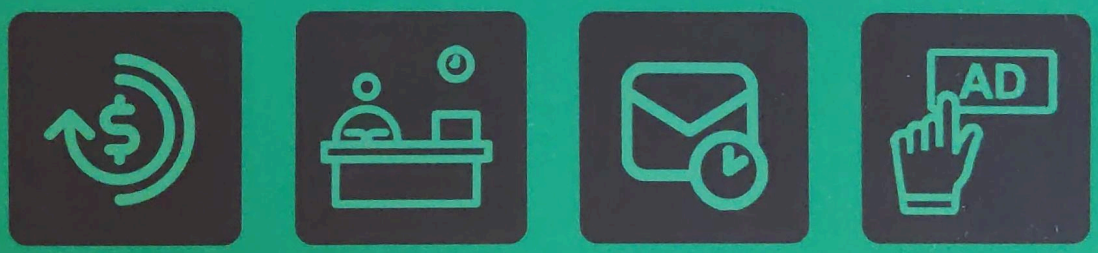
VOLUME 2

SYSTEM DESIGN INTERVIEW



AN INSIDER'S GUIDE

ByteByteGo

Alex Xu & Sahn Lam

System Design Interview

An Insider's Guide

Volume 2

Alex Xu | Sahn Lam

ByteByteGo

SYSTEM DESIGN INTERVIEW - AN INSIDER'S GUIDE (VOLUME 2)

Copyright ©2022 Byte Code LLC

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review.

Join the community

We created a members-only Discord group. It is designed for community discussions on the following topics:

- System design fundamentals.
- Showcasing design diagrams and getting feedback.
- Finding mock interview buddies.
- General chat with community members.

Come join us and introduce yourself to the community, today! Use the link below or scan the barcode.

Invite link: http://bit.ly/systemdiscord

Contents

Foreword		iii
Acknowledgements		v
Chapter 1	Proximity Service	1
Chapter 2	Nearby Friends	35
Chapter 3	Google Maps	59
Chapter 4	Distributed Message Queue	91
Chapter 5	Metrics Monitoring and Alerting System	131
Chapter 6	Ad Click Event Aggregation	159
Chapter 7	Hotel Reservation System	195
Chapter 8	Distributed Email Service	225
Chapter 9	S3-like Object Storage	253
Chapter 10	Real-time Gaming Leaderboard	289
Chapter 11	Payment System	315
Chapter 12	Digital Wallet	341
Chapter 13	Stock Exchange	379
Afterword		415
Index		417

Foreword

We are delighted you are joining us to become better equipped for system design interviews. System design interviews are the most difficult to tackle of all technical interview questions. The questions test the interviewees' ability to design a scalable software system. This could be a news feed, Google search, chat application, or any other system. These questions are intimidating and there is no fixed pattern to follow when tackling them. The questions are usually very broad and vague. They are open-ended, with several plausible angles of attack, and often no perfect answer.

Many companies ask system design interview questions because the communication and problem-solving skills they test for are similar to the skills that software engineers use in their daily work. A candidate is evaluated on how they analyze a vague problem and how they solve it, step by step.

System design questions are open-ended. As in the real world, a design can have numerous variations. The desired outcome is an architecture that satisfies the agreed design goals. The discussions may go in different directions. Some interviewers may choose high-level architecture to cover all aspects of the challenge, whereas others might focus on one or more specific areas. Typically, system requirements, constraints, and bottlenecks should be well understood by the candidate, to shape the direction of the interview.

The objective of this book is to provide a reliable strategy and knowledge base for approaching a broad range of system design questions. The right strategy and knowledge are vital for the success of an interview.

This book also provides a step-by-step framework for how to tackle a system design question. It provides many examples to illustrate the systematic approach, with detailed steps that you can follow. With regular practice, you will be well-equipped to tackle system design interview questions.

This book can be seen as a sequel to the book: System Design Interview - An Insider's Guide (Volume 1: https://bit.ly/systemdesigning). Although reading Volume 1 is helpful, it is not a necessity to do so before you read this. This book should be accessible to readers who have a basic understanding of distributed systems. Let's get started!

Additional Resources

This book contains references at the end of each chapter. The following Github repository contains all the clickable links.

https://bit.ly/systemDesignLinks

You can connect with Alex on social media, where he shares system design interview tips every week.

twitter.com/alexxubyte

bit.ly/linkedinaxu

Acknowledgements

We wish we could say all the designs in this book are original. The truth is that most of the ideas discussed here can also be found elsewhere; in engineering blogs, research papers, code, tech talks, and other places. We have collected these elegant ideas and considered them, then added our personal experiences, to present them here in an easy-to-understand way. Additionally, this book has been written with the significant input and reviews of more than a dozen engineers and managers, some of whom made large writing contributions to the chapters. Thank you so much!

- Proximity Service, Meng Duan (Tencent)
- Nearby Friends, Yan Guo (Amazon)
- Google Maps, Ali Aminian (Adobe, Google)
- Distributed Message Queue, Lionel Liu (eBay)
- Distributed Message Queue, Tanmay Deshpande (Schlumberger)
- Ad Click Event Aggregation, Xinda Bian (Ant Group)
- Real-time Gaming Leaderboard, Jossie Haines (Tile)
- Distributed Email Servers, Kevin Henrikson (Instacart)
- Distributed Email Servers, JJ Zhuang (Instacart)
- S3-like Object Store, Zhiteng Huang (eBay)

We are particularly grateful to those who provided detailed feedback on an earlier draft of this book:

- Darshit Dave (Bloomberg)
- Dwaraknath Bakshi (Twitter)
- Fei Nan (Gusto, Airbnb)
- Richard Hsu (Amazon)
- Simon Gao (Google)
- Stanly Mathew Thomas (Microsoft)

- Wenhan Wang (Tiktok)
- Shiwakant Bharti (Amazon)

A huge thanks to our editors, Dominic Gover and Doug Warren. Your feedback was invaluable.

Last but not least, very special thanks to Elvis Ren and Hua Li for their invaluable contributions. This book wouldn't be what it is without them.

1 Proximity Service

In this chapter, we design a proximity service. A proximity service is used to discover nearby places such as restaurants, hotels, theaters, museums, etc., and is a core component that powers features like finding the best restaurants nearby on Yelp or finding k-nearest gas stations on Google Maps. Figure 1.1 shows the user interface via which you can search for nearby restaurants on Yelp [1]. Note the map tiles used in this book are from Stamen Design [2] and data are from OpenStreetMap [3].

Figure 1.1: Nearby search on Yelp

Step 1 - Understand the Problem and Establish Design Scope

Yelp supports many features and it is not feasible to design all of them in an interview session, so it's important to narrow down the scope by asking questions. The interactions between the interviewer and the candidate could look like this:

Candidate: Can a user specify the search radius? If there are not enough businesses within the search radius, does the system expand the search?

Interviewer: That's a great question. Let's assume we only care about businesses within a specified radius. If time allows, we can then discuss how to expand the search if there are not enough businesses within the radius.

Candidate: What's the maximal radius allowed? Can I assume it's 20km (12.5 miles)?
Interviewer: That's a reasonable assumption.

Candidate: Can a user change the search radius on the UI?
Interviewer: Yes, we have the following options: 0.5km (0.31 mile), 1km (0.62 mile), 2km (1.24 mile), 5km (3.1 mile), and 20km (12.42 mile).

Candidate: How does business information get added, deleted, or updated? Do we need to reflect these operations in real-time?
Interviewer: Business owners can add, delete or update a business. Assume we have a business agreement upfront that newly added/updated businesses will be effective the next day.

Candidate: A user might be moving while using the app/website, so the search results could be slightly different after a while. Do we need to refresh the page to keep the results up to date?
Interviewer: Let's assume a user's moving speed is slow and we don't need to constantly refresh the page.

