.
- Storage:
    - Data storage: messages are persisted in data storage in partitions.

- State storage: consumer states are managed by state storage.
- Metadata storage: configuration and properties of topics are persisted in metadata storage.

- Coordination service:
  - Service discovery: which brokers are alive.
  - Leader election: one of the brokers is selected as the active controller. There is only one active controller in the cluster. The active controller is responsible for assigning partitions.
  - Apache ZooKeeper [2] or etcd [3] are commonly used to elect a controller.

Step 3 - Design Deep Dive

To achieve high throughput while satisfying the high data retention requirement, we made three important design choices, which we explain in detail now.

- We chose an on-disk data structure that takes advantage of the great sequential access performance of rotational disks and the aggressive disk caching strategy of modern operating systems.
- We designed the message data structure to allow a message to be passed from the producer to the queue and finally to the consumer, with no modifications. This minimizes the need for copying which is very expensive in a high volume and high traffic system.
- We designed the system to favor batching. Small I/O is an enemy of high throughput. So, wherever possible, our design encourages batching. The producers send messages in batches. The message queue persists messages in even larger batches. The consumers fetch messages in batches when possible, too.

Data storage

Now let's explore the options to persist messages in more detail. In order to find the best choice, let's consider the traffic pattern of a message queue.

- Write-heavy, read-heavy.
- No update or delete operations. As a side note, a traditional message queue does not persist messages unless the queue falls behind, in which case there will be "delete" operations when the queue catches up. What we are talking about here is the persistence of a data streaming platform.
- Predominantly sequential read/write access.

Option 1: Database

The first option is to use a database.

- Relational database: create a topic table and write messages to the table as rows.

- NoSQL database: create a collection as a topic and write messages as documents.

Databases can handle the storage requirement, but they are not ideal because it is hard to design a database that supports both write-heavy and read-heavy access patterns at a large scale. The database solution does not fit our specific data usage patterns very well.

This means a database is not the best choice and could become a bottleneck of the system.

Option 2: Write-ahead log (WAL)

The second option is write-ahead log (WAL). WAL is just a plain file where new entries are appended to an append-only log. WAL is used in many systems, such as the redo log in MySQL [4] and the WAL in ZooKeeper.

We recommend persisting messages as WAL log files on disk. WAL has a pure sequential read/write access pattern. The disk performance of sequential access is very good [5]. Also, rotational disks have large capacity and they are pretty affordable.

As shown in Figure 4.9, a new message is appended to the tail of a partition, with a monotonically increasing offset. The easiest option is to use the line number of the log file as the offset. However, a file cannot grow infinitely, so it is a good idea to divide it into segments.

With segments, new messages are appended only to the active segment file. When the active segment reaches a certain size, a new active segment is created to receive new messages, and the currently active segment becomes inactive, like the rest of the non-active segments. Non-active segments only serve read requests. Old non-active segment files can be truncated if they exceed the retention or capacity limit.

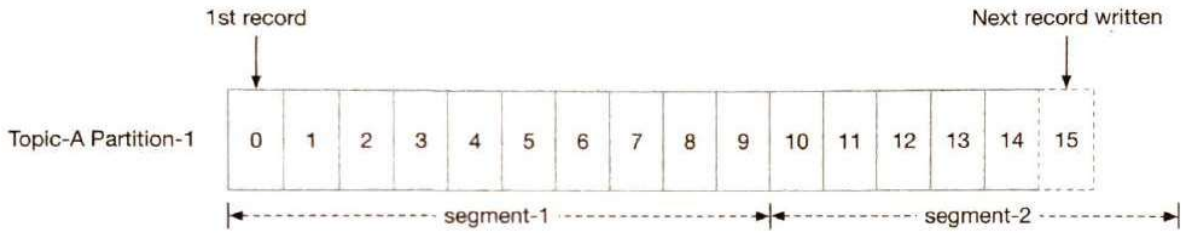


Figure 4.9: Append new messages

Segment files of the same partition are organized in a folder named `Partition-{:partition_id}`. The structure is shown in Figure 4.10.

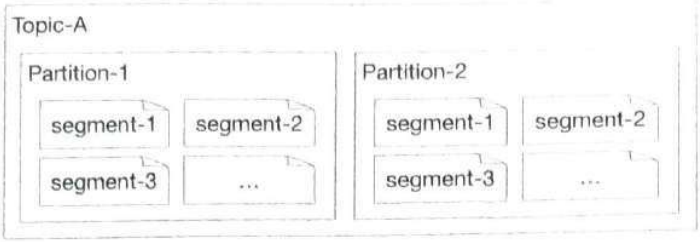


Figure 4.10: Data segment file distribution in topic partitions

A note on disk performance

To meet the high data retention requirement, our design relies heavily on disk drives to hold a large amount of data. There is a common misconception that rotational disks are slow, but this is really only the case for random access. For our workload, as long as we design our on-disk data structure to take advantage of the sequential access pattern, the modern disk drives in a RAID configuration (i.e., with disks striped together for higher performance) could comfortably achieve several hundred MB/sec of read and write speed. This is more than enough for our needs, and the cost structure is favorable.

Also, a modern operating system caches disk data in main memory very aggressively, so much so that it would happily use all available free memory to cache disk data. The WAL takes advantage of the heavy OS disk caching, too, as we described above.

Message data structure

The data structure of a message is key to high throughput. It defines the contract between the producers, message queue, and consumers. Our design achieves high performance by eliminating unnecessary data copying while the messages are in transit from the producers to the queue and finally to the consumers. If any parts of the system disagree on this contract, messages will need to be mutated which involves expensive copying. It could seriously hurt the performance of the system.

Below is a sample schema of the message data structure:

Field Name	Data Type
key	byte[]
value	byte[]
topic	string
partition	integer
offset	long
timestamp	long
size	integer
crc	integer

Table 4.1: Data schema of a message

Message key

The key of the message is used to determine the partition of the message. If the key is not defined, the partition is randomly chosen. Otherwise, the partition is chosen by `hash(key) % numPartitions`. If we need more flexibility, the producer can define its own mapping algorithm to choose partitions. Please note that the key is not equivalent to the partition number.

The key can be a string or a number. It usually carries some business information. The partition number is a concept in the message queue, which should not be explicitly exposed to clients.

With a proper mapping algorithm, if the number of partitions changes, messages can still be evenly sent to all the partitions.

Message value

The message value is the payload of a message. It can be plain text or a compressed binary block.

Reminder
The key and value of a message are different from the key-value pair in a key-value (KV) store. In the KV store, keys are unique, and we can find the value by key. In a message, keys do not need to be unique. Sometimes they are not even mandatory, and we don't need to find a value by key.

Other fields of a message

- Topic: the name of the topic that the message belongs to.
- Partition: the ID of the partition that the message belongs to.
- Offset: the position of the message in the partition. We can find a message via the combination of three fields: topic, partition, offset.
- Timestamp: the timestamp of when this message is stored.
- Size: the size of this message.
- CRC: Cyclic redundancy check (CRC) is used to ensure the integrity of raw data.

To support additional features, some optional fields can be added on demand. For example, messages can be filtered by tags, if tags are part of the optional fields.

Batching

Batching is pervasive in this design. We batch messages in the producer, the consumer, and the message queue itself. Batching is critical to the performance of the system. In this section, we focus primarily on batching in the message queue. We discuss batching for producer and consumer in more detail, shortly.

Batching is critical to improving performance because:

- It allows the operating system to group message together in a single network request and amortizes the cost of expensive network round trips
- The broker writes messages to the append logs in large chunks, which leads to larger blocks of sequential writes and larger contiguous blocks of disk cache, maintained by the operating system. Both lead to much greater sequential disk access throughput

There is a tradeoff between throughput and latency. If the system is deployed as a traditional message queue where latency might be more important, the system could be tuned to use a smaller batch size. Disk performance will suffer a little bit in this use case. If tuned for throughput, there might need to be a higher number of partitions per topic, to make up for the slower sequential disk write throughput.

So far, we've covered the main disk storage subsystem and its associated on-disk data structure. Now, let's switch gears and discuss the producer and consumer flows. Then we will come back and finish the deep dive into the rest of the message queue.

Producer flow

If a producer wants to send messages to a partition, which broker should it connect to? The first option is to introduce a routing layer. All messages sent to the routing layer are routed to the "correct" broker. If the brokers are replicated, the "correct" broker is the leader replica. We will cover replication later.

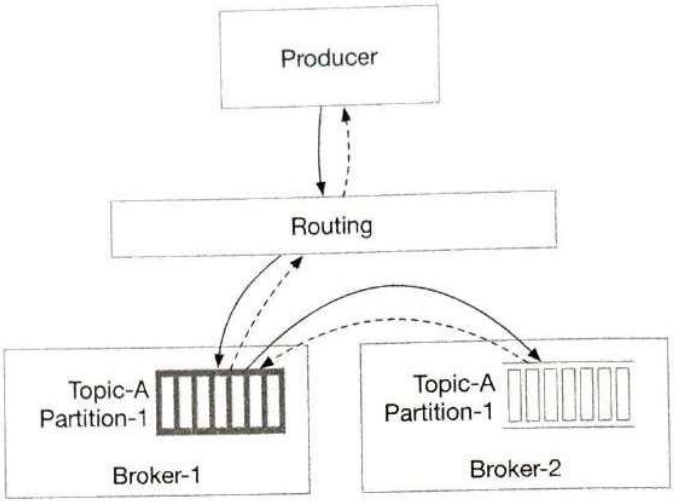


Figure 4.11: Routing layer

As shown in Figure 4.11, the producer tries to send messages to Partition-1 of Topic-A.

1. The producer sends messages to the routing layer.
2. The routing layer reads the replica distribution plan[1] from the metadata storage and caches it locally. When a message arrives, it routes the message to the leader replica of Partition-1, which is stored in Broker-1.

[1]The distribution of replicas for each partition is called a replica distribution plan

3. The leader replica receives the message and follower replicas pull data from the leader.
4. When "enough" replicas have synchronized the message, the leader commits the data (persisted on disk), which means the data can be consumed. Then it responds to the producer.

You might be wondering why we need both leader and follower replicas. The reason is fault tolerance. We dive deep into this process in the "In-sync replicas" section on page 113.

This approach works, but it has a few drawbacks:

- A new routing layer means additional network latency caused by overhead and additional network hops.
- Request batching is one of the big drivers of efficiency. This design doesn't take that into consideration.

Figure 4.12 shows the improved design.

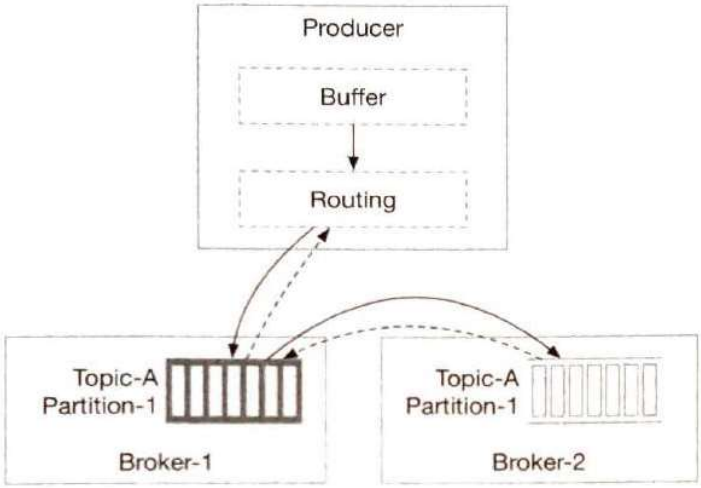


Figure 4.12: Producer with buffer and routing

The routing layer is wrapped into the producer and a buffer component is added to the producer. Both can be installed in the producer as part of the producer client library. This change brings several benefits:

- Fewer network hops mean lower latency.
- Producers can have their own logic to determine which partition the message should be sent to.
- Batching buffers messages in memory and sends out larger batches in a single request. This increases throughput.

The choice of the batch size is a classic tradeoff between throughput and latency (Figure

4.13). With a large batch size, the throughput increases but latency is higher, due to a longer wait time to accumulate the batch. With a small batch size, requests are sent sooner so the latency is lower, but throughput suffers. Producers can tune the batch size based on use cases.

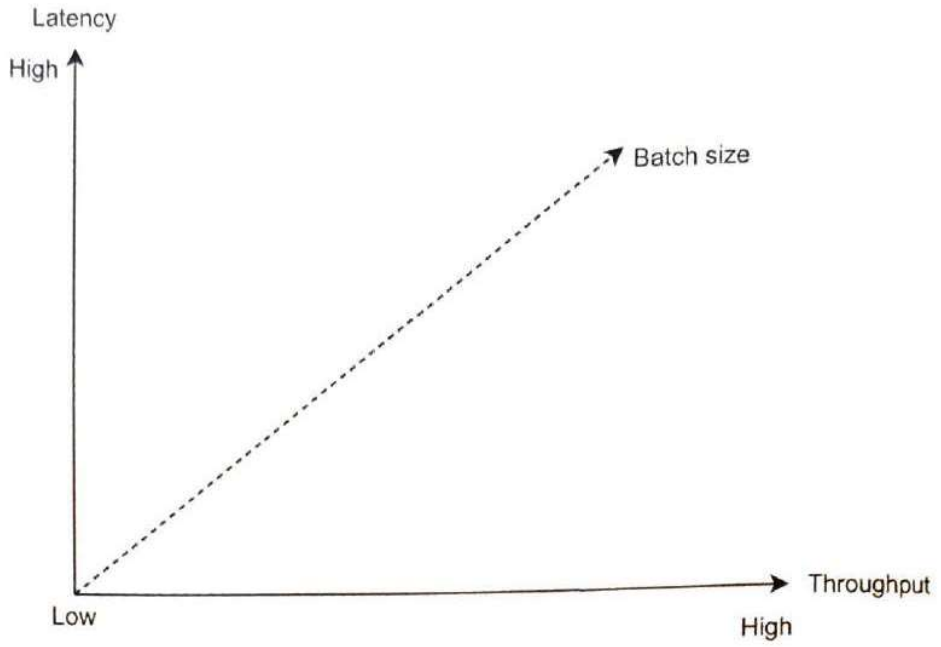


Figure 4.13: The choice of the batch size

Consumer flow

The consumer specifies its offset in a partition and receives back a chunk of events beginning from that position. An example is shown in Figure 4.14.

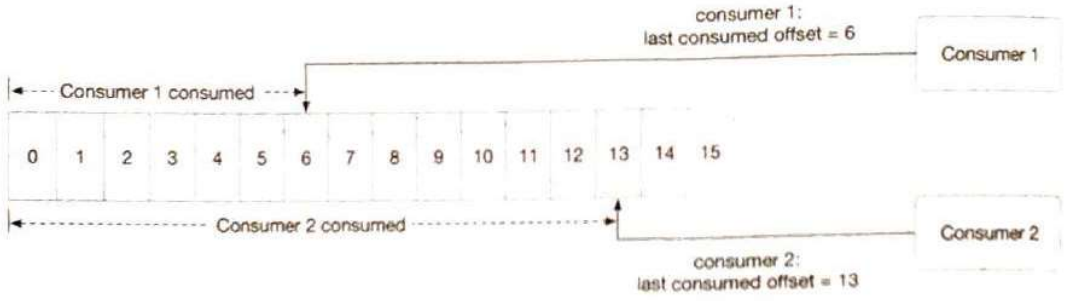


Figure 4.14: Consumer flow

Push vs pull

An important question to answer is whether brokers should push data to consumers, or if consumers should pull data from the brokers.

Push model

Pros:

- Low latency. The broker can push messages to the consumer immediately upon receiving them.

Cons:

- If the rate of consumption falls below the rate of production, consumers could be overwhelmed.
- It is difficult to deal with consumers with diverse processing power because the brokers control the rate at which data is transferred.

Pull model

Pros:

- Consumers control the consumption rate. We can have one set of consumers process messages in real-time and another set of consumers process messages in batch mode.
- If the rate of consumption falls below the rate of production, we can scale out the consumers, or simply catch up when it can.
- The pull model is more suitable for batch processing. In the push model, the broker has no knowledge of whether consumers will be able to process messages immediately. If the broker sends one message at a time to the consumer and the consumer is backed up, new messages will end up waiting in the buffer. A pull model pulls all available messages after the consumer's current position in the log (or up to the configurable max size). It is suitable for aggressive batching of data.

Cons:

- When there is no message in the broker, a consumer might still keep pulling data, wasting resources. To overcome this issue, many message queues support long polling mode, which allows pulls to wait a specified amount of time for new messages [6].

Based on these considerations, most message queues choose the pull model.

Figure 4.15 shows the workflow of the consumer pull model.

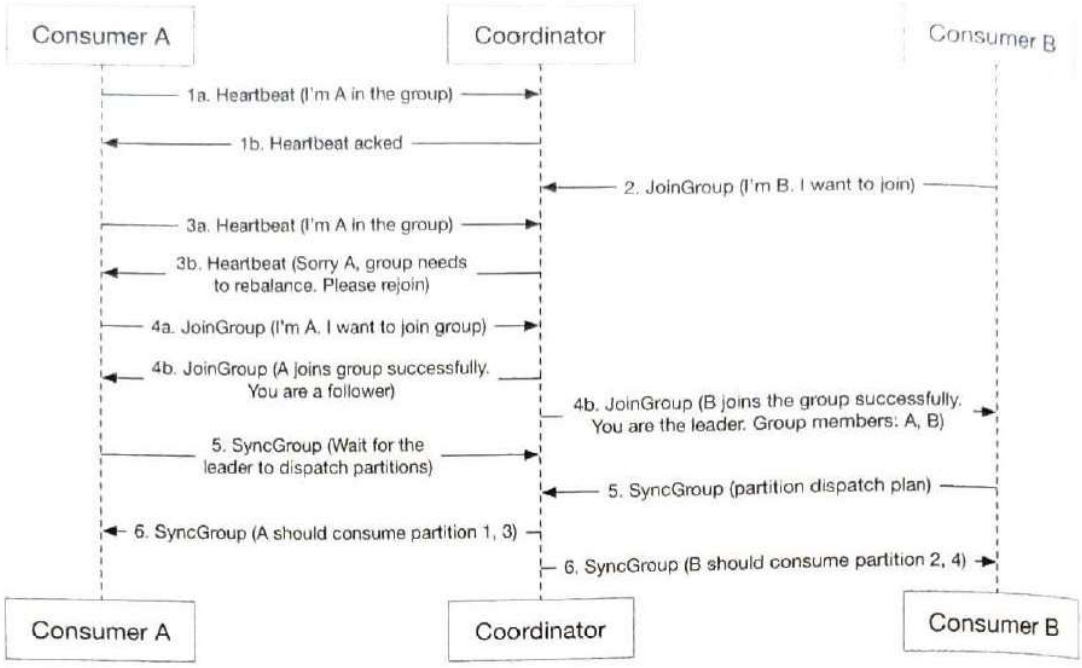


Figure 4.18: New consumer joins

1. Initially, only Consumer A is in the group. It consumes all the partitions and keeps the heartbeat with the coordinator.
2. Consumer B sends a request to join the group.
3. The coordinator knows it's time to rebalance, so it notifies all the consumers in the group in a passive way. When Consumer A's heartbeat is received by the coordinator, it asks Consumer A to rejoin the group.
4. Once all the consumers have rejoined the group, the coordinator chooses one of them as the leader and informs all the consumers about the election result.
5. The leader consumer generates the partition dispatch plan and sends it to the coordinator. Follower consumers ask the coordinator about the partition dispatch plan.
6. Consumers start consuming messages from newly assigned partitions.

Figure 4.19 shows the flow when an existing Consumer A leaves the group.

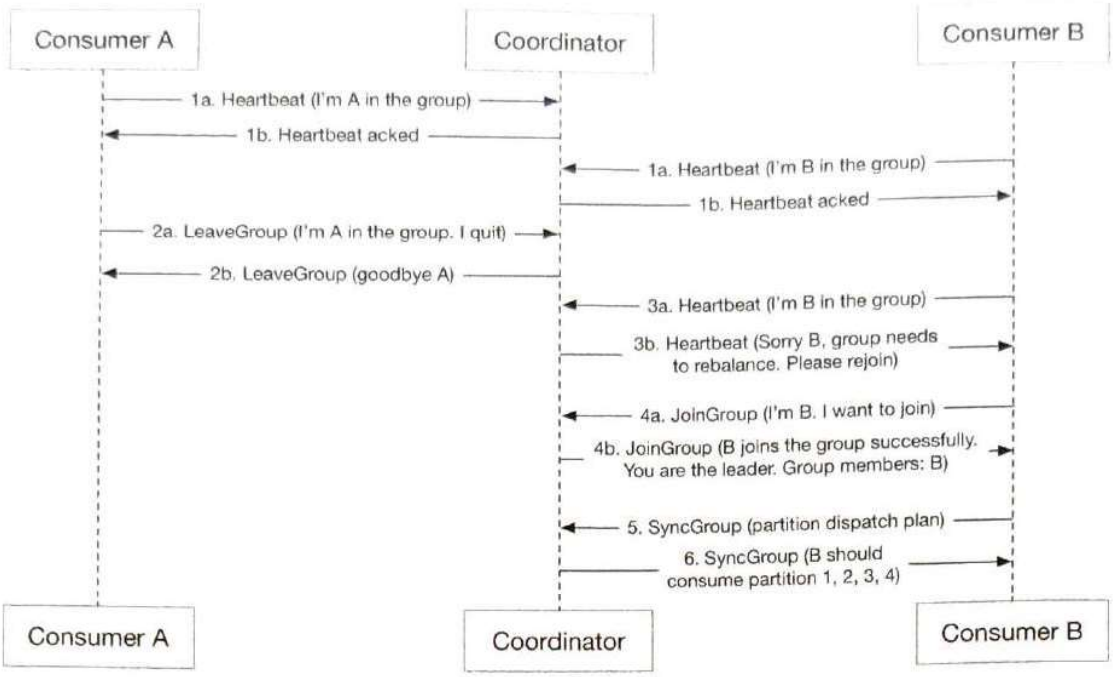


Figure 4.19: Existing consumer leaves

1. Consumer A and B are in the same consumer group.
2. Consumer A needs to be shut down, so it requests to leave the group.
3. The coordinator knows it's time to rebalance. When Consumer B's heartbeat is received by the coordinator, it asks Consumer B to rejoin the group.
4. The remaining steps are the same as the ones shown in Figure 4.18.

Figure 4.20 shows the flow when an existing Consumer A crashes.

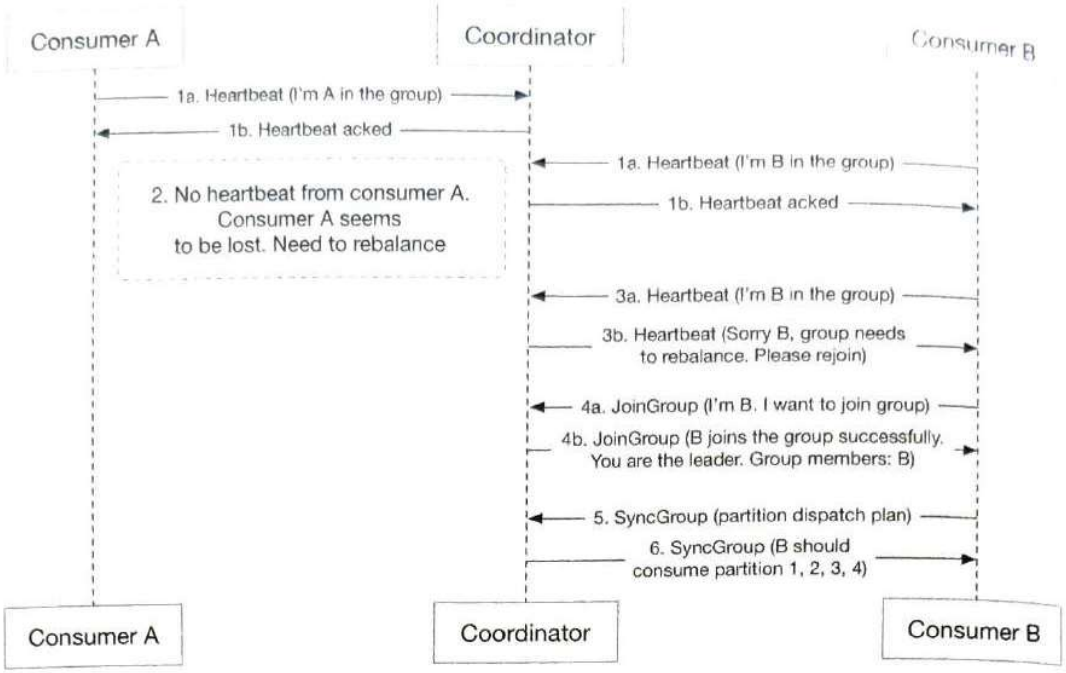


Figure 4.20: Existing consumer crashes

1. Consumer A and B keep heartbeats with the coordinator.
2. Consumer A crashes, so there is no heartbeat sent from Consumer A to the coordinator. Since the coordinator doesn't get any heartbeat signal within a specified amount of time from Consumer A, it marks the consumer as dead.
3. The coordinator triggers the rebalance process.
4. The following steps are the same as the ones in the previous scenario.

Now that we finished the detour on producer and consumer flows, let's come back and finish the deep dive on the rest of the message queue broker.

State storage

In the message queue broker, the state storage stores:

- The mapping between partitions and consumers.
- The last consumed offsets of consumer groups for each partition. As shown in Figure 4.21, the last consumed offset for consumer group-1 is 6 and the offset for consumer group-2 is 13.

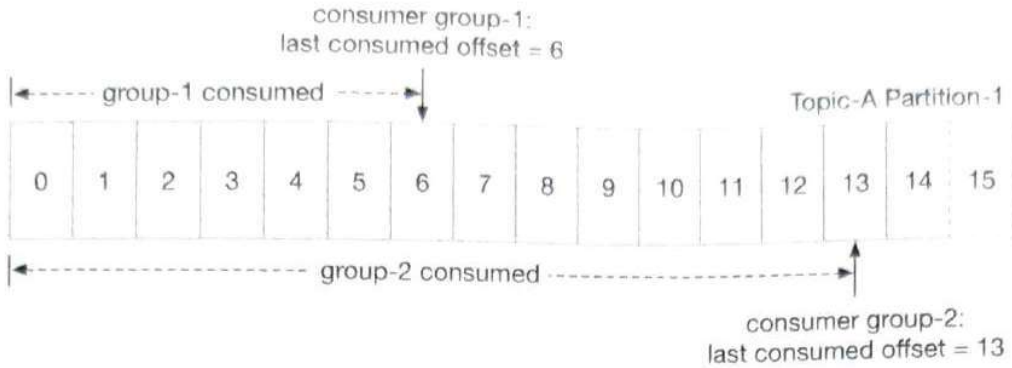


Figure 4.21: Last consumed offset of consumer groups

For example, as shown in Figure 4.21, a consumer in group-1 consumes messages from the partition in sequence and commits the consumed offset 6. This means all the messages before and at offset 6 are already consumed. If the consumer crashes, another new consumer in the same group will resume consumption by reading the last consumed offset from the state storage.

The data access patterns for consumer states are:

- Frequent read and write operations but the volume is not high.
- Data is updated frequently and is rarely deleted.
- Random read and write operations.
- Data consistency is important.

Lots of storage solutions can be used for storing the consumer state data. Considering the data consistency and fast read/write requirements, a KV store like ZooKeeper is a great choice. Kafka has moved the offset storage from ZooKeeper to Kafka brokers. Interested readers can read the reference material [8] to learn more.

Metadata storage

The metadata storage stores the configuration and properties of topics, including a number of partitions, retention period, and distribution of replicas.

Metadata does not change frequently and the data volume is small, but it has a high consistency requirement. ZooKeeper is a good choice for storing metadata.

ZooKeeper

By reading previous sections, you probably have already sensed that ZooKeeper is very helpful for designing a distributed message queue. If you are not familiar with it, ZooKeeper is an essential service for distributed systems offering a hierarchical key-value store. It is commonly used to provide a distributed configuration service, synchronization service, and naming registry [2].

ZooKeeper is used to simplify our design as shown in Figure 4.22.

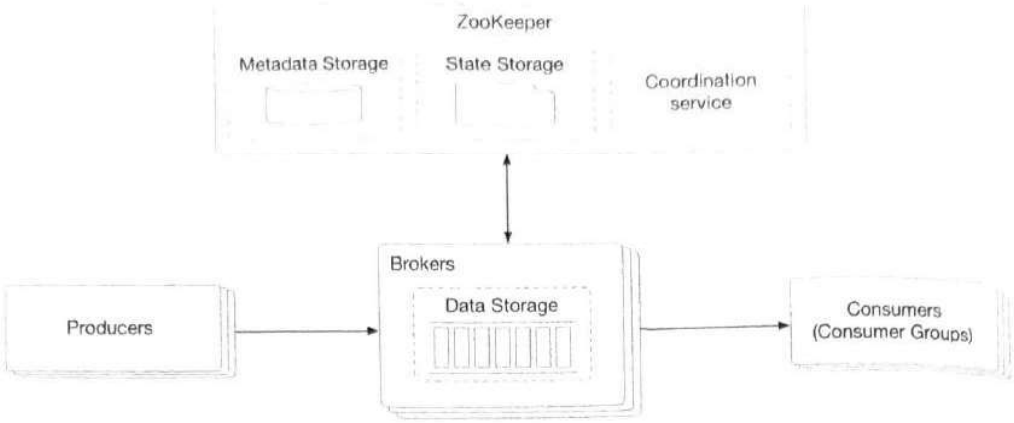


Figure 4.22: ZooKeeper

Let's briefly go over the change.

- Metadata and state storage are moved to ZooKeeper.
- The broker now only needs to maintain the data storage for messages.
- ZooKeeper helps with the leader election of the broker cluster.

Replication

In distributed systems, hardware issues are common and cannot be ignored. Data gets lost when a disk is damaged or fails permanently. Replication is the classic solution to achieve high availability.

As in Figure 4.23, each partition has 3 replicas, distributed across different broker nodes.

For each partition, the highlighted replicas are the leaders and the others are followers. Producers only send messages to the leader replica. The follower replicas keep pulling new messages from the leader. Once messages are synchronized to enough replicas, the leader returns an acknowledgment to the producer. We will go into detail about how to define "enough" in the In-sync Replicas section on page 113.

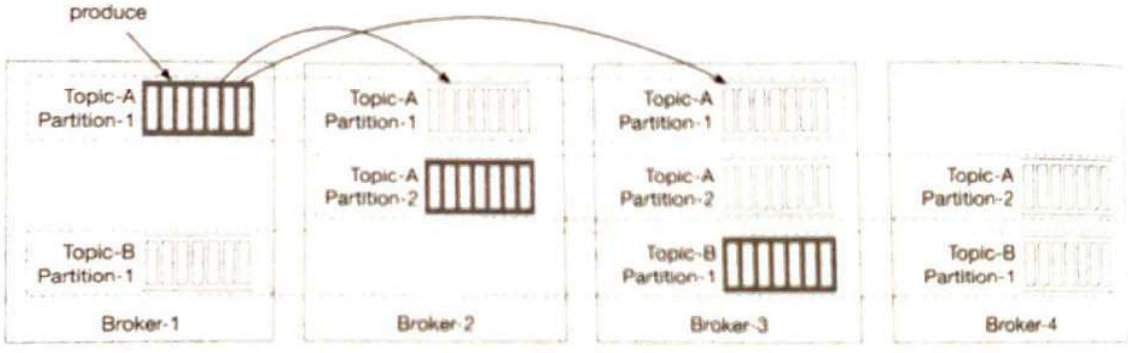


Figure 4.23: Replication

The distribution of replicas for each partition is called a replica distribution plan. For

example, the replica distribution plan in Figure 4.23 can be described as:

- Partition-1 of Topic-A: 3 replicas, leader in Broker-1, followers in Broker-2 and 3;
- Partition-2 of Topic-A: 3 replicas, leader in Broker-2, followers in Broker-3 and 4;
- Partition-1 of Topic-B: 3 replicas, leader in Broker-3, followers in Broker-4 and 1.

Who makes the replica distribution plan? It works as follows; with the help of the coordination service, one of the broker nodes is elected as the leader. It generates the replica distribution plan and persists the plan in metadata storage. All the brokers now can work according to the plan.

If you are interested in knowing more about replications, check out "Chapter 5. Replication" of the book "Design Data-Intensive Applications" [9].

In-sync replicas

We mentioned that messages are persisted in multiple partitions to avoid single node failure, and each partition has multiple replicas. Messages are only written to the leader, and followers synchronize data from the leader. One problem we need to solve is keeping them in sync.

In-sync replicas (ISR) refer to replicas that are "in-sync" with the leader. The definition of "in-sync" depends on the topic configuration. For example, if the value of replica.lag.max.messages is 4, it means that as long as the follower is behind the leader by no more than 3 messages, it will not be removed from ISR [10]. The leader is an ISR by default.