Functional requirements

Based on this conversation, we focus on 3 key features:

- Return all businesses based on a user's location (latitude and longitude pair) and radius.
- Business owners can add, delete or update a business, but this information doesn't need to be reflected in real-time.
- Customers can view detailed information about a business.

Non-functional requirements

From the business requirements, we can infer a list of non-functional requirements. You should also check these with the interviewer.

- Low latency. Users should be able to see nearby businesses quickly.
- Data privacy. Location info is sensitive data. When we design a location-based service (LBS), we should always take user privacy into consideration. We need to comply with data privacy laws like General Data Protection Regulation (GDPR) [4] and California Consumer Privacy Act (CCPA) [5], etc.
- High availability and scalability requirements. We should ensure our system can handle the spike in traffic during peak hours in densely populated areas.

Back-of-the-envelope estimation

Let's take a look at some back-of-the-envelope calculations to determine the potential scale and challenges our solution will need to address. Assume we have 100 million daily active users and 200 million businesses.

Calculate QPS
• Seconds in a day $= 24 \times 60 \times 60 = 86{,}400$. We can round it up to 10^5 for easier calculation. **10^5 is used throughout this book** to represent seconds in a day. • Assume a user makes 5 search queries per day. • Search QPS $= \dfrac{100 \text{ million} \times 5}{10^5} = 5{,}000$

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

In this section, we discuss the following:

- API design
- High-level design
- Algorithms to find nearby businesses
- Data model

API design

We use the RESTful API convention to design a simplified version of the APIs.

`GET /v1/search/nearby`

This endpoint returns businesses based on certain search criteria. In real-life applications, search results are usually paginated. Pagination [6] is not the focus of this chapter, but is worth mentioning during an interview.

Request Parameters:

Field	Description	Type
`latitude`	Latitude of a given location	decimal
`longitude`	Longitude of a given location	decimal
`radius`	Optional. Default is 5000 meters (about 3 miles)	int

Table 1.1: Request parameters

```
{
  "total": 10,
  "businesses":[{business object}]
}
```

The business object contains everything needed to render the search result page, but we may still need additional attributes such as pictures, reviews, star rating, etc., to render

the business detail page. Therefore, when a user clicks on the business detail page, a new endpoint call to fetch the detailed information of a business is usually required.

APIs for a business

The APIs related to a business object are shown in the table below.

API	Detail
`GET /v1/businesses/:id`	Return detailed information about a business
`POST /v1/businesses`	Add a business
`PUT /v1/businesses/:id`	Update details of a business
`DELETE /v1/businesses/:id`	Delete a business

Table 1.2: APIs for a business

If you are interested in real-world APIs for place/business search, two examples are Google Places API [7] and Yelp business endpoints [8].

Data model

In this section, we discuss the read/write ratio and the schema design. The scalability of the database is covered in deep dive.

Read/write ratio

Read volume is high because the following two features are very commonly used:

- Search for nearby businesses.
- View the detailed information of a business.

On the other hand, the write volume is low because adding, removing, and editing business info are infrequent operations.

For a read-heavy system, a relational database such as MySQL can be a good fit. Let's take a closer look at the schema design.

Data schema

The key database tables are the business table and the geospatial (geo) index table.

Business table

The business table contains detailed information about a business. It is shown in Table 1.3 and the primary key is `business_id`.

business	
business_id	PK
address	
city	
state	
country	
latitude	
longtitude	

Table 1.3: Business table

Geo index table

A geo index table is used for the efficient processing of spatial operations. Since this table requires some knowledge about geohash, we will discuss it in the "Scale the database" section on page 24.

High-level design

The high-level design diagram is shown in Figure 1.2. The system comprises two parts: location-based service (LBS) and business-related service. Let's take a look at each component of the system.

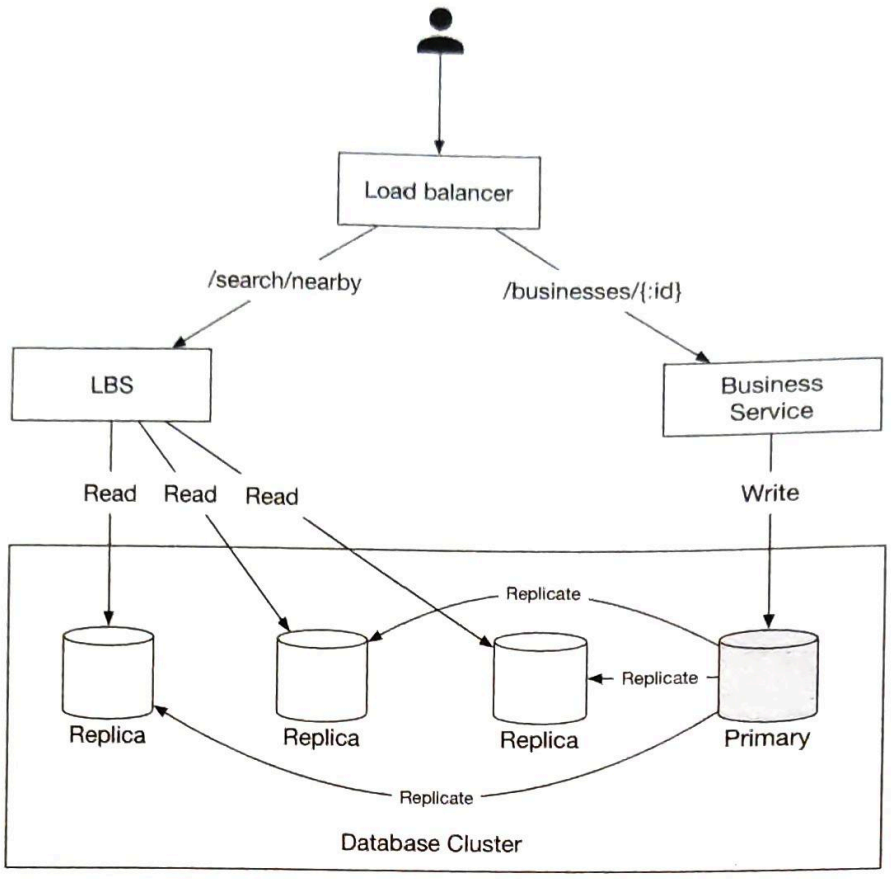


Figure 1.2: High-level design

Load balancer

The load balancer automatically distributes incoming traffic across multiple services. Normally, a company provides a single DNS entry point and internally routes the API calls to the appropriate services based on the URL paths.

Location-based service (LBS)

The LBS service is the core part of the system which finds nearby businesses for a given radius and location. The LBS has the following characteristics:

- It is a read-heavy service with no write requests.
- QPS is high, especially during peak hours in dense areas.
- This service is stateless so it's easy to scale horizontally.

Business service

Business service mainly deals with two types of requests:

- Business owners create, update, or delete businesses. Those requests are mainly write operations, and the QPS is not high.
- Customers view detailed information about a business. QPS is high during peak hours.

Database cluster

The database cluster can use the primary-secondary setup. In this setup, the primary database handles all the write operations, and multiple replicas are used for read operations. Data is saved to the primary database first and then replicated to replicas. Due to the replication delay, there might be some discrepancy between data read by the LBS and the data written to the primary database. This inconsistency is usually not an issue because business information doesn't need to be updated in real-time.

Scalability of business service and LBS

Both the business service and LBS are stateless services, so it's easy to automatically add more servers to accommodate peak traffic (e.g. mealtime) and remove servers during off-peak hours (e.g. sleep time). If the system operates on the cloud, we can set up different regions and availability zones to further improve availability [9]. We discuss this more in the deep dive.

Algorithms to fetch nearby businesses

In real life, companies might use existing geospatial databases such as Geohash in Redis [10] or Postgres with PostGIS extension [11]. You are not expected to know the internals of those geospatial databases during an interview. It's better to demonstrate your problem-solving skills and technical knowledge by explaining how the geospatial index works, rather than to simply throw out database names.