Let's use an example as shown in Figure 4.24 to shows how ISR works.

- The committed offset in the leader replica is 13. Two new messages are written to the leader, but not committed yet. Committed offset means that all messages before and at this offset are already synchronized to all the replicas in ISR.
- Replica-2 and replica-3 have fully caught up with the leader, so they are in ISR and can fetch new messages.
- Replica-4 did not fully catch up with the leader within the configured lag time, so it is not in ISR. When it catches up again, it can be added to ISR.

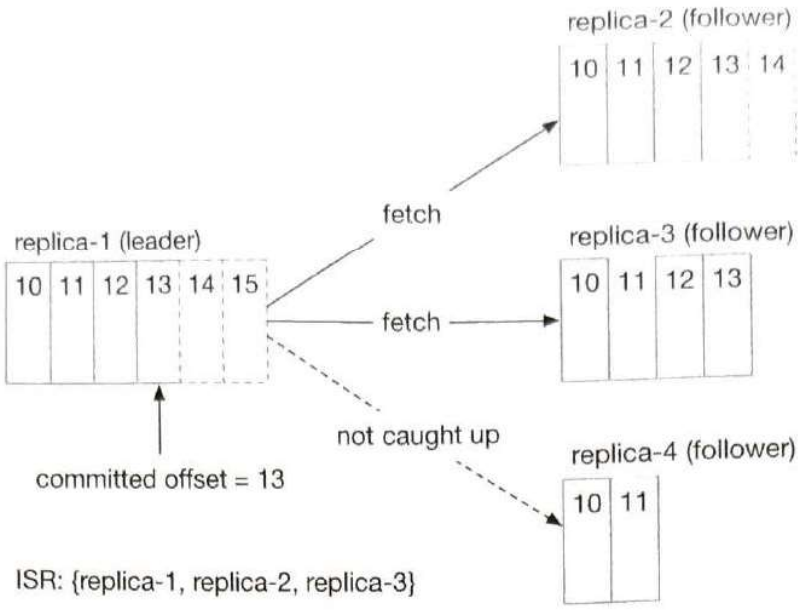


Figure 4.24: How ISR works

Why do we need ISR? The reason is that ISR reflects the trade-off between performance and durability. If producers don't want to lose any messages, the safest way to do that is to ensure all replicas are already in sync before sending an acknowledgment. But a slow replica will cause the whole partition to become slow or unavailable.

Now that we've discussed ISR, let's take a look at acknowledgment settings. Producers can choose to receive acknowledgments until the k number of ISRs has received the message, where k is configurable.

ACK=all

Figure 4.25 illustrates the case with ACK=all. With ACK=all, the producer gets an ACK when all ISRs have received the message. This means it takes a longer time to send a message because we need to wait for the slowest ISR, but it gives the strongest message durability.

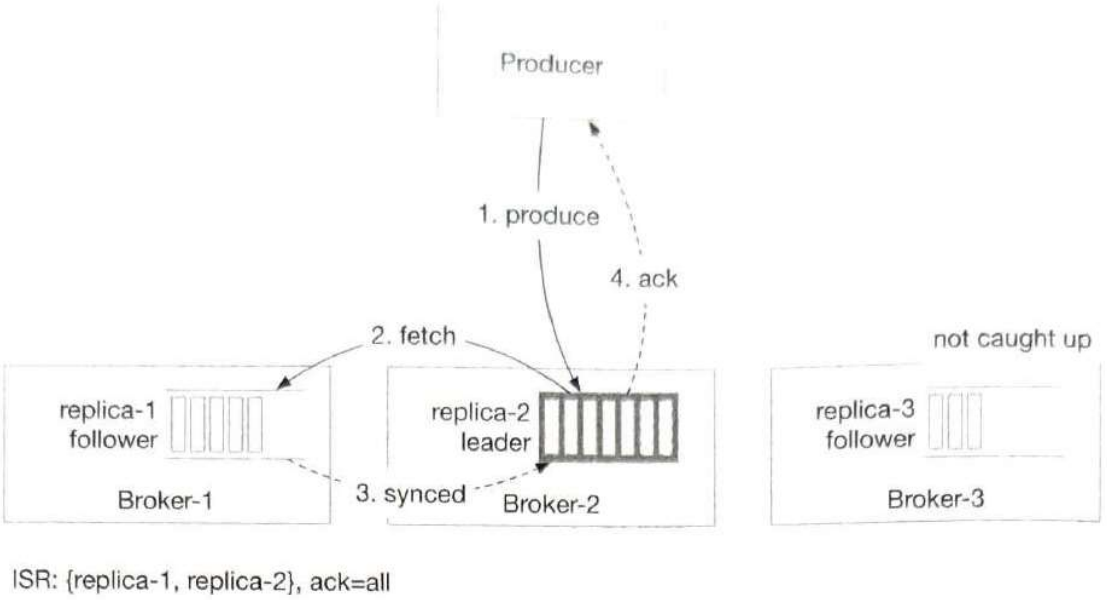


Figure 4.25: ACK=all

ACK=1

With ACK=1, the producer receives an ACK once the leader persists the message. The latency is improved by not waiting for data synchronization. If the leader fails immediately after a message is acknowledged but before it is replicated by follower nodes, then the message is lost. This setting is suitable for low latency systems where occasional data loss is acceptable.

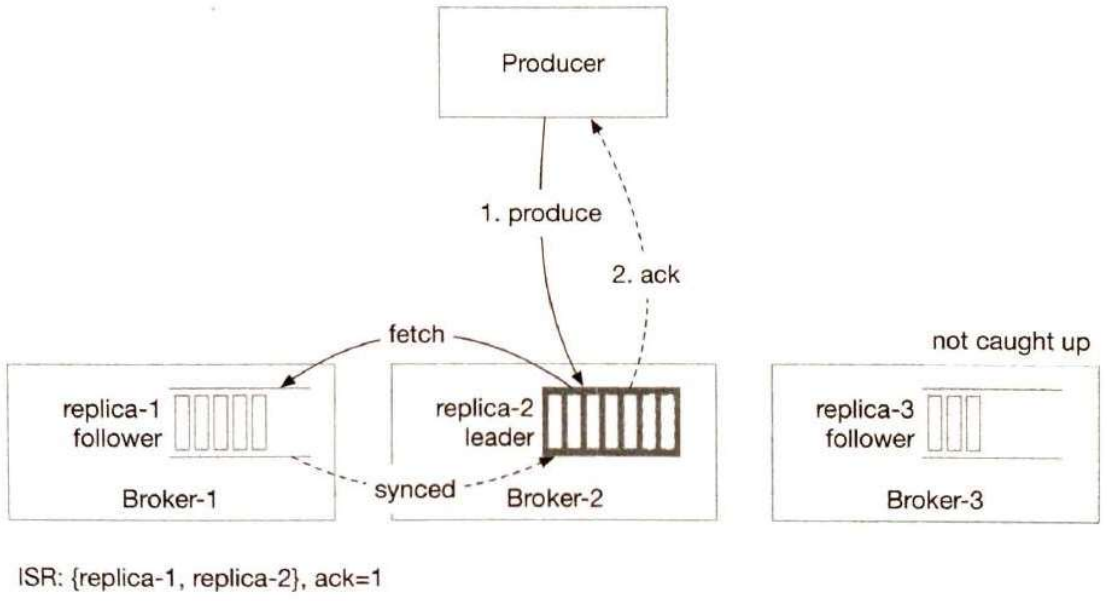


Figure 4.26: ACK=1

ACK=0

The producer keeps sending messages to the leader without waiting for any acknowledgment, and it never retries. This method provides the lowest latency at the cost of potential message loss. This setting might be good for use cases like collecting metrics or logging data since data volume is high and occasional data loss is acceptable.

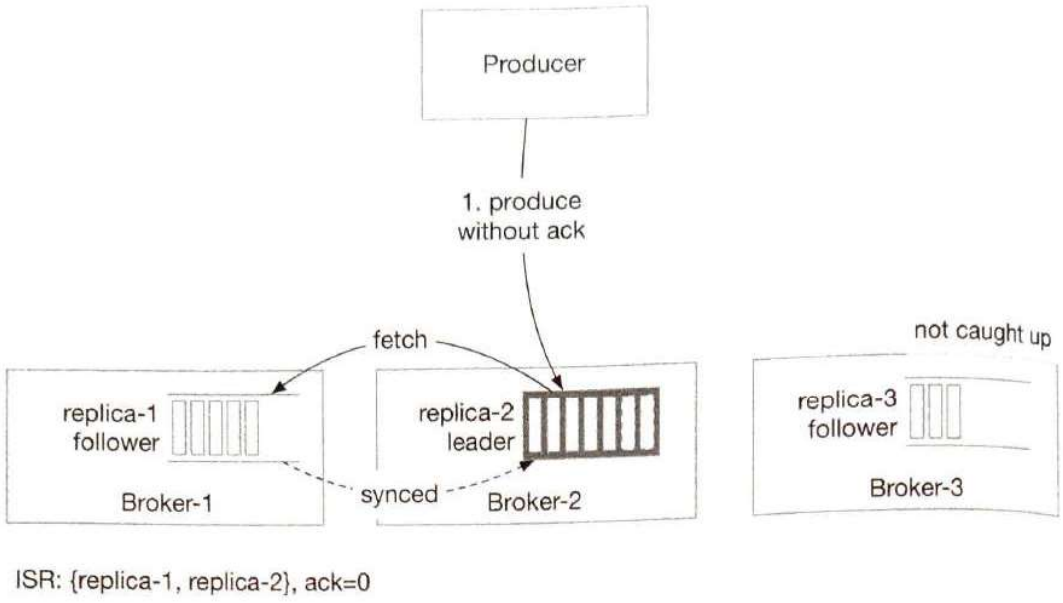


Figure 4.27: ACK=0

Configurable ACK allows us to trade durability for performance.

Now let's look at the consumer side. The easiest setup is to let consumers connect to a leader replica to consume messages.

You might be wondering if the leader replica would be overwhelmed by this design and why messages are not read from ISRs. The reasons are:

- Design and operational simplicity.
- Since messages in one partition are dispatched to only one consumer within a consumer group, this limits the number of connections to the leader replica.
- The number of connections to the leader replicas is usually not large as long as a topic is not super hot.
- If a topic is hot, we can scale by expanding the number of partitions and consumers.

In some scenarios, reading from the leader replica might not be the best option. For example, if a consumer is located in a different data center from the leader replica, the read performance suffers. In this case, it is worthwhile to enable consumers to read from the closest ISRs. Interested readers can check out the reference material about this [11].

ISR is very important. How does it determine if a replica is ISR or not? Usually, the leader

for every partition tracks the ISR list by computing the lag of every replica from itself. If you are interested in detailed algorithms, you can find the implementations in reference materials [12] [13].

Scalability

By now we have made great progress designing the distributed message queue system. In the next step, let's evaluate the scalability of different system components:

- Producers
- Consumers
- Brokers
- Partitions

Producer

The producer is conceptually much simpler than the consumer because it doesn't need group coordination. The scalability of producers can easily be achieved by adding or removing producer instances.

Consumer

Consumer groups are isolated from each other, so it is easy to add or remove a consumer group. Inside a consumer group, the rebalancing mechanism helps to handle the cases where a consumer gets added or removed, or when it crashes. With consumer groups and the rebalance mechanism, the scalability and fault tolerance of consumers can be achieved.

Broker

Before discussing scalability on the broker side, let's first consider the failure recovery of brokers.

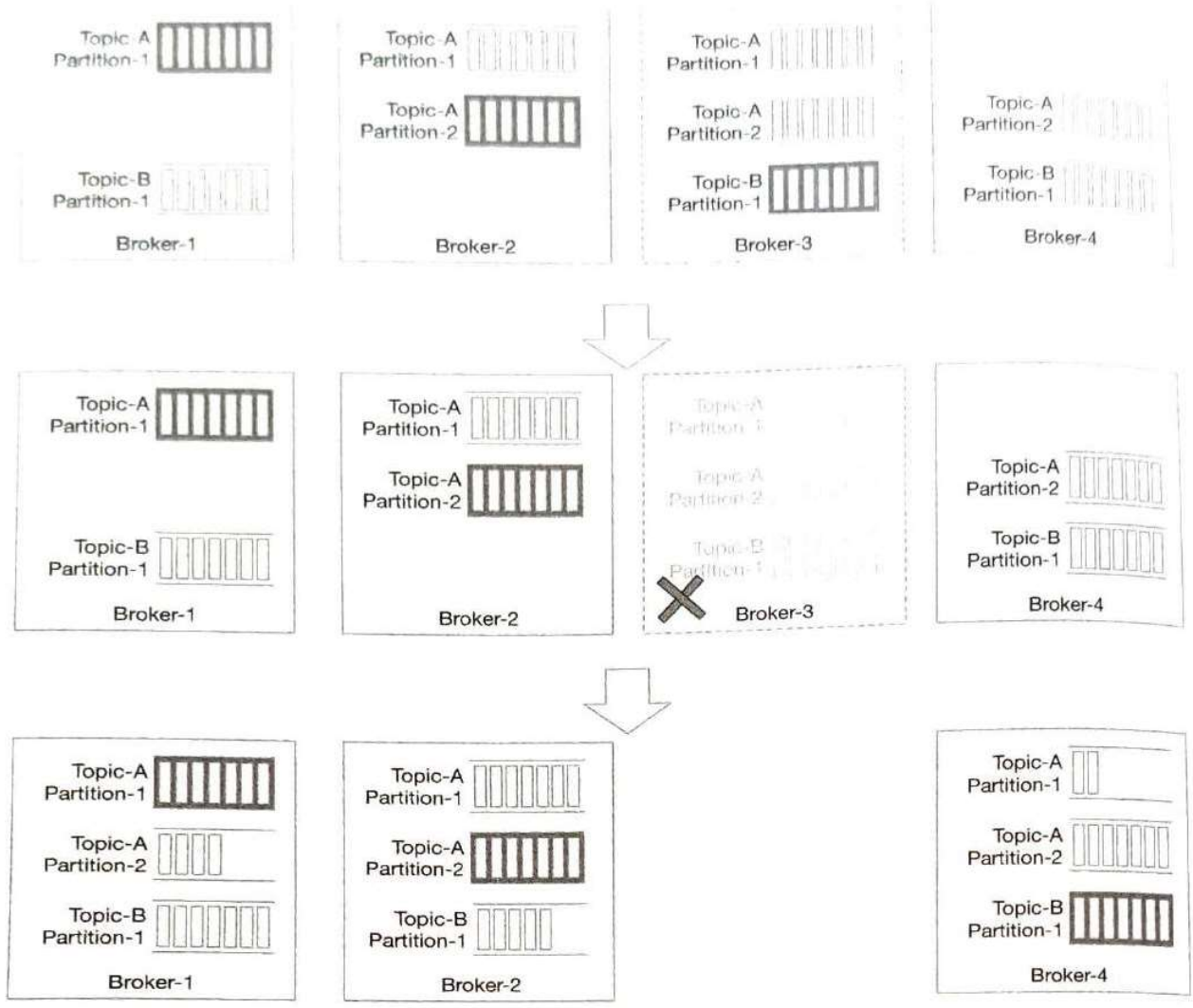


Figure 4.28: Broker node crashes

Let's use an example in Figure 4.28 to explain how failure recovery works.

1. Assume there are 4 brokers and the partition (replica) distribution plan is shown below:
    - Partition-1 of topic A: replicas in Broker-1 (leader), 2, and 3.
    - Partition-2 of topic A: replicas in Broker-2 (leader), 3, and 4.
    - Partition-1 of topic B: replicas in Broker-3 (leader), 4, and 1.
2. Broker-3 crashes, which means all the partitions on the node are lost. The partition distribution plan is changed to:
    - Partition-1 of topic A: replicas in Broker-1 (leader) and 2.
    - Partition-2 of topic A: replicas in Broker-2 (leader) and 4.
    - Partition-1 of topic B: replicas in Broker-4 and 1.

3. The broker controller detects Broker-3 is down and generates a new partition distribution plan for the remaining broker nodes:
    - Partition-1 of topic A: replicas in Broker-1 (leader), 2, and 4 (new).
    - Partition-2 of topic A: replicas in Broker-2 (leader), 4, and 1 (new).
    - Partition-1 of topic B: replicas in Broker-4 (leader), 1, and 2 (new).
4. The new replicas work as followers and catch up with the leader.