The next step is to explore different options for fetching nearby businesses. We will list a few options, go over the thought process, and discuss trade-offs.

Option 1: Two-dimensional search

The most intuitive but naive way to get nearby businesses is to draw a circle with the predefined radius and find all the businesses within the circle as shown in Figure 1.3.

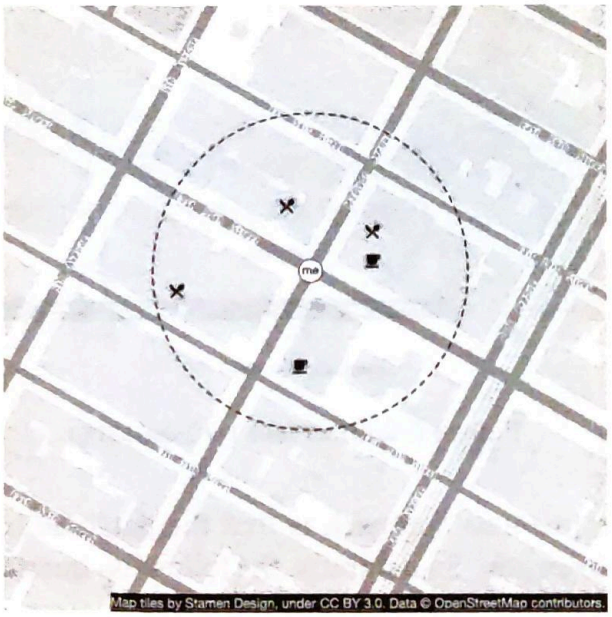


Figure 1.3: Two dimensional search

This process can be translated into the following pseudo SQL query:

```
SELECT business_id, latitude, longitude,
FROM business
WHERE (latitude BETWEEN {:my_lat} - radius AND {:my_lat} + radius)
AND
  (longitude BETWEEN {:my_long} - radius AND {:my_long} + radius)
```

This query is not efficient because we need to scan the whole table. What if we build indexes on longitude and latitude columns? Would this improve the efficiency? The answer is not by much. The problem is that we have two-dimensional data and the dataset returned from each dimension could still be huge. For example, as shown in Figure 1.4, we can quickly retrieve dataset 1 and dataset 2, thanks to indexes on longitude and latitude columns. But to fetch businesses within the radius, we need to perform an intersect operation on those two datasets. This is not efficient because each dataset contains lots of data.

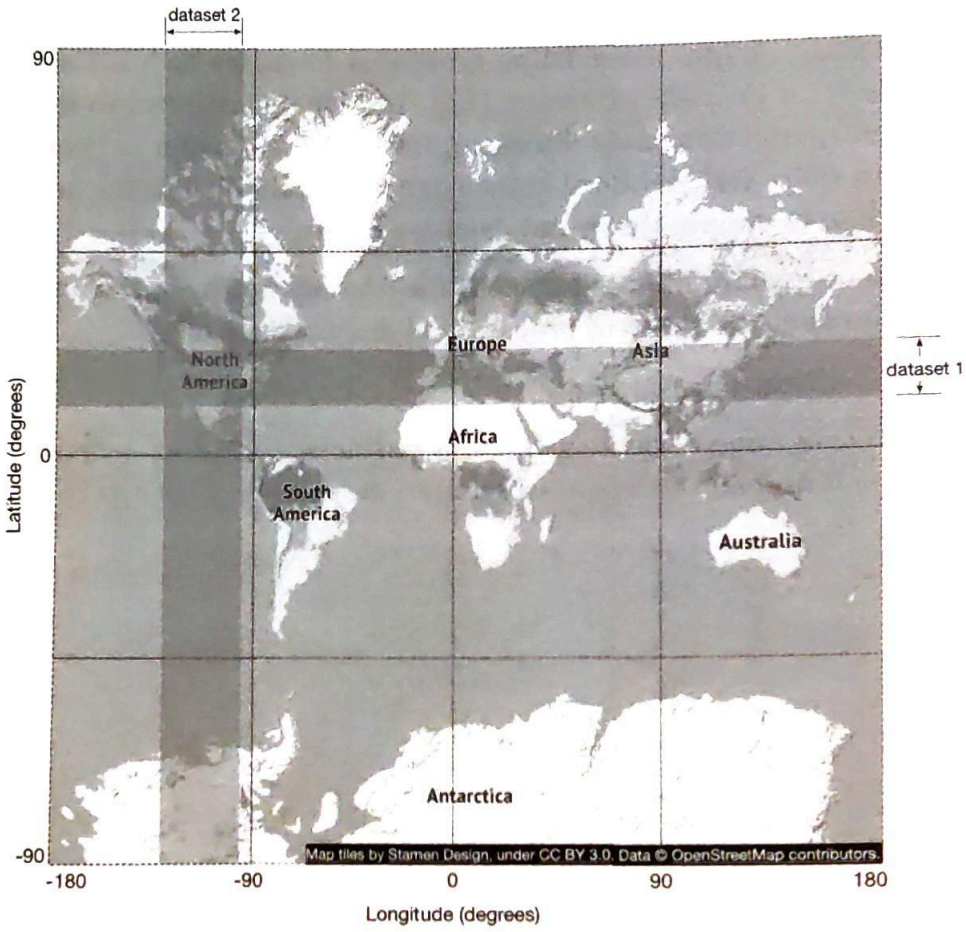


Figure 1.4: Intersect two datasets

The problem with the previous approach is that the database index can only improve search speed in one dimension. So naturally, the follow-up question is, can we map two-dimensional data to one dimension? The answer is yes.

Before we dive into the answers, let's take a look at different types of indexing methods.

In a broad sense, there are two types of geospatial indexing approaches, as shown in Figure 1.5. The highlighted ones are the algorithms we discuss in detail because they are commonly used in the industry.

- Hash: even grid, geohash, cartesian tiers [12], etc.
- Tree: quadtree, Google S2, RTree [13], etc.

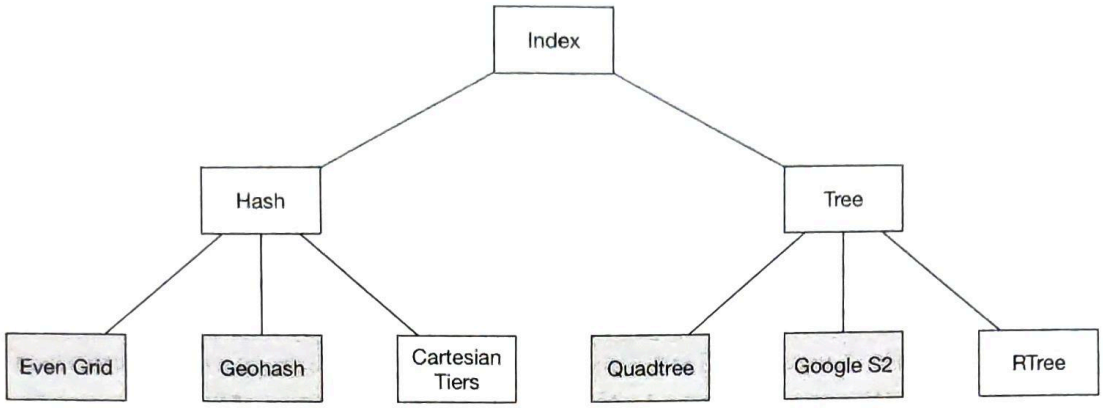


Figure 1.5: Different types of geospatial indexes

Even though the underlying implementations of those approaches are different, the high-level idea is the same, that is, **to divide the map into smaller areas and build indexes for fast search**. Among those, geohash, quadtree, and Google S2 are most widely used in real-world applications. Let's take a look at them one by one.

Reminder
In a real interview, you usually don't need to explain the implementation details of indexing options. However, it is important to have some basic understanding of the need for geospatial indexing, how it works at a high level, and also its limitations.

Option 2: Evenly divided grid

One simple approach is to evenly divide the world into small grids (Figure 1.6). This way, one grid could have multiple businesses, and each business on the map belongs to one grid.

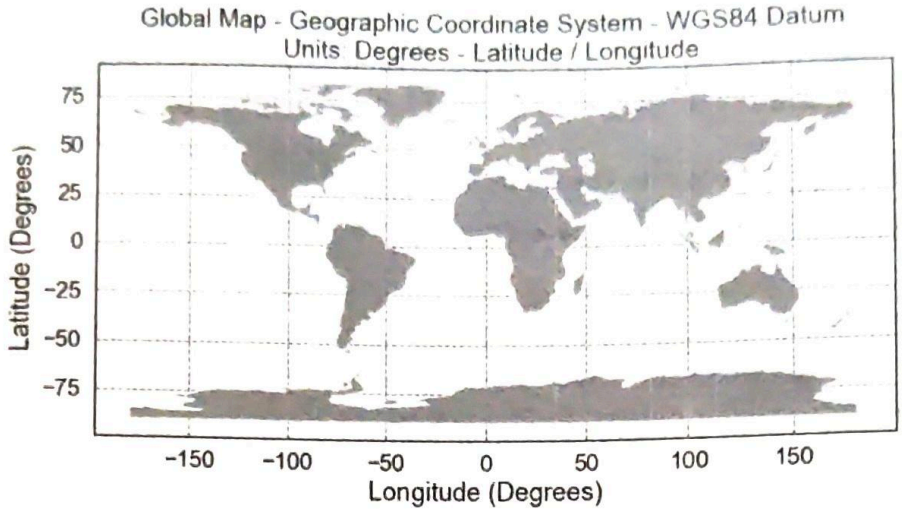


Figure 1.6: Global map (source: [14])

This approach works to some extent, but it has one major issue: the distribution of businesses is not even. There could be lots of businesses in downtown New York, while other grids in deserts or oceans have no business at all. By dividing the world into even grids, we produce a very uneven data distribution. Ideally, we want to use more granular grids for dense areas and large grids in sparse areas. Another potential challenge is to find neighboring grids of a fixed grid.

Option 3: Geohash

Geohash is better than the evenly divided grid option. It works by reducing the two-dimensional longitude and latitude data into a one-dimensional string of letters and digits. Geohash algorithms work by recursively dividing the world into smaller and smaller grids with each additional bit. Let's go over how geohash works at a high level.

First, divide the planet into four quadrants along with the prime meridian and equator.

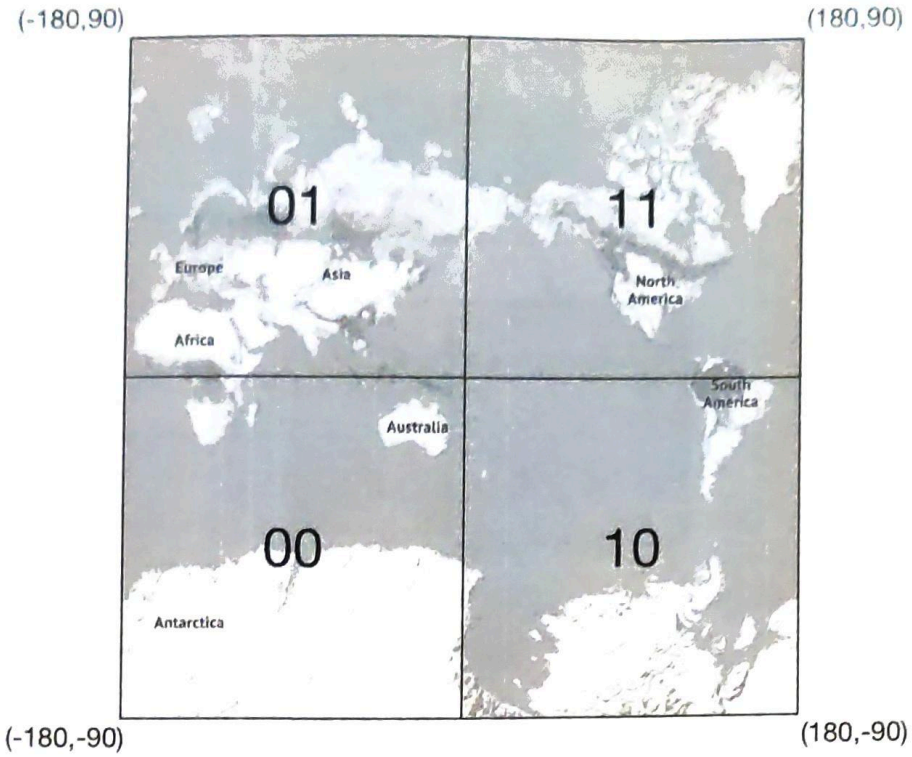


Figure 1.7: Geohash

- Latitude range $[-90, 0]$ is represented by 0
- Latitude range $[0, 90]$ is represented by 1
- Longitude range $[-180, 0]$ is represented by 0
- Longitude range $[0, 180]$ is represented by 1

Second, divide each grid into four smaller grids. Each grid can be represented by alternating between longitude bit and latitude bit.

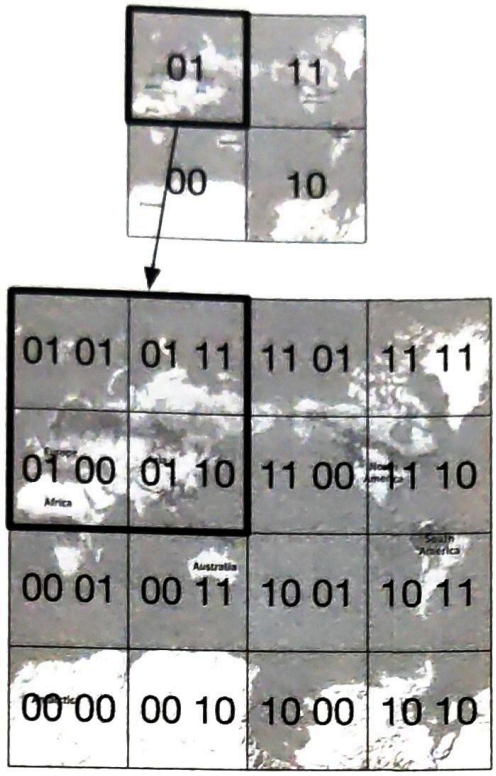


Figure 1.8: Divide grid

Repeat this subdivision until the grid size is within the precision desired. Geohash usually uses base32 representation [15]. Let's take a look at two examples.

- geohash of the Google headquarter (length = 6):
  `1001 10110 01001 10000 11011 11010` (base32 in binary) →
  `9q9hvu` (base32)
- geohash of the Facebook headquarter (length = 6):
  `1001 10110 01001 10001 10000 10111` (base32 in binary) →
  `9q9jhr` (base32)

Geohash has 12 precisions (also called levels) as shown in Table 1.4. The precision factor determines the size of the grid. We are only interested in geohashes with lengths between 4 and 6. This is because when it's longer than 6, the grid size is too small, while if it is smaller than 4, the grid size is too large (see Table 1.4).

geohash length	Grid width × height
1	5,009.4km × 4,992.6km (the size of the planet)
2	1,252.3km × 624.1km
3	156.5km × 156km
4	39.1km × 19.5km
5	4.9km × 4.9km
6	1.2km × 609.4m
7	152.9m × 152.4m
8	38.2m × 19m
9	4.8m × 4.8m
10	1.2m × 59.5cm
11	14.9cm × 14.9cm
12	3.7cm × 1.9cm

Table 1.4: Geohash length to grid size mapping (source: [16])

How do we choose the right precision? We want to find the minimal geohash length that covers the whole circle drawn by the user-defined radius. The corresponding relationship between the radius and the length of geohash is shown in the table below.