To make the broker fault-tolerant, here are additional considerations:

- The minimum number of ISRs specifies how many replicas the producer must receive before a message is considered to be successfully committed. The higher the number, the safer. But on the other hand, we need to balance latency and safety.
- If all replicas of a partition are in the same broker node, then we cannot tolerate the failure of this node. It is also a waste of resources to replicate data in the same node. Therefore, replicas should not be in the same node.
- If all the replicas of a partition crash, the data for that partition is lost forever. When choosing the number of replicas and replica locations, there's a trade-off between data safety, resource cost, and latency. It is safer to distribute replicas across data centers, but this will incur much more latency and cost, to synchronize data between replicas. As a workaround, data mirroring can help to copy data across data centers, but this is out of scope. The reference material [14] covers this topic.

Now let's get back to discussing the scalability of brokers. The simplest solution would be to redistribute the replicas when broker nodes are added or removed.

However, there is a better approach. The broker controller can temporarily allow more replicas in the system than the number of replicas in the config file. When the newly added broker catches up, we can remove the ones that are no longer needed. Let's use an example as shown in Figure 4.29 to understand the approach.

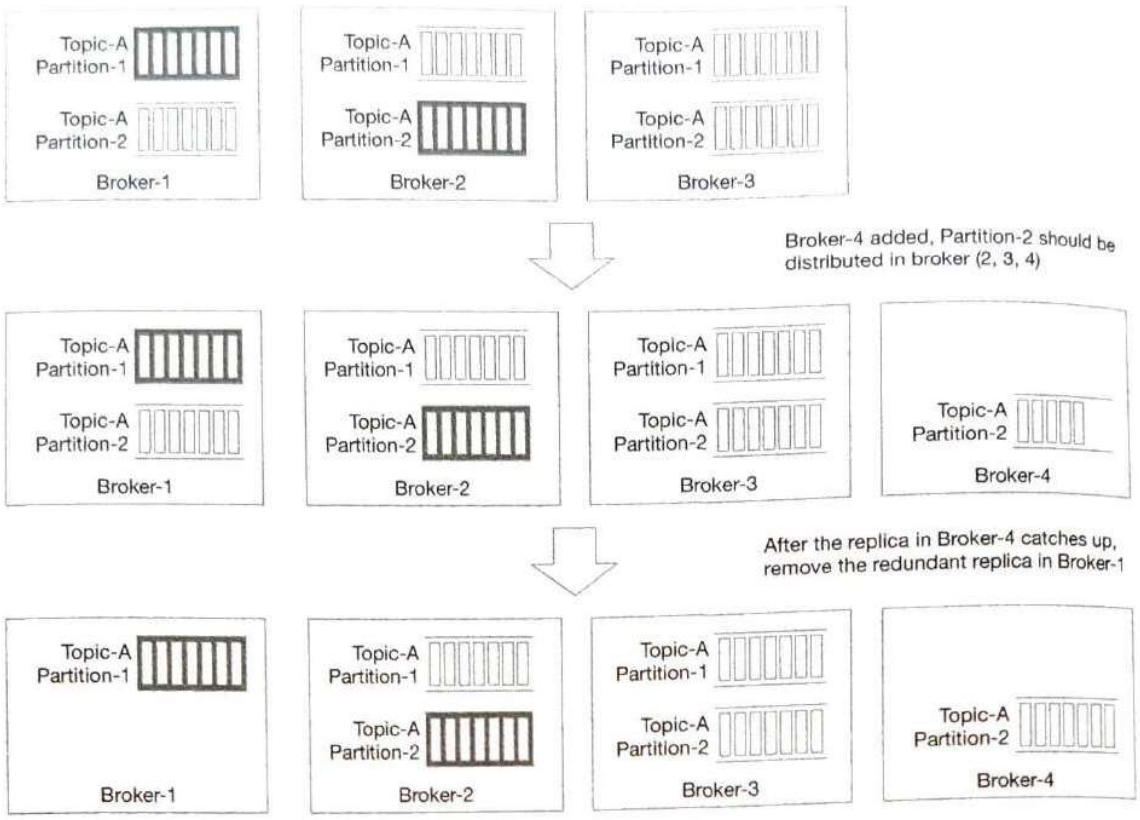


Figure 4.29: Add new broker node

1. The initial setup: 3 brokers, 2 partitions, and 3 replicas for each partition.
2. New Broker-4 is added. Assume the broker controller changes the replica distribution of Partition-2 to the broker (2, 3, 4). The new replica in Broker-4 starts to copy data from leader Broker-2. Now the number of replicas for Partition-2 is temporarily more than 3.
3. After the replica in Broker-4 catches up, the redundant partition in Broker-1 is gracefully removed.

By following this process, data loss while adding brokers can be avoided. A similar process can be applied to remove brokers safely.

Partition

For various operational reasons, such as scaling the topic, throughput tuning, balancing availability/ throughput, etc., we may change the number of partitions. When the number of partitions changes, the producer will be notified after it communicates with any broker, and the consumer will trigger consumer rebalancing. Therefore, it is safe for both the producer and consumer.

Now let's consider the data storage layer when the number of partitions changes. As in Figure 4.30, we have added a partition to the topic.

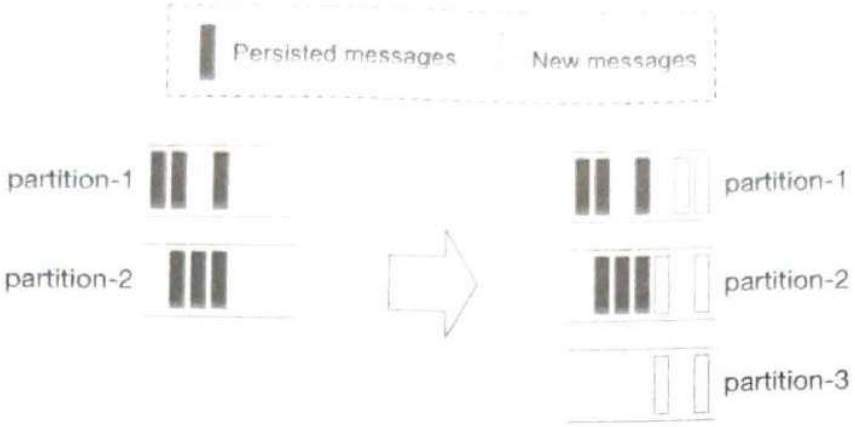


Figure 4.30: Partition increase

- Persisted messages are still in the old partitions, so there's no data migration.
- After the new partition (partition-3) is added, new messages will be persisted in all 3 partitions.

So it is straightforward to scale the topic by increasing partitions.

Decrease the number of partitions

Decreasing partitions is more complicated, as illustrated in Figure 4.31.

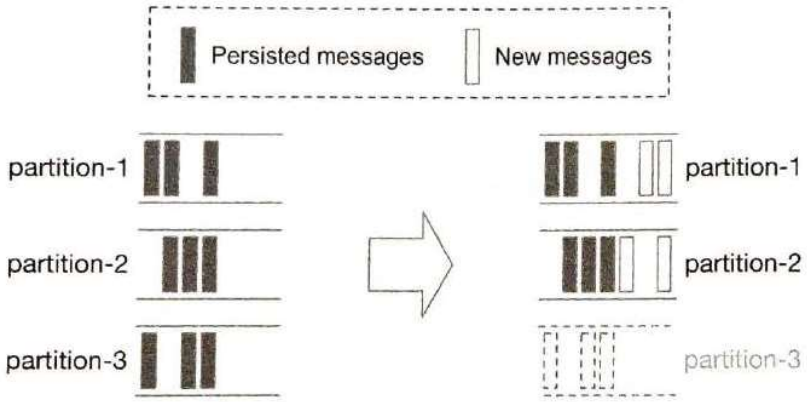


Figure 4.31: Partition decrease

- Partition-3 is decommissioned so new messages are only received by the remaining partitions (partition-1 and partition-2).
- The decommissioned partition (partition-3) cannot be removed immediately because data might be currently consumed by consumers for a certain amount of time. Only after the configured retention period passes, data can be truncated and storage space is freed up. Reducing partitions is not a shortcut to reclaiming data space.
- During this transitional period (while partition-3 is decommissioned), producers only send messages to the remaining 2 partitions, but consumers can still consume from all 3 partitions. After the retention period of the decommissioned partition expires,

consumer groups need rebalancing.

Data delivery semantics

Now that we understand the different components of a distributed message queue, let's discuss different delivery semantics: at-most once, at-least once, and exactly once.

At-most once

As the name suggests, at-most once means a message will be delivered not more than once. Messages may be lost but are not redelivered. This is how at-most once delivery works at the high level.

- The producer sends a message asynchronously to a topic without waiting for an acknowledgment (ACK=0). If message delivery fails, there is no retry.
- Consumer fetches the message and commits the offset before the data is processed. If the consumer crashes just after offset commit, the message will not be re-consumed.

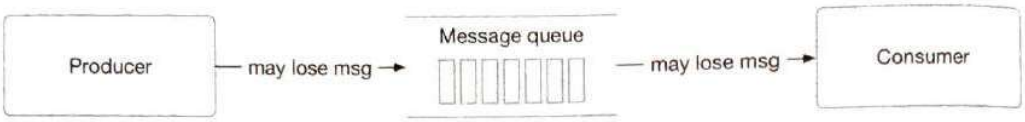


Figure 4.32: At-most once

It is suitable for use cases like monitoring metrics, where a small amount of data loss is acceptable.

At-least once

With this data delivery semantic, it's acceptable to deliver a message more than once, but no message should be lost. Here is how it works at a high level.

- Producer sends a message synchronously or asynchronously with a response callback, setting ACK=1 or ACK=all, to make sure messages are delivered to the broker. If the message delivery fails or timeouts, the producer will keep retrying.
- Consumer fetches the message and commits the offset only after the data is successfully processed. If the consumer fails to process the message, it will re-consume the message so there won't be data loss. On the other hand, if a consumer processes the message but fails to commit the offset to the broker, the message will be re-consumed when the consumer restarts, resulting in duplicates.
- A message might be delivered more than once to the brokers and consumers.

Figure 4.33: At-least once

Use cases: With at-least once, messages won't be lost but the same message might be delivered multiple times. While not ideal from a user perspective, at-least once delivery semantics are usually good enough for use cases where data duplication is not a big issue or deduplication is possible on the consumer side. For example, with a unique key in each message, a message can be rejected when writing duplicate data to the database.

Exactly once

Exactly once is the most difficult delivery semantic to implement. It is friendly to users, but it has a high cost for the system's performance and complexity.

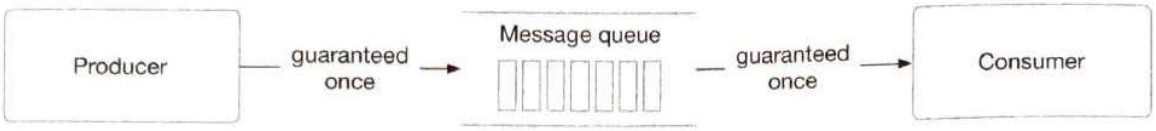


Figure 4.34: Exactly once

Use cases: Financial-related use cases (payment, trading, accounting, etc.). Exactly once is especially important when duplication is not acceptable and the downstream service or third party doesn't support idempotency.

Advanced features

In this section, we talk briefly about some advanced features, such as message filtering, delayed messages, and scheduled messages.

Message filtering

A topic is a logical abstraction that contains messages of the same type. However, some consumer groups may only want to consume messages of certain subtypes. For example, the ordering system sends all the activities about the order to a topic, but the payment system only cares about messages related to checkout and refund.

One option is to build a dedicated topic for the payment system and another topic for the ordering system. This method is simple, but it might raise some concerns.

- What if other systems ask for different subtypes of messages? Do we need to build dedicated topics for every single consumer request?
- It is a waste of resources to save the same messages on different topics.
- The producer needs to change every time a new consumer requirement comes, as the producer and consumer are now tightly coupled.

Therefore, we need to resolve this requirement using a different approach. Luckily, mes-

sage filtering comes to the rescue.

A naive solution for message filtering is that the consumer fetches the full set of messages and filters out unnecessary messages during processing time. This approach is flexible but introduces unnecessary traffic that will affect system performance.

A better solution is to filter messages on the broker side so that consumers will only get messages they care about. Implementing this requires some careful consideration. If data filtering requires data decryption or deserialization, it will degrade the performance of the brokers. Additionally, if messages contain sensitive data, they should not be readable in the message queue.

Therefore, the filtering logic in the broker should not extract the message payload. It is better to put data used for filtering into the metadata of a message, which can be efficiently read by the broker. For example, we can attach a tag to each message. With a message tag, a broker can filter messages in that dimension. If more tags are attached, the messages can be filtered in multiple dimensions. Therefore, a list of tags can support most of the filtering requirements. To support more complex logic such as mathematical formulae, the broker will need a grammar parser or a script executor, which might be too heavyweight for the message queue.

With tags attached to each message, a consumer can subscribe to messages based on the specified tag, as shown in Figure 4.35. Interested readers can refer to the reference material [15].

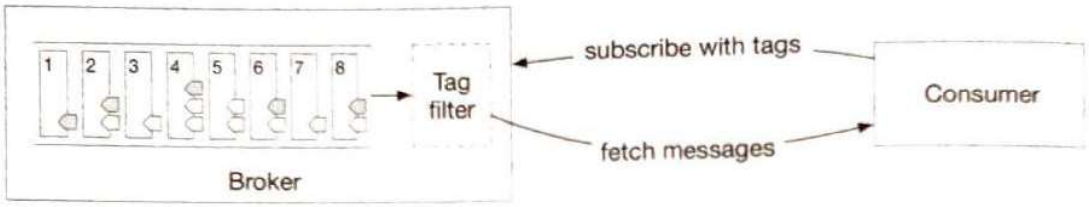


Figure 4.35: Message filtering by tags

Delayed messages & scheduled messages

Sometimes you want to delay the delivery of messages to a consumer for a specified period of time. For example, an order should be closed if not paid within 30 minutes after the order is created. A delayed verification message (check if the payment is completed) is sent immediately but is delivered to the consumer 30 minutes later. When the consumer receives the message, it checks the payment status. If the payment is not completed, the order will be closed. Otherwise, the message will be ignored.

Different from sending instant messages, we can send delayed messages to temporary storage on the broker side instead of to the topics immediately, and then deliver them to the topics when time's up. The high-level design for this is shown in Figure 4.36.

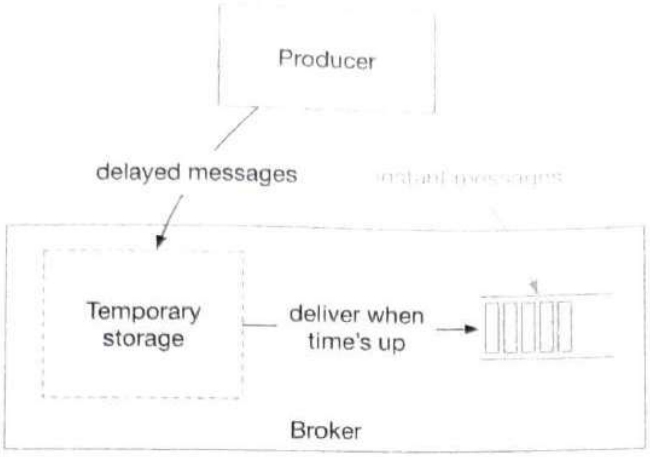


Figure 4.36: Delayed messages

Core components of the system include the temporary storage and the timing function.

- The temporary storage can be one or more special message topics.
- The timing function is out of scope, but here are 2 popular solutions:
  - Dedicated delay queues with predefined delay levels [16]. For example, RocketMQ doesn't support delayed messages with arbitrary time precision, but delayed messages with specific levels are supported. Message delay levels are 1s, 5s, 10s, 30s, 1m, 2m, 3m, 4m, 6m, 8m, 9m, 10m, 20m, 30m, 1h, and 2h.
  - Hierarchical time wheel [17].

A scheduled message means a message should be delivered to the consumer at the scheduled time. The overall design is very similar to delayed messages.

Step 4 - Wrap Up

In this chapter, we have presented the design of a distributed message queue with some advanced features commonly found in data streaming platforms. If there is extra time at the end of the interview, here are some additional talking points:

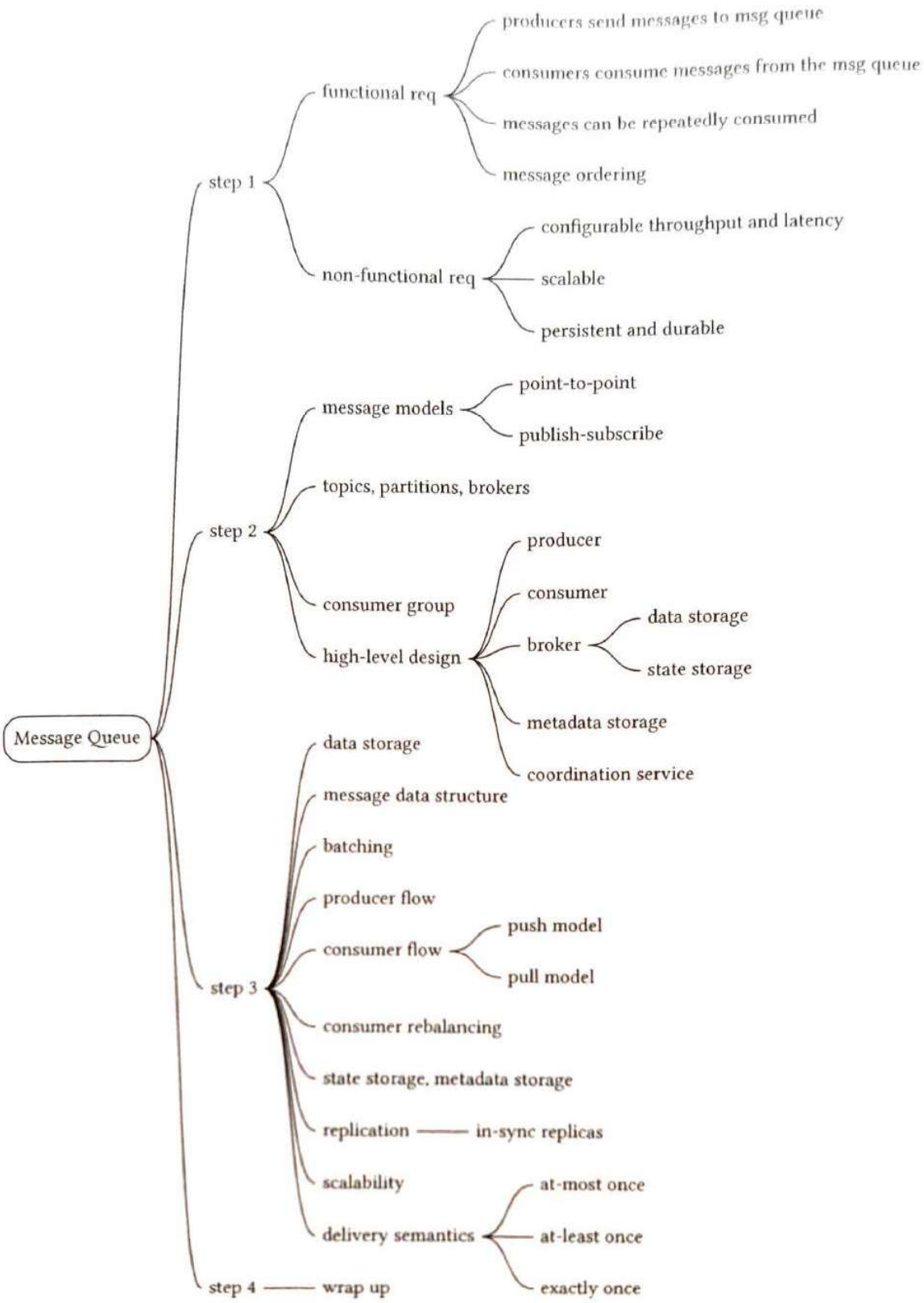
- Protocol: it defines rules, syntax, and APIs on how to exchange information and transfer data between different nodes. In a distributed message queue, the protocol should be able to:
  - Cover all the activities such as production, consumption, heartbeat, etc.
  - Effectively transport data with large volumes.
  - Verify the integrity and correctness of the data.

  Some popular protocols include Advanced Message Queuing Protocol (AMQP) [18] and Kafka protocol [19].

- Retry consumption. If some messages cannot be consumed successfully, we need to retry the operation. In order not to block incoming messages, how can we retry the operation after a certain time period? One idea is to send failed messages to a dedicated retry topic, so they can be consumed later.
- Historical data archive. Assume there is a time-based or capacity-based log retention mechanism. If a consumer needs to replay some historical messages that are already truncated, how can we do it? One possible solution is to use storage systems with large capacities, such as HDFS [20] or object storage, to store historical data.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Queue Length Limit. https://www.rabbitmq.com/maxlength.html.

[2] Apache ZooKeeper - Wikipedia. https://en.wikipedia.org/wiki/Apache_ZooKeeper.

[3] etcd. https://etcd.io/.

[4] MySQL. https://www.mysql.com/.

[5] Comparison of disk and memory performance. https://deliveryimages.acm.org/10.1145/1570000/1563874/jacobs3.jpg.

[6] Push vs. pull. https://kafka.apache.org/documentation/#design_pull.

[7] Kafka 2.0 Documentation. https://kafka.apache.org/20/documentation.html#consumerconfigs.

[8] Kafka No Longer Requires ZooKeeper. https://towardsdatascience.com/kafka-no-longer-requires-zookeeper-ebfbf3862104.

[9] Martin Kleppmann. Replication. In *Designing Data-Intensive Applications*, pages 151–197. O'Reilly Media, 2017.

[10] ISR in Apache Kafka. https://www.cloudkarafka.com/blog/what-does-in-sync-in-apache-kafka-really-mean.html.

[11] Global map in a geographic Coordinate Reference System. https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica.

[12] Hands-free Kafka Replication. https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/.

[13] Kafka high watermark. https://rongxinblog.wordpress.com/2016/07/29/kafka-high-watermark/.

[14] Kafka mirroring. https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=27846330.

[15] Message filtering in RocketMQdtree. https://partners-intl.aliyun.com/help/doc-detail/29543.htm.

[16] Scheduled messages and delayed messages in Apache RocketMQ. https://partners-intl.aliyun.com/help/doc-detail/43349.htm.

[17] Hashed and hierarchical timing wheels. http://www.cs.columbia.edu/~nahum/w6998/papers/sosp87-timing-wheels.pdf.

[18] Advanced Message Queuing Protocol. https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol.

[19] Kafka protocol guide. https://kafka.apache.org/protocol.

[20] HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

5 Metrics Monitoring and Alerting System

In this chapter, we explore the design of a scalable metrics monitoring and alerting system. A well-designed monitoring and alerting system plays a key role in providing clear visibility into the health of the infrastructure to ensure high availability and reliability.

Figure 5.1 shows some of the most popular metrics monitoring and alerting services in the marketplace. In this chapter, we design a similar service that can be used internally by a large company.

Figure 5.1: Popular metrics monitoring and alerting services

Step 1 - Understand the Problem and Establish Design Scope

A metrics monitoring and alerting system can mean many different things to different companies, so it is essential to nail down the exact requirements first with the interviewer. For example, you do not want to design a system that focuses on logs such as web server error or access logs if the interviewer has only infrastructure metrics in mind.

Let's first fully understand the problem and establish the scope of the design before diving into the details.

Candidate: Who are we building the system for? Are we building an in-house system for a large corporation like Facebook or Google, or are we designing a SaaS service like Datadog [1], Splunk [2], etc?
Interviewer: That's a great question. We are building it for internal use only.

Candidate: Which metrics do we want to collect?
Interviewer: We want to collect operational system metrics. These can be low-level usage data of the operating system, such as CPU load, memory usage, and disk space consumption. They can also be high-level concepts such as requests per second of a service or the running server count of a web pool. Business metrics are not in the scope of this design.

Candidate: What is the scale of the infrastructure we are monitoring with this system?
Interviewer: 100 million daily active users, 1,000 server pools, and 100 machines per pool.

Candidate: How long should we keep the data?
Interviewer: Let's assume we want 1 year retention.

Candidate: May we reduce the resolution of the metrics data for long-term storage?
Interviewer: That's a great question. We would like to be able to keep newly received data for 7 days. After 7 days, you may roll them up to a 1 minute resolution for 30 days. After 30 days, you may further roll them up at a 1 hour resolution.

Candidate: What are the supported alert channels?
Interviewer: Email, phone, PagerDuty [3], or webhooks (HTTP endpoints).

Candidate: Do we need to collect logs, such as error log or access log?
Interviewer: No.

Candidate: Do we need to support distributed system tracing?
Interviewer: No.

High-level requirements and assumptions

Now you have finished gathering requirements from the interviewer and have a clear scope of the design. The requirements are:

- The infrastructure being monitored is large-scale.
  - 100 million daily active users
  - Assume we have 1,000 server pools, 100 machines per pool, 100 metrics per machine $\Rightarrow \sim$ 10 million metrics
  - 1 year data retention
  - Data retention policy: raw form for 7 days, 1 minute resolution for 30 days, 1 hour resolution for 1 year.
- A variety of metrics can be monitored, for example:
  - CPU usage

- Request count
- Memory usage
- Message count in message queues

Non-functional requirements

- Scalability. The system should be scalable to accommodate growing metrics and alert volume.
- Low latency. The system needs to have low query latency for dashboards and alerts.
- Reliability. The system should be highly reliable to avoid missing critical alerts.
- Flexibility. Technology keeps changing, so the pipeline should be flexible enough to easily integrate new technologies in the future.

Which requirements are out of scope?

- Log monitoring. The Elasticsearch, Logstash, Kibana (ELK) stack is very popular for collecting and monitoring logs [4].
- Distributed system tracing [5] [6]. Distributed tracing refers to a tracing solution that tracks service requests as they flow through distributed systems. It collects data as requests go from one service to another.

Step 2 - Propose High-level Design and Get Buy-in

In this section, we discuss some fundamentals of building the system, the data model, and the high-level design.

Fundamentals

A metrics monitoring and alerting system generally contains five components, as illustrated in Figure 5.2.

- Data collection: collect metric data from different sources.
- Data transmission: transfer data from sources to the metrics monitoring system.
- Data storage: organize and store incoming data.
- Alerting: analyze incoming data, detect anomalies, and generate alerts. The system must be able to send alerts to different communication channels.
- Visualization: present data in graphs, charts, etc. Engineers are better at identifying patterns, trends, or problems when data is presented visually, so we need visualization functionality.

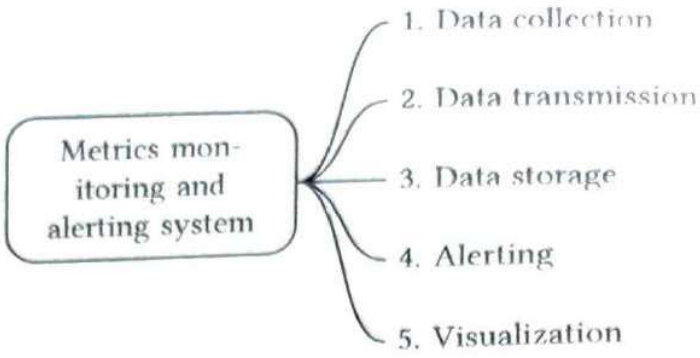


Figure 5.2: Five components of the system

Data model

Metrics data is usually recorded as a time series that contains a set of values with their associated timestamps. The series itself can be uniquely identified by its name, and optionally by a set of labels.

Let's take a look at two examples.

Example 1:

What is the CPU load on production server instance `i631` at `20:00`?

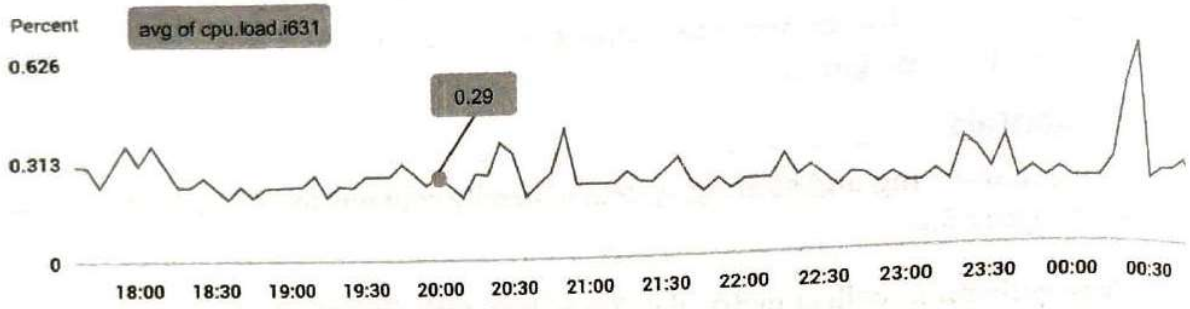


Figure 5.3: Popular metrics monitoring and alerting services

The data point highlighted in Figure 5.3 can be represented by Table 5.1.

metric_name	cpu.load
labels	host:i631,env:prod
timestamp	1613707265
value	0.29

Table 5.1: The data point represented by a table

In this example, the time series is represented by the metric name, the labels (`host:i631,env:prod`), and a single point value at a specific time.

Example 2:

What is the average CPU load across all web servers in the us-west region for the last 10 minutes? Conceptually, we would pull up something like this from storage where the metric name is `CPU.load` and the region label is `us-west`:

```
CPU.load host=webserver01,region=us-west 1613707265 50
CPU.load host=webserver01,region=us-west 1613707265 62
CPU.load host=webserver02,region=us-west 1613707265 43
CPU.load host=webserver02,region=us-west 1613707265 53
...
CPU.load host=webserver01,region=us-west 1613707265 76
CPU.load host=webserver01,region=us-west 1613707265 83
```

The average CPU load could be computed by averaging the values at the end of each line. The format of the lines in the above example is called the line protocol. It is a common input format for many monitoring software in the market. Prometheus [7] and OpenTSDB [8] are two examples.

Every time series consists of the following [9]:

Name	Type
A metric name	String
A set of tags/labels	List of <key:value> pairs
An array of values and their timestamps	An array of <value, timestamp> pairs

Table 5.2: Time series

Data access pattern

In Figure 5.4, each label on the y-axis represents a time series (uniquely identified by the names and labels) while the x-axis represents time.

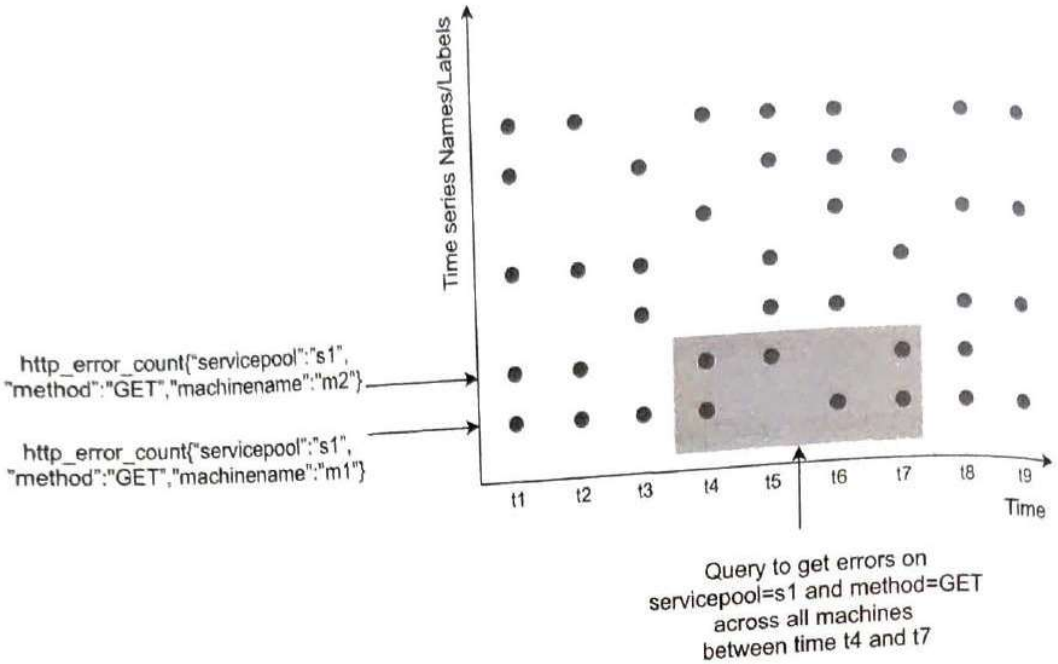


Figure 5.4: Data access pattern

The write load is heavy. As you can see, there can be many time-series data points written at any moment. As we mentioned in the "High-level requirements" section on page 132, about 10 million operational metrics are written per day, and many metrics are collected at high frequency, so the traffic is undoubtedly write-heavy.

At the same time, the read load is spiky. Both visualization and alerting services send queries to the database, and depending on the access patterns of the graphs and alerts, the read volume could be bursty.

In other words, the system is under constant heavy write load, while the read load is spiky.