Radius (Kilometers)	Geohash length
0.5km (0.31 mile)	6
1km (0.62 mile)	5
2km (1.24 mile)	5
5km (3.1 mile)	4
20km (12.42 mile)	4

Table 1.5: Radius to geohash mapping

This approach works great most of the time, but there are some edge cases with how the geohash boundary is handled that we should discuss with the interviewer.

Boundary issues

Geohashing guarantees that the longer a shared prefix is between two geohashes, the closer they are. As shown in Figure 1.9, all the grids have a shared prefix: `9q8zn`.

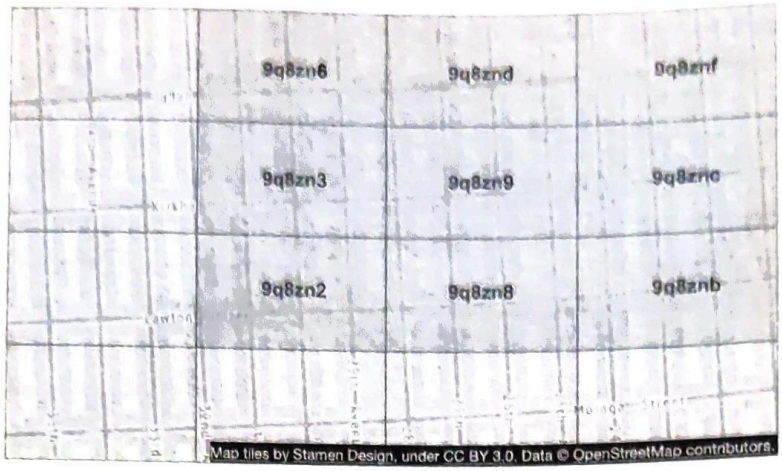


Figure 1.9: Shared prefix

Boundary issue 1

However, the reverse is not true: two locations can be very close but have no shared prefix at all. This is because two close locations on either side of the equator or prime meridian belong to different "halves" of the world. For example, in France, La Roche-Chalais (geohash: `u000`) is just 30km from Pomerol (geohash: `ezzz`) but their geohashes have no shared prefix at all [17].

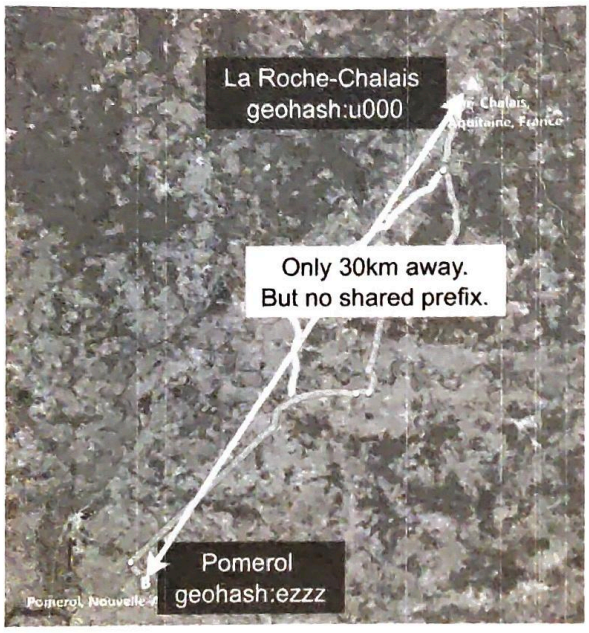


Figure 1.10: No shared prefix

Because of this boundary issue, a simple prefix SQL query below would fail to fetch all nearby businesses.

```
SELECT * FROM geohash_index WHERE geohash LIKE '9q8zn%'
```

Boundary issue 2

Another boundary issue is that two positions can have a long shared prefix, but they belong to different geohashes as shown in Figure 1.11.

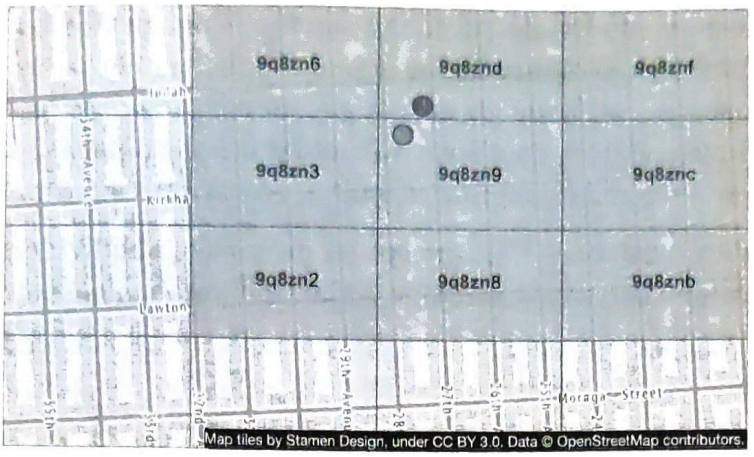


Figure 1.11: Boundary issue

A common solution is to fetch all businesses not only within the current grid but also from its neighbors. The geohashes of neighbors can be calculated in constant time and more details about this can be found here [17].

Not enough businesses

Now let's tackle the bonus question. What should we do if there are not enough businesses returned from the current grid and all the neighbors combined?

Option 1: only return businesses within the radius. This option is easy to implement, but the drawback is obvious. It doesn't return enough results to satisfy a user's needs.

Option 2: increase the search radius. We can remove the last digit of the geohash and use the new geohash to fetch nearby businesses. If there are not enough businesses, we continue to expand the scope by removing another digit. This way, the grid size is gradually expanded until the result is greater than the desired number of results. Figure 1.12 shows the conceptual diagram of the expanding search process.

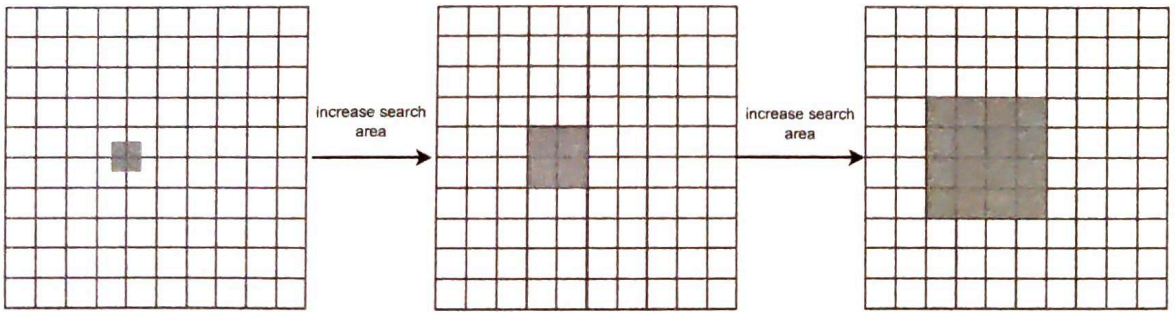


Figure 1.12: Expand the search process

Option 4: Quadtree

Another popular solution is quadtree. A quadtree [18] is a data structure that is commonly used to partition a two-dimensional space by recursively subdividing it into four quadrants (grids) until the contents of the grids meet certain criteria. For example, the criterion can be to keep subdividing until the number of businesses in the grid is not more than 100. This number is arbitrary as the actual number can be determined by business needs. With a quadtree, we build an in-memory tree structure to answer queries. Note that quadtree is an in-memory data structure and it is not a database solution. It runs on each LBS server, and the data structure is built at server start-up time.

The following figure visualizes the conceptual process of subdividing the world into a quadtree. Let's assume the world contains 200m (million) businesses.