Data storage system

The data storage system is the heart of the design. It's not recommended to build your own storage system or use a general-purpose storage system (for example, MySQL [10]) for this job.

A general-purpose database, in theory, could support time-series data, but it would require expert-level tuning to make it work at our scale. Specifically, a relational database is not optimized for operations you would commonly perform against time-series data. For example, computing the moving average in a rolling time window requires complicated SQL that is difficult to read (there is an example of this in the deep dive section). Besides, to support tagging/labeling data, we need to add an index for each tag. Moreover, a general-purpose relational database does not perform well under constant heavy write load. At our scale, we would need to expend significant effort in tuning the database, and even then, it might not perform well.

How about NoSQL? In theory, a few NoSQL databases on the market could handle time-series data effectively. For example, Cassandra and Bigtable [11] can both be used for time series data. However, this would require deep knowledge of the internal workings of each NoSQL to devise a scalable schema for effectively storing and querying time-series data. With industrial-scale time-series databases readily available, using a general-purpose NoSQL database is not appealing.

There are many storage systems available that are optimized for time-series data. The optimization lets us use far fewer servers to handle the same volume of data. Many of these databases also have custom query interfaces specially designed for the analysis of time-series data that are much easier to use than SQL. Some even provide features to manage data retention and data aggregation. Here are a few examples of time-series databases.

OpenTSDB is a distributed time-series database, but since it is based on Hadoop and HBase, running a Hadoop/HBase cluster adds complexity. Twitter uses MetricsDB [12], and Amazon offers Timestream as a time-series database [13]. According to DB-engines [14], the two most popular time-series databases are InfluxDB [15] and Prometheus, which are designed to store large volumes of time-series data and quickly perform real-time analysis on that data. Both of them primarily rely on an in-memory cache and on-disk storage. And they both handle durability and performance quite well. As shown in Figure 5.5, an InfluxDB with 8 cores and 32GB RAM can handle over 250,000 writes per second.

vCPU or CPU	RAM	IOPS	Writes per second	Queries* per second	Unique series
2-4 cores	2-4 GB	500	< 5,000	< 5	< 100,000
4-6 cores	8-32 GB	500-1000	< 250,000	< 25	< 1,000,000
8+ cores	32+ GB	1000+	> 250,000	> 25	> 1,000,000

Figure 5.5: InfluxDb benchmarking

Since a time-series database is a specialized database, you are not expected to understand the internals in an interview unless you explicitly mentioned it in your resume. For the purpose of an interview, it's important to understand the metrics data are time-series in nature and we can select time-series databases such as InfluxDB for storage to store them.

Another feature of a strong time-series database is efficient aggregation and analysis of a large amount of time-series data by labels, also known as tags in some databases. For example, InfluxDB builds indexes on labels to facilitate the fast lookup of time-series by labels [15]. It provides clear best-practice guidelines on how to use labels, without overloading the database. The key is to make sure each label is of low cardinality (having

a small set of possible values). This feature is critical for visualization, and it would take a lot of effort to build this with a general-purpose database.

High-level design

The high-level design diagram is shown in Figure 5.6.

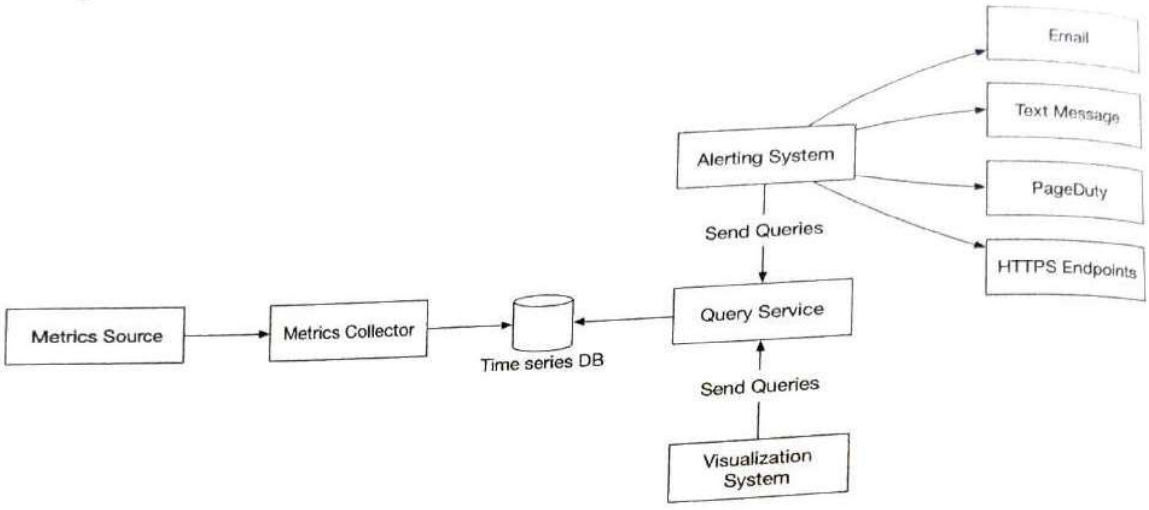


Figure 5.6: High-level design

- **Metrics source.** This can be application servers, SQL databases, message queues, etc.
- **Metrics collector.** It gathers metrics data and writes data into the time-series database.
- **Time-series database.** This stores metrics data as time series. It usually provides a custom query interface for analyzing and summarizing a large amount of time-series data. It maintains indexes on labels to facilitate the fast lookup of time-series data by labels.
- **Query service.** The query service makes it easy to query and retrieve data from the time-series database. This should be a very thin wrapper if we choose a good time-series database. It could also be entirely replaced by the time-series database's own query interface.
- **Alerting system.** This sends alert notifications to various alerting destinations.
- **Visualization system.** This shows metrics in the form of various graphs/charts.

Step 3 - Design Deep Dive

In a system design interview, candidates are expected to dive deep into a few key components or flows. In this section, we investigate the following topics in detail:

- Metrics collection
- Scaling the metrics transmission pipeline

- Query service
- Storage layer
- Alerting system
- Visualization system

Metrics collection

For metrics collection like counters or CPU usage, occasional data loss is not the end of the world. It's acceptable for clients to fire and forget. Now let's take a look at the metrics collection flow. This part of the system is inside the dashed box (Figure 5.7).

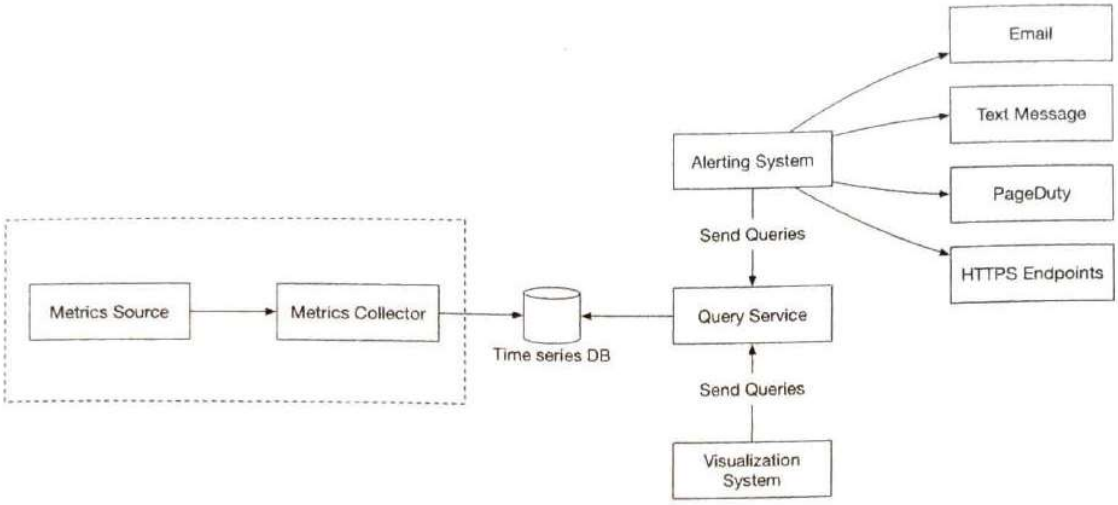


Figure 5.7: Metrics collection flow

Pull vs push models

There are two ways metrics data can be collected, pull or push. It is a routine debate as to which one is better and there is no clear answer. Let's take a close look.

Pull model

Figure 5.8 shows data collection with a pull model over HTTP. We have dedicated metric collectors which pull metrics values from the running applications periodically.

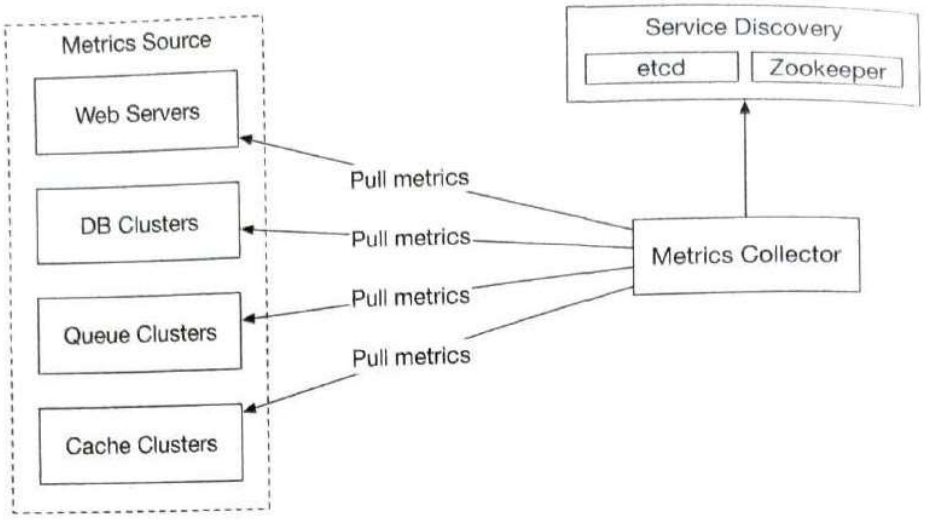


Figure 5.8: Pull model

In this approach, the metrics collector needs to know the complete list of service endpoints to pull data from. One naive approach is to use a file to hold DNS/IP information for every service endpoint on the "metric collector" servers. While the idea is simple, this approach is hard to maintain in a large-scale environment where servers are added or removed frequently, and we want to ensure that metric collectors don't miss out on collecting metrics from any new servers. The good news is that we have a reliable, scalable, and maintainable solution available through Service Discovery, provided by etcd [16], ZooKeeper [17], etc., wherein services register their availability and the metrics collector can be notified by the Service Discovery component whenever the list of service endpoints changes.

Service discovery contains configuration rules about when and where to collect metrics as shown in Figure 5.9.

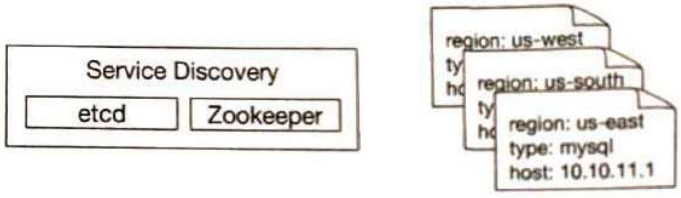


Figure 5.9: Service discovery

Figure 5.10 explains the pull model in detail.

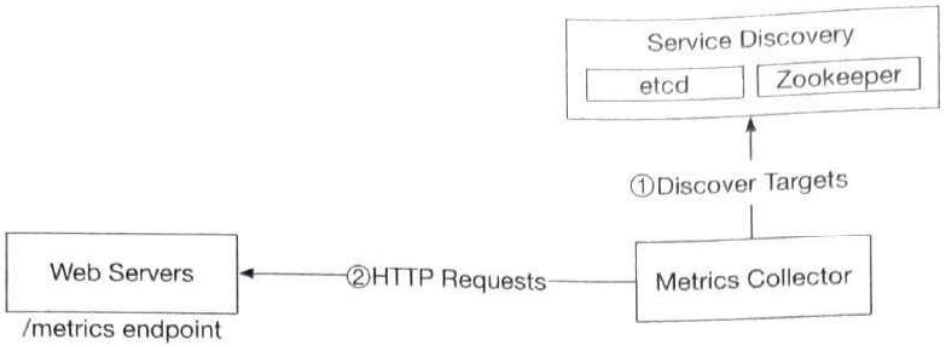


Figure 5.10: Pull model in detail

1. The metrics collector fetches configuration metadata of service endpoints from Service Discovery. Metadata include pulling interval, IP addresses, timeout and retry parameters, etc.
2. The metrics collector pulls metrics data via a pre-defined HTTP endpoint (for example, `/metrics`). To expose the endpoint, a client library usually needs to be added to the service. In Figure 5.10, the service is Web Servers.
3. Optionally, the metrics collector registers a change event notification with Service Discovery to receive an update whenever the service endpoints change. Alternatively, the metrics collector can poll for endpoint changes periodically.

At our scale, a single metrics collector will not be able to handle thousands of servers. We must use a pool of metrics collectors to handle the demand. One common problem when there are multiple collectors is that multiple instances might try to pull data from the same resource and produce duplicate data. There must exist some coordination scheme among the instances to avoid this.

One potential approach is to designate each collector to a range in a consistent hash ring, and then map every single server being monitored by its unique name in the hash ring. This ensures one metrics source server is handled by one collector only. Let's take a look at an example.

As shown in Figure 5.11, there are four collectors and six metrics source servers. Each collector is responsible for collecting metrics from a distinct set of servers. Collector 2 is responsible for collecting metrics from Server 1 and Server 5.

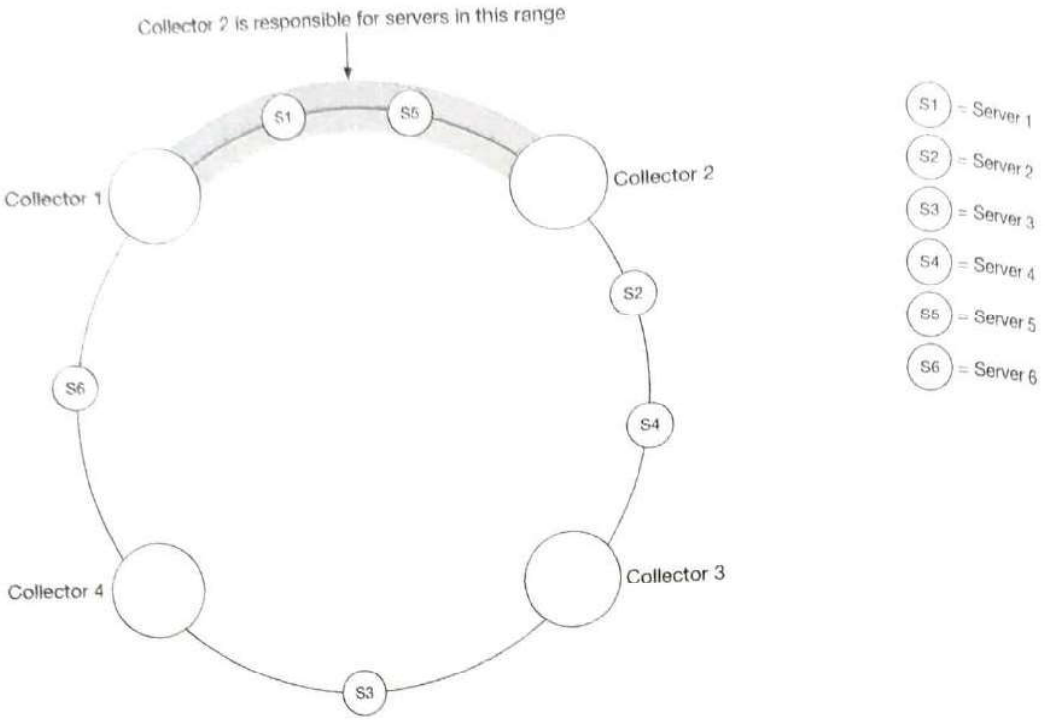


Figure 5.11: Consistent hashing

Push model

As shown in Figure 5.12, in a push model various metrics sources, such as web servers, database servers, etc., directly send metrics to the metrics collector.