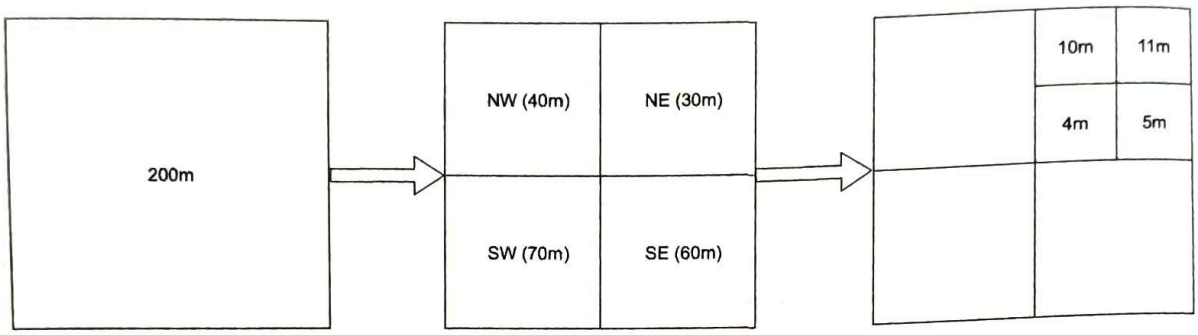


Figure 1.13: Quadtree

Figure 1.14 explains the quadtree building process in more detail. The root node represents the whole world map. The root node is recursively broken down into 4 quadrants until no nodes are left with more than 100 businesses.

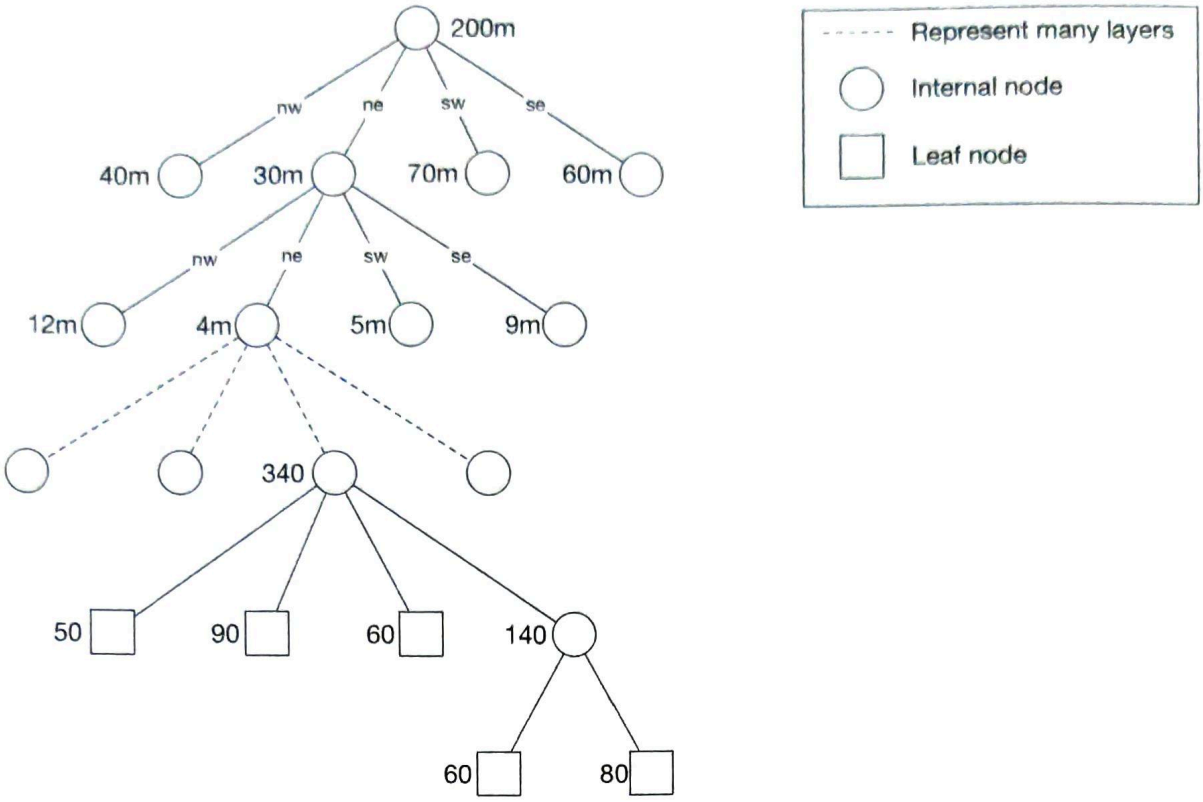


Figure 1.14: Build quadtree

The pseudocode for building quadtree is shown below:

```
public void buildQuadtree(TreeNode node) {
  if (countNumberOfBusinessesInCurrentGrid(node) > 100) {
    node.subdivide();
    for (TreeNode child : node.getChildren()) {
      buildQuadtree(child);
    }
  }
}
```

How much memory does it need to store the whole quadtree?

To answer this question, we need to know what kind of data is stored.

Data on a leaf node

Name	Size
Top left coordinates and bottom-right coordinates to identify the grid	32 bytes (8 bytes ×4)
List of business IDs in the grid	8 bytes per ID × 100 (maximal number of businesses allowed in one grid)
Total	832 bytes

Table 1.6: Leaf node

Data on internal node

Name	Size
Top left coordinates and bottom-right coordinates to identify the grid	32 bytes (8 bytes $\times 4$)
Pointers to 4 children	32 bytes (8 bytes $\times 4$)
Total	64 bytes

Table 1.7: Internal node

Even though the tree-building process depends on the number of businesses within a grid, this number does not need to be stored in the quadtree node because it can be inferred from records in the database.

Now that we know the data structure for each node, let's take a look at the memory usage.

- Each grid can store a maximal of 100 businesses
- Number of leaf nodes $=\sim \frac{200 \text{ million}}{100} =\sim 2$ million
- Number of internal nodes $= 2 \text{ million} \times \frac{1}{3} =\sim 0.67$ million. If you do not know why the number of internal nodes is one-third of the leaf nodes, please read the reference material [19].
- Total memory requirement $= 2 \text{ million} \times 832 \text{ bytes} + 0.67 \text{ million} \times 64 \text{ bytes} = \sim 1.71\text{GB}$. Even if we add some overhead to build the tree, the memory requirement to build the tree is quite small.

In a real interview, we shouldn't need such detailed calculations. The key takeaway here is that the quadtree index doesn't take too much memory and can easily fit in one server. Does it mean we should use only one server to store the quadtree index? The answer is no. Depending on the read volume, a single quadtree server might not have enough CPU or network bandwidth to serve all read requests. If that is the case, it will be necessary to spread the read load among multiple quadtree servers.

How long does it take to build the whole quadtree?

Each leaf node contains approximately 100 business IDs. The time complexity to build the tree is $\frac{n}{100} \log \frac{n}{100}$, where n is the total number of businesses. It might take a few minutes to build the whole quadtree with 200 million businesses.

How to get nearby businesses with quadtree?

1. Build the quadtree in memory.
2. After the quadtree is built, start searching from the root and traverse the tree, until we find the leaf node where the search origin is. If that leaf node has 100 businesses, return the node. Otherwise, add businesses from its neighbors until enough businesses are returned.

Operational considerations for quadtree

As mentioned above, it may take a few minutes to build a quadtree with 200 million businesses at the server start-up time. It is important to consider the operational implications of such a long server start-up time. While the quadtree is being built, the server cannot serve traffic. Therefore, we should roll out a new release of the server incrementally to a small subset of servers at a time. This avoids taking a large swath of the server cluster offline and causes service brownout. Blue/green deployment [20] can also be used, but an entire cluster of new servers fetching 200 million businesses at the same time from the database service can put a lot of strain on the system. This can be done, but it may complicate the design and you should mention that in the interview.