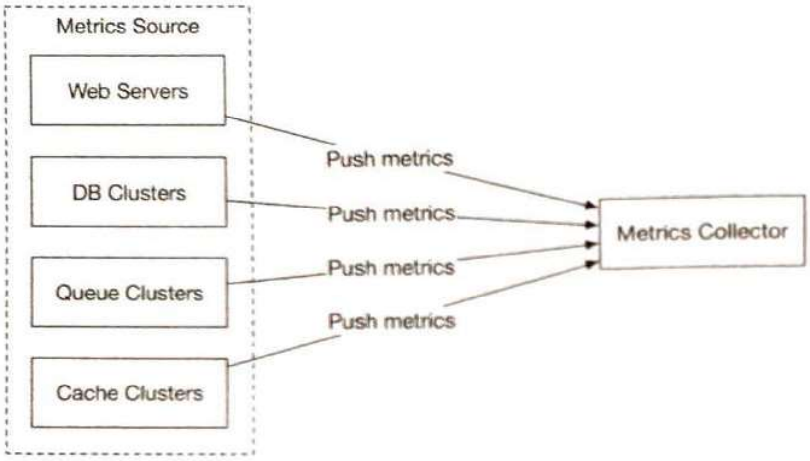


Figure 5.12: Push model

In a push model, a collection agent is commonly installed on every server being monitored. A collection agent is a piece of long-running software that collects metrics from the services running on the server and pushes those metrics periodically to the metrics collector. The collection agent may also aggregate metrics (especially a simple counter) locally, before sending them to metric collectors.

Aggregation is an effective way to reduce the volume of data sent to the metrics collector. If the push traffic is high and the metrics collector rejects the push with an error, the agent could keep a small buffer of data locally (possibly by storing them locally on disk), and resend them later. However, if the servers are in an auto-scaling group where they are rotated out frequently, then holding data locally (even temporarily) might result in data loss when the metrics collector falls behind.

To prevent the metrics collector from falling behind in a push model, the metrics collector should be in an auto-scaling cluster with a load balancer in front of it (Figure 5.13). The cluster should scale up and down based on the CPU load of the metric collector servers.

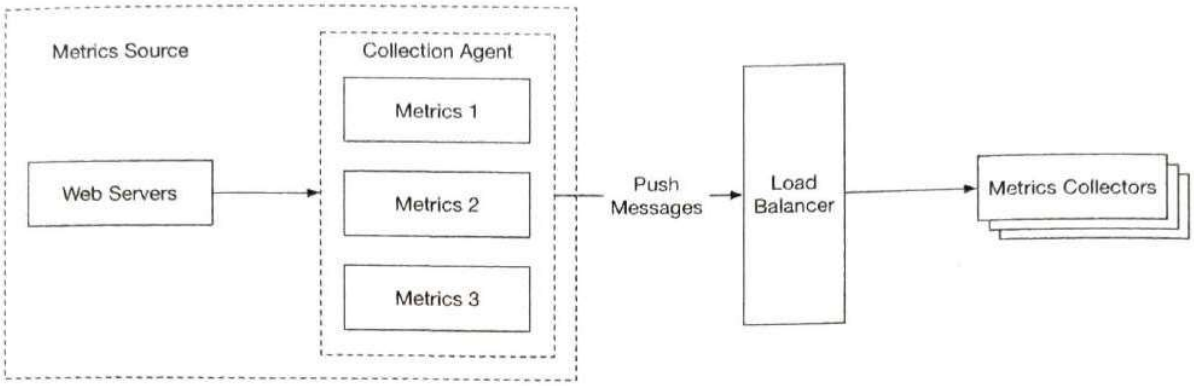


Figure 5.13: Load balancer

Pull or push?

So, which one is the better choice for us? Just like many things in life, there is no clear answer. Both sides have widely adopted real-world use cases.

- Examples of pull architectures include Prometheus.
- Examples of push architectures include Amazon CloudWatch [18] and Graphite [19].

Knowing the advantages and disadvantages of each approach is more important than picking a winner during an interview. Table 5.3 compares the pros and cons of push and pull architectures [20] [21] [22] [23].

	Pull	Push
Easy debugging	The /metrics endpoint on application servers used for pulling metrics can be used to view metrics at any time. You can even do this on your laptop. **Pull wins.**	
Health check	If an application server doesn't respond to the pull, you can quickly figure out if an application server is down. **Pull wins.**	If the metrics collector doesn't receive metrics, the problem might be caused by network issues.
Short-lived jobs		Some of the batch jobs might be short-lived and don't last long enough to be pulled. **Push wins.** This can be fixed by introducing push gateways for the pull model [24].
Firewall or complicated network setups	Having servers pulling metrics requires all metric endpoints to be reachable. This is potentially problematic in multiple data center setups. It might require a more elaborate network infrastructure.	If the metrics collector is set up with a load balancer and an auto-scaling group, it is possible to receive data from anywhere. **Push wins.**

Performance	Pull methods typically use TCP.	Push methods typically use UDP. This means the push method provides lower-latency transports of metrics. The counterargument here is that the effort of establishing a TCP connection is small compared to sending the metrics payload.
Data authenticity	Application servers to collect metrics from are defined in config files in advance. Metrics gathered from those servers are guaranteed to be authentic.	Any kind of client can push metrics to the metrics collector. This can be fixed by whitelisting servers from which to accept metrics, or by requiring authentication.

Table 5.3: Pull vs push

As mentioned above, pull vs push is a routine debate topic and there is no clear answer. A large organization probably needs to support both, especially with the popularity of serverless [25] these days. There might not be a way to install an agent from which to push data in the first place.

Scale the metrics transmission pipeline

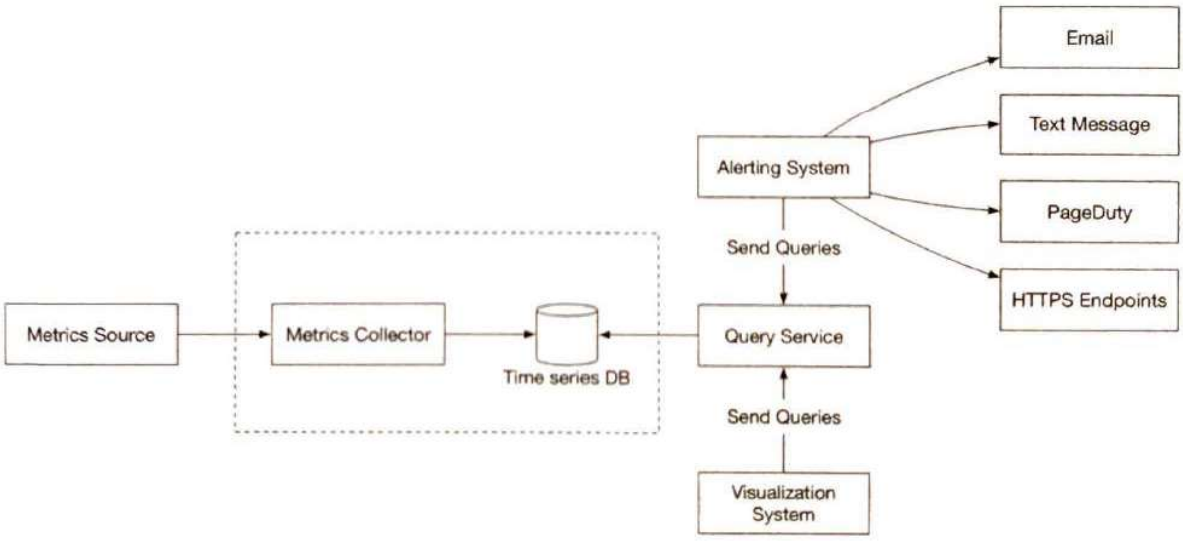


Figure 5.14: Metrics transmission pipeline

Let's zoom in on the metrics collector and time-series databases. Whether you use the push or pull model, the metrics collector is a cluster of servers, and the cluster receives enormous amounts of data. For either push or pull, the metrics collector cluster is set up for auto-scaling, to ensure that there are an adequate number of collector instances to handle the demand.

However, there is a risk of data loss if the time-series database is unavailable. To mitigate this problem, we introduce a queueing component as shown in Figure 5.15.

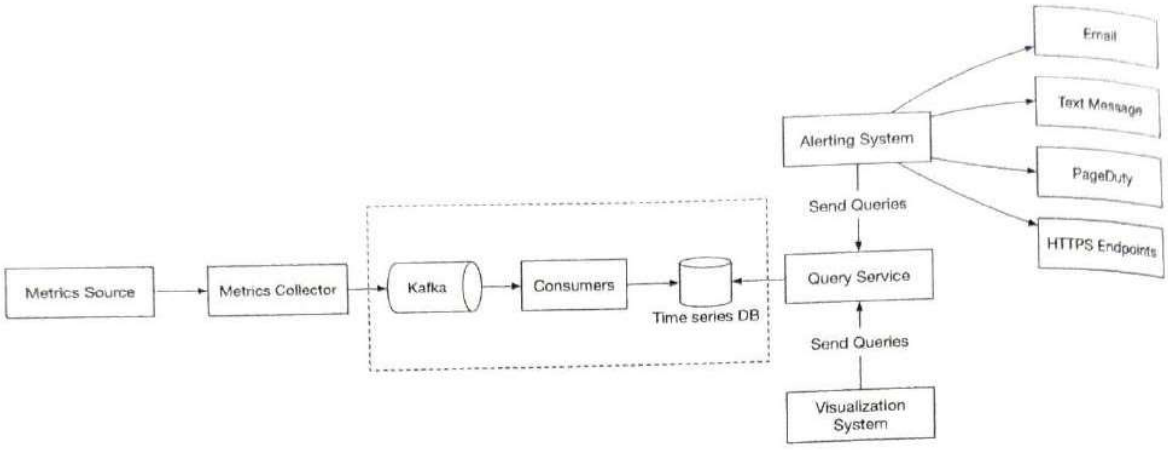


Figure 5.15: Add queues

In this design, the metrics collector sends metrics data to queuing systems like Kafka. Then consumers or streaming processing services such as Apache Storm, Flink, and Spark, process and push data to the time-series database. This approach has several advantages:

- Kafka is used as a highly reliable and scalable distributed messaging platform.
- It decouples the data collection and data processing services from each other.
- It can easily prevent data loss when the database is unavailable, by retaining the data in Kafka.

Scale through Kafka

There are a couple of ways that we can leverage Kafka's built-in partition mechanism to scale our system.

- Configure the number of partitions based on throughput requirements.
- Partition metrics data by metric names, so consumers can aggregate data by metrics names.
- Further partition metrics data with tags/labels.
- Categorize and prioritize metrics so that important metrics can be processed first.

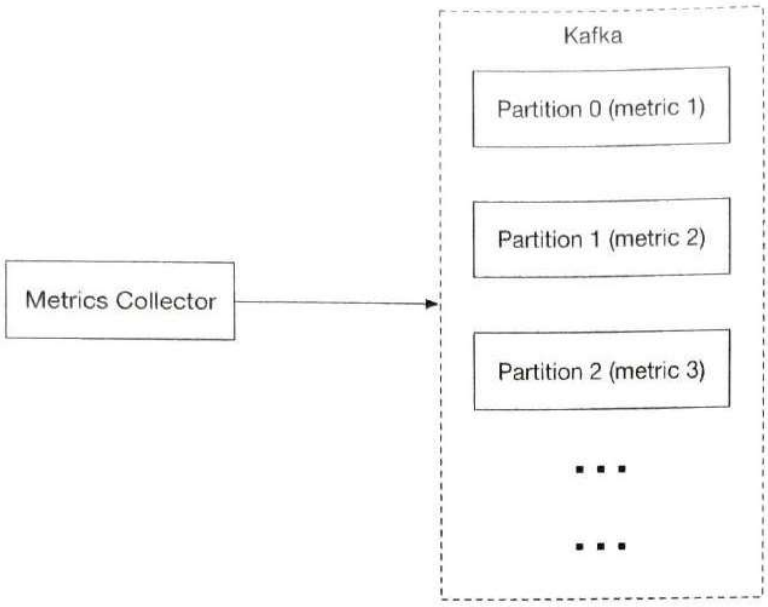


Figure 5.16: Kafka partition

Alternative to Kafka

Maintaining a production-scale Kafka system is no small undertaking. You might get pushback from the interviewer about this. There are large-scale monitoring ingestion systems in use without using an intermediate queue. Facebook's Gorilla [26] in-memory time-series database is a prime example; it is designed to remain highly available for writes, even when there is a partial network failure. It could be argued that such a design is as reliable as having an intermediate queue like Kafka.

Where aggregations can happen

Metrics can be aggregated in different places; in the collection agent (on the client-side), the ingestion pipeline (before writing to storage), and the query side (after writing to storage). Let's take a closer look at each of them.

Collection agent. The collection agent installed on the client-side only supports simple aggregation logic. For example, aggregate a counter every minute before it is sent to the metrics collector.

Ingestion pipeline. To aggregate data before writing to the storage, we usually need stream processing engines such as Flink. The write volume will be significantly reduced since only the calculated result is written to the database. However, handling late-arriving events could be a challenge and another downside is that we lose data precision and some flexibility because we no longer store the raw data.

Query side. Raw data can be aggregated over a given time period at query time. There is no data loss with this approach, but the query speed might be slower because the query result is computed at query time and is run against the whole dataset.

Query service

The query service comprises a cluster of query servers, which access the time-series databases and handle requests from the visualization or alerting systems. Having a dedicated set of query servers decouples time-series databases from the clients (visualization and alerting systems). And this gives us the flexibility to change the time-series database or the visualization and alerting systems, whenever needed.

Cache layer

To reduce the load of the time-series database and make query service more performant, cache servers are added to store query results, as shown in Figure 5.17.

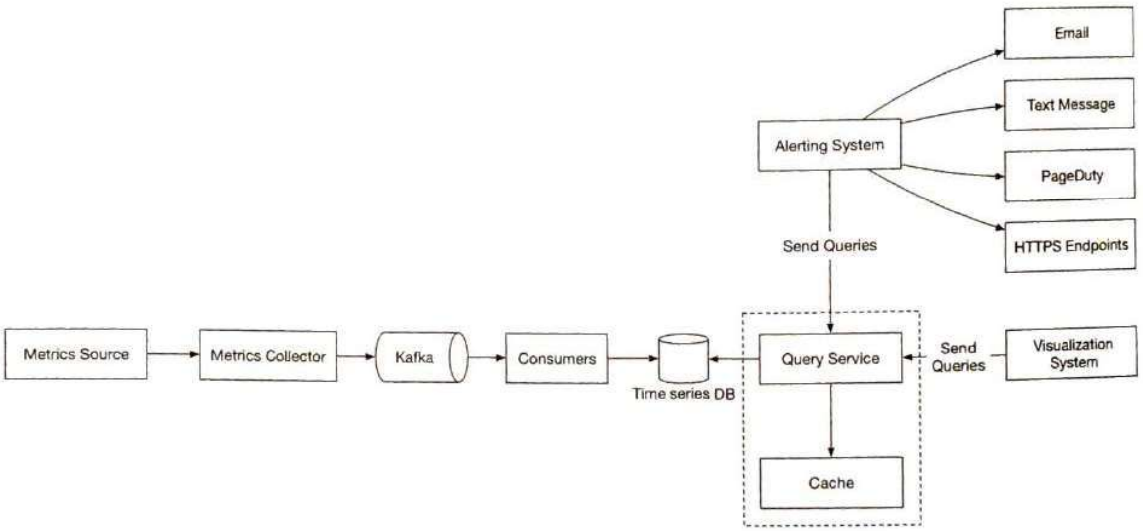


Figure 5.17: Cache layer

The case against query service

There might not be a pressing need to introduce our own abstraction (a query service) because most industrial-scale visual and alerting systems have powerful plugins to interface with well-known time-series databases on the market. And with a well-chosen time-series database, there is no need to add our own caching, either.

Time-series database query language

Most popular metrics monitoring systems like Prometheus and InfluxDB don't use SQL and have their own query languages. One major reason for this is that it is hard to build SQL queries to query time-series data. For example, as mentioned here [27], computing an exponential moving average might look like this in SQL:

```
select id,
       temp,
       avg(temp) over (partition by group_nr order by
time_read)
       as rolling_avg
from (
    select id,
         temp,
         time_read,
         interval_group,
         id - row_number() over (partition by interval_group
order
         by time_read) as group_nr
    from (
         select id,
         time_read,
         "epoch"::timestamp + "900 seconds"::interval * (
extract(epoch from time_read)::int4 / 900) as interval_group,
         temp
         from readings
    ) t1
) t2
order by time_read;
```

While in Flux, a language that's optimized for time-series analysis (used in InfluxDB), it looks like this. As you can see, it's much easier to understand.

```
from(db:"telegraf")
    |> range(start:-1h)
    |> filter(fn: (r) => r._measurement == "foo")
    |> exponentialMovingAverage(size:-10s)
```

Storage layer

Now let's dive into the storage layer.

Choose a time-series database carefully

According to a research paper published by Facebook [26], at least 85% of all queries to the operational data store were for data collected in the past 26 hours. If we use a time-series database that harnesses this property, it could have a significant impact on overall system performance. If you are interested in the design of the storage engine, please refer to the design document of the InfluxDB storage engine [28].

Space optimization

As explained in high-level requirements, the amount of metric data to store is enormous. Here are a few strategies for tackling this.

Data encoding and compression

Data encoding and compression can significantly reduce the size of data. Those features are usually built into a good time-series database. Here is a simple example.

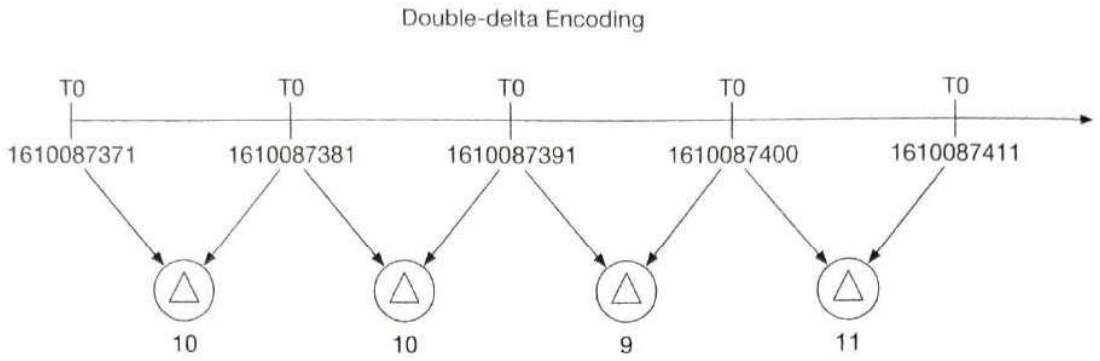


Figure 5.18: Data encoding

As you can see in the image above, 1610087371 and 1610087381 differ by only 10 seconds, which takes only 4 bits to represent, instead of the full timestamp of 32 bits. So, rather than storing absolute values, the delta of the values can be stored along with one base value like: 1610087371, 10, 10, 9, 11.

Downsampling

Downsampling is the process of converting high-resolution data to low-resolution to reduce overall disk usage. Since our data retention is 1 year, we can downsample old data. For example, we can let engineers and data scientists define rules for different metrics. Here is an example:

- Retention: 7 days, no sampling
- Retention: 30 days, downsample to 1 minute resolution
- Retention: 1 year, downsample to 1 hour resolution

Let's take a look at another concrete example. It aggregates 10-second resolution data to 30-second resolution data.

metric	timestamp	hostname	metric_value
cpu	2021-10-24T19:00:00Z	host-a	10
cpu	2021-10-24T19:00:10Z	host-a	16
cpu	2021-10-24T19:00:20Z	host-a	20
cpu	2021-10-24T19:00:30Z	host-a	30
cpu	2021-10-24T19:00:40Z	host-a	20
cpu	2021-10-24T19:00:50Z	host-a	30

Table 5.4: 10-second resolution data

Rollup from 10 second resolution data to 30 second resolution data.

metric	timestamp	hostname	Metric_value (avg)
cpu	2021-10-24T19:00:00Z	host-a	19
cpu	2021-10-24T19:00:30Z	host-a	25

Table 5.5: 30-second resolution data

Cold storage

Cold storage is the storage of inactive data that is rarely used. The financial cost for cold storage is much lower.

In a nutshell, we should probably use third-party visualization and alerting systems, instead of building our own.

Alerting system

For the purpose of the interview, let's look at the alerting system, shown in Figure 5.19 below.

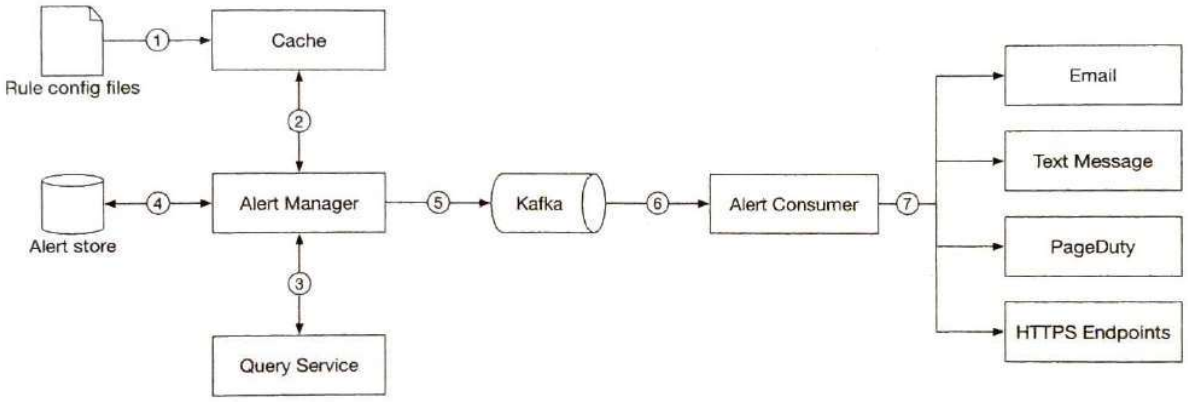


Figure 5.19: Alerting system

The alert flow works as follows:

1. Load config files to cache servers. Rules are defined as config files on the disk. YAML [29] is a commonly used format to define rules. Here is an example of alert rules:

```
- name: instance_down
rules:

# Alert for any instance that is unreachable for >5
minutes.
- alert: instance_down
    expr: up == 0
    for: 5m
    labels:
    severity: page
```

2. The alert manager fetches alert configs from the cache.

3. Based on config rules, the alert manager calls the query service at a predefined interval. If the value violates the threshold, an alert event is created. The alert manager is responsible for the following:
    - Filter, merge, and dedupe alerts. Here is an example of merging alerts that are triggered within one instance within a short amount of time (instance 1) (Figure 5.20).

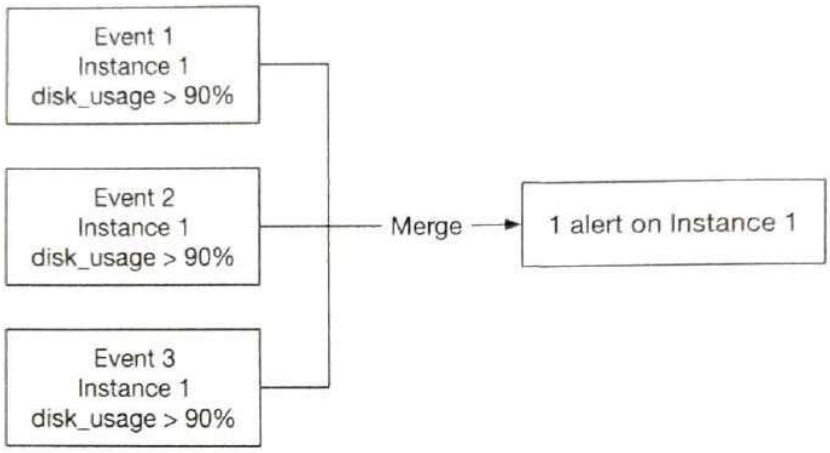


Figure 5.20: Merge alerts

    - Access control. To avoid human error and keep the system secure, it is essential to restrict access to certain alert management operations to authorized individuals only.
    - Retry. The alert manager checks alert states and ensures a notification is sent at least once.
4. The alert store is a key-value database, such as Cassandra, that keeps the state (inactive, pending, firing, resolved) of all alerts. It ensures a notification is sent at least once.
5. Eligible alerts are inserted into Kafka.
6. Alert consumers pull alert events from Kafka.
7. Alert consumers process alert events from Kafka and send notifications over to different channels such as email, text message, PagerDuty, or HTTP endpoints.

Alerting system - build vs buy

There are many industrial-scale alerting systems available off-the-shelf, and most provide tight integration with the popular time-series databases. Many of these alerting systems integrate well with existing notification channels, such as email and PagerDuty. In the real world, it is a tough call to justify building your own alerting system. In interview settings, especially for a senior position, be ready to justify your decision.

Visualization system

Visualization is built on top of the data layer. Metrics can be shown on the metrics dashboard over various time scales and alerts can be shown on the alerts dashboard. Figure 5.21 shows a dashboard that displays some of the metrics like the current server requests, memory/CPU utilization, page load time, traffic, and login information [30].

Figure 5.21: Grafana UI

A high-quality visualization system is hard to build. The argument for using an off-the-shelf system is very strong. For example, Grafana can be a very good system for this purpose. It integrates well with many popular time-series databases which you can buy.

Step 4 - Wrap Up

In this chapter, we presented the design for a metrics monitoring and alerting system. At a high level, we talked about data collection, time-series database, alerts, and visualization. Then we went in-depth into some of the most important techniques/components:

- Pull vs pull model for collecting metrics data.
- Utilize Kafka to scale the system.
- Choose the right time-series database.

- Use downsampling to reduce data size.
- Build vs buy options for alerting and visualization systems.

We went through a few iterations to refine the design, and our final design looks like this:

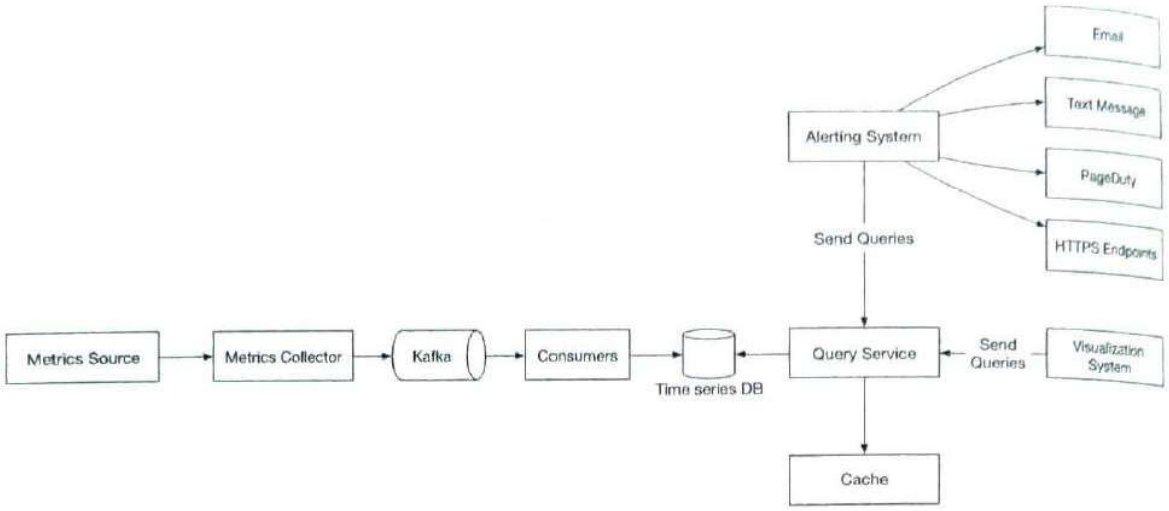
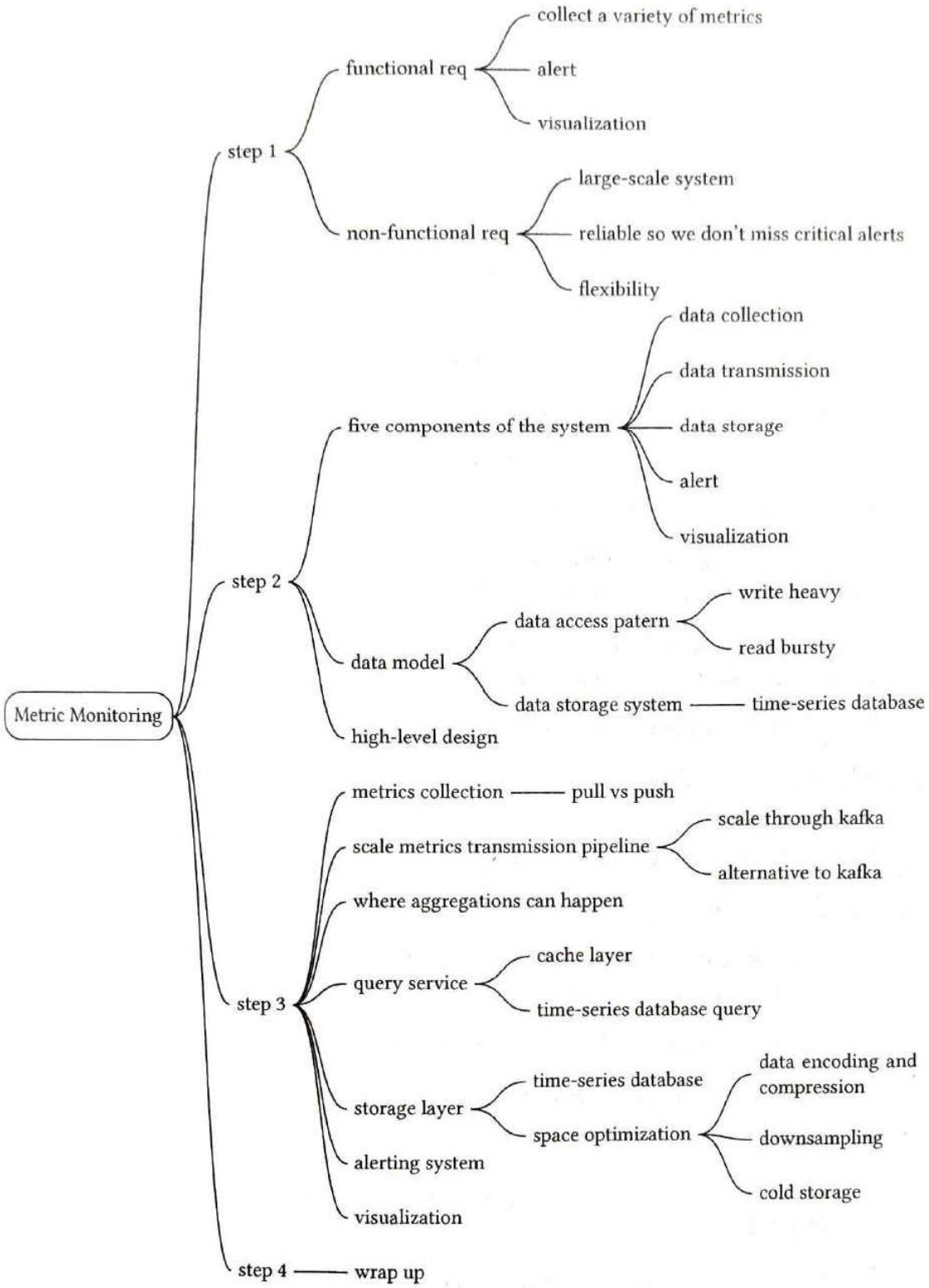


Figure 5.22: Final design

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Datadog. https://www.datadoghq.com/.

[2] Splunk. https://www.splunk.com/.

[3] PagerDuty. https://www.pagerduty.com/.

[4] Elastic stack. https://www.elastic.co/elastic-stack.

[5] Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. https://research.google/pubs/pub36356/.

[6] Distributed Systems Tracing with Zipkin. https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.html.

[7] Prometheus. https://prometheus.io/docs/introduction/overview/.

[8] OpenTSDB - A Distributed, Scalable Monitoring System. http://opentsdb.net/.

[9] Data model. :https://prometheus.io/docs/concepts/data_model/.

[10] MySQL. https://www.mysql.com/.

[11] Schema design for time-series data | Cloud Bigtable Documentation. https://cloud.google.com/bigtable/docs/schema-design-time-series.

[12] MetricsDB. TimeSeriesDatabaseforstoringmetricsatTwitter:https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/metricsdb.html.

[13] Amazon Timestream. https://aws.amazon.com/timestream/.

[14] DB-Engines Ranking of time-series DBMS. https://db-engines.com/en/ranking/time+series+dbms.

[15] InfluxDB. https://www.influxdata.com/.

[16] etcd. https://etcd.io/.

[17] Service Discovery with ZooKeeper. https://cloud.spring.io/spring-cloud-zookeeper/1.2.x/multi/multi_spring-cloud-zookeeper-discovery.html.

[18] Amazon CloudWatch. https://aws.amazon.com/cloudwatch/.

[19] Graphite. https://graphiteapp.org/.

[20] Push vs Pull. http://bit.ly/3aJEPxE.

[21] Pull doesn't scale - or does it? https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/.

[22] Monitoring Architecture. https://developer.lightbend.com/guides/monitoring-at-scale/monitoring-architecture/architecture.html.

[23] Push vs Pull in Monitoring Systems. https://giedrius.blog/2019/05/11/push-vs-pull-in-monitoring-systems/.