Another operational consideration is how to update the quadtree as businesses are added and removed over time. The easiest approach would be to incrementally rebuild the quadtree, a small subset of servers at a time, across the entire cluster. But this would mean some servers would return stale data for a short period of time. However, this is generally an acceptable compromise based on the requirements. This can be further mitigated by setting up a business agreement that newly added/updated businesses will only be effective the next day. This means we can update the cache using a nightly job. One potential problem with this approach is that tons of keys will be invalidated at the same time, causing heavy load on cache servers.

It's also possible to update the quadtree on the fly as businesses are added and removed. This certainly complicates the design, especially if the quadtree data structure could be accessed by multiple threads. This will require some locking mechanism which could dramatically complicate the quadtree implementation.

Real-world quadtree example

Yext [21] provided an image (Figure 1.15) that shows a constructed quadtree near Denver [21]. We want smaller, more granular grids for dense areas and larger grids for sparse areas.

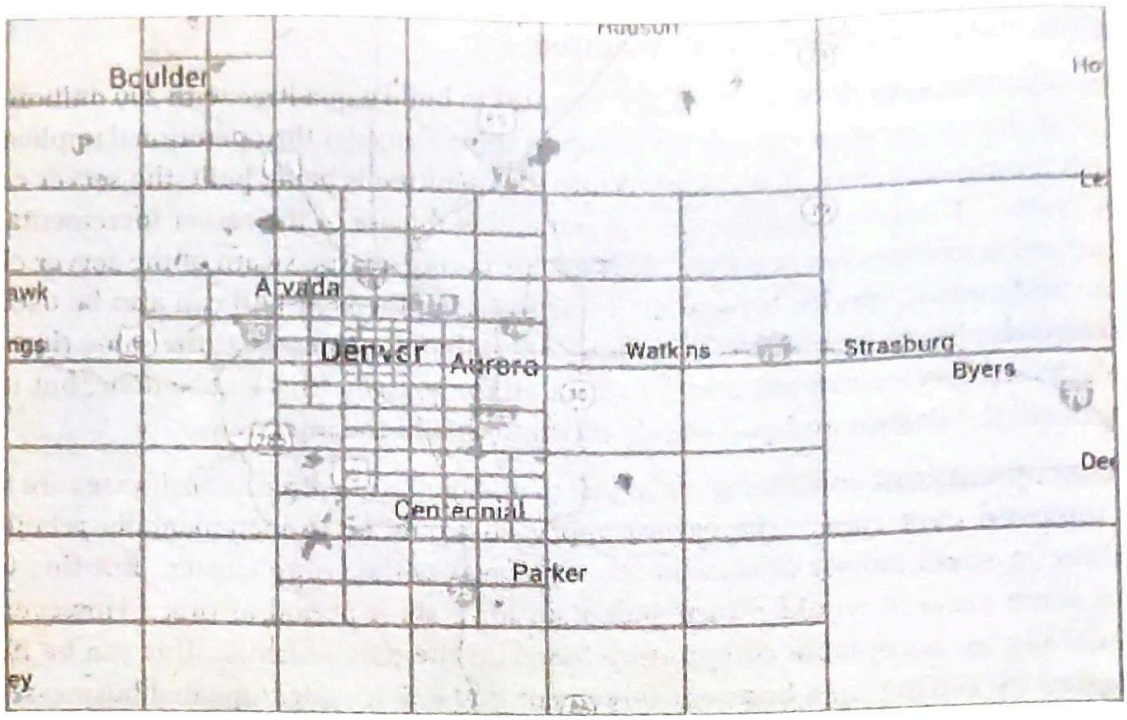


Figure 1.15: Real-world example of a quadtree

Option 5: Google S2

Google S2 geometry library [22] is another big player in this field. Similar to Quadtree, it is an in-memory solution. It maps a sphere to a 1D index based on the Hilbert curve (a space-filling curve) [23]. The Hilbert curve has a very important property: two points that are close to each other on the Hilbert curve are close in 1D space (Figure 1.16). Search on 1D space is much more efficient than on 2D. Interested readers can play with an online tool [24] for the Hilbert curve.

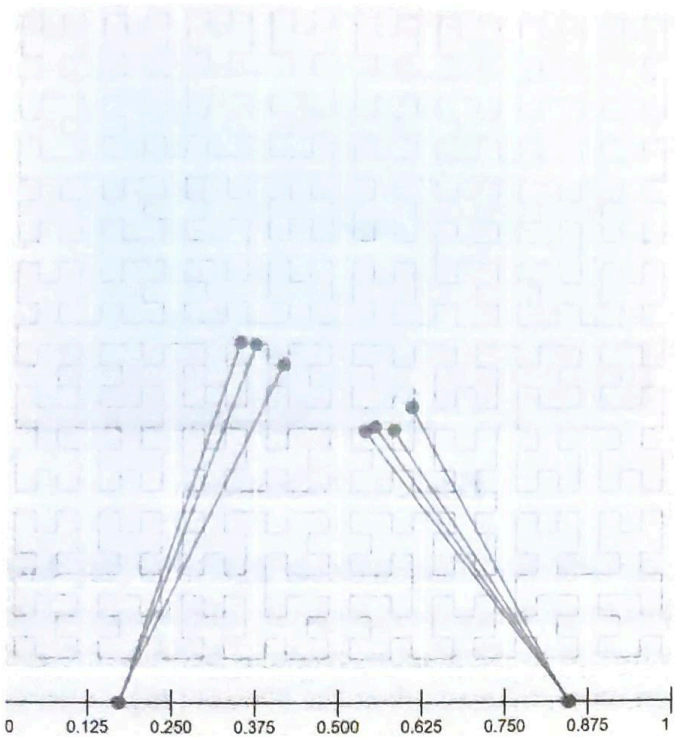


Figure 1.16: Hilbert curve (source: [24])

S2 is a complicated library and you are not expected to explain its internals during an interview. But because it's widely used in companies such as Google, Tinder, etc., we will briefly cover its advantages.

- S2 is great for geofencing because it can cover arbitrary areas with varying levels (Figure 1.17). According to Wikipedia, "A geofence is a virtual perimeter for a real-world geographic area. A geo-fence could be dynamically generated—as in a radius around a point location, or a geo-fence can be a predefined set of boundaries (such as school zones or neighborhood boundaries)" [25].

  Geofencing allows us to define perimeters that surround the areas of interest and to send notifications to users who are out of the areas. This can provide richer functionalities than just returning nearby businesses.

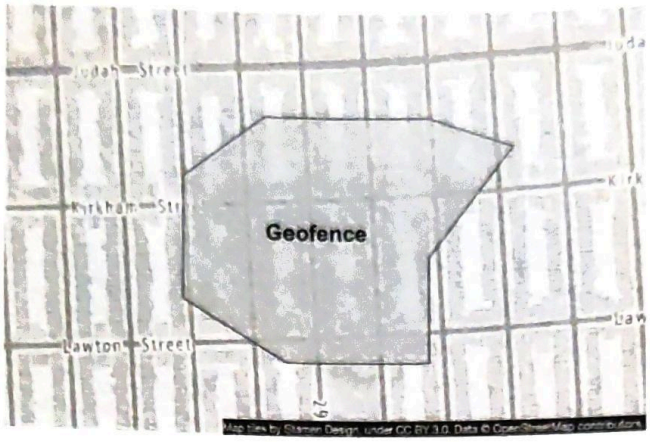


Figure 1.17: Geofence

- Another advantage of S2 is its Region Cover algorithm [26]. Instead of having a fixed level (precision) as in geohash, we can specify min level, max level, and max cells in S2. The result returned by S2 is more granular because the cell sizes are flexible. If you want to learn more, take a look at the S2 tool [26].

Recommendation

To find nearby businesses efficiently, we have discussed a few options: geohash, quadtree and S2. As you can see from Table 1.8, different companies or technologies adopt different options.

Geo Index	Companies
geohash	Bing map [27], Redis [10], MongoDB [28], Lyft [29]
quadtree	Yext [21]
Both geohash and quadtree	Elasticsearch [30]
S2	Google Maps, Tinder [31]

Table 1.8: Different types of geo indexes

During an interview, we suggest choosing **geohash or quadtree** because S2 is more complicated to explain clearly in an interview.

Geohash vs quadtree

Before we conclude this section, let's do a quick comparison between geohash and quadtree.

Geohash

- Easy to use and implement. No need to build a tree.
- Supports returning businesses within a specified radius.
- When the precision (level) of geohash is fixed, the size of the grid is fixed as well. It cannot dynamically adjust the grid size, based on population density. More complex logic is needed to support this.
- Updating the index is easy. For example, to remove a business from the index,

we just need to remove it from the corresponding row with the same geohash and business_id. See Figure 1.18 for a concrete example.

geohash	business_id
9q8zn	3
~~9q8zn~~	~~8~~
9q8zn	4

Figure 1.18: Remove a business

Quadtree

- Slightly harder to implement because it needs to build the tree.
- Supports fetching k-nearest businesses. Sometimes we just want to return k-nearest businesses and don't care if businesses are within a specified radius. For example, when you are traveling and your car is low on gas, you just want to find the nearest k gas stations. These gas stations may not be near you, but the app needs to return the nearest k results. For this type of query, a quadtree is a good fit because its subdividing process is based on the number k and it can automatically adjust the query range until it returns k results.
- It can dynamically adjust the grid size based on population density (see the Denver example in Figure 1.15).
- Updating the index is more complicated than geohash. A quadtree is a tree structure. If a business is removed, we need to traverse from the root to the leaf node, to remove the business. For example, if we want to remove the business with ID $= 2$, we have to travel from the root all the way down to the leaf node, as shown in Figure 1.19. Updating the index takes $O(\log n)$, but the implementation is complicated if the data structure is accessed by a multi-threaded program, as locking is required. Also, rebalancing the tree can be complicated. Rebalancing is necessary if, for example, a leaf node has no room for a new addition. A possible fix is to over-allocate the ranges.

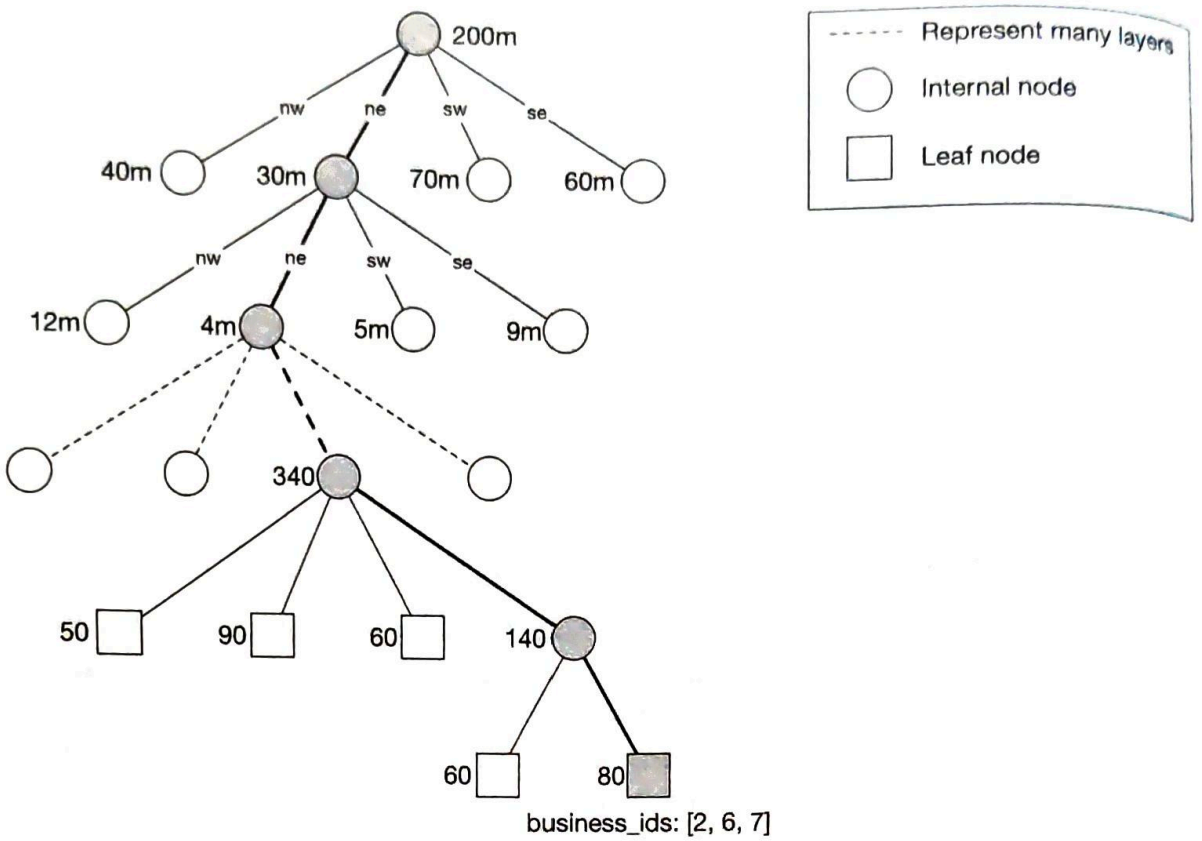


Figure 1.19: Update quadtree

Step 3 - Design Deep Dive

By now you should have a good picture of what the overall system looks like. Now let's dive deeper into a few areas.

- Scale the database
- Caching
- Region and availability zones
- Filter results by time or business type
- Final architecture diagram

Scale the database

We will discuss how to scale two of the most important tables: the business table and the geospatial index table.

Business table

The data for the business table may not all fit in one server, so it is a good candidate for sharding. The easiest approach is to shard everything by business ID. This sharding scheme ensures that load is evenly distributed among all the shards, and operationally it is easy to maintain.

Geospatial index table

Both geohash and quadtree are widely used. Due to geohash's simplicity, we use it as an example. There are two ways to structure the table.

Option 1: For each geohash key, there is a JSON array of business IDs in a single row. This means all business IDs within a geohash are stored in one row.

geospatial_index
geohash
list_of_business_ids

Table 1.9: `list_of_business_ids` is a JSON array

Option 2: If there are multiple businesses in the same geohash, there will be multiple rows, one for each business. This means different business IDs within a geohash are stored in different rows.

geospatial_index
geohash
business_id

Table 1.10: `business_id` is a single ID

Here are some sample rows for option 2.

geohash	business_id
32feac	343
32feac	347
f31cad	112
f31cad	113

Table 1.11: Sample rows of the geospatial index table

Recommendation: we recommend option 2 because of the following reasons:

For option 1, to update a business, we need to fetch the array of `business_ids` and scan the whole array to find the business to update. When inserting a new business, we have to scan the entire array to make sure there is no duplicate. We also need to lock the row to prevent concurrent updates. There are a lot of edge cases to handle.

For option 2, if we have two columns with a compound key of (`geohash, business_id`), the addition and removal of a business are very simple. There would be no need to lock anything.

Scale the geospatial index

One common mistake about scaling the geospatial index is to quickly jump to a sharding scheme without considering the actual data size of the table. In our case, the full dataset

for the geospatial index table is not large (quadtree index only takes 1.71G memory and storage requirement for geohash index is similar). The whole geospatial index can easily fit in the working set of a modern database server. However, depending on the read volume, a single database server might not have enough CPU or network bandwidth to handle all read requests. If that is the case, it is necessary to spread the read load among multiple database servers.

There are two general approaches for spreading the load of a relational database server. We can add read replicas, or shard the database.

Many engineers like to talk about sharding during interviews. However, it might not be a good fit for the geohash table as sharding is complicated. For instance, the sharding logic has to be added to the application layer. Sometimes, sharding is the only option. In this case, though, everything can fit in the working set of a database server, so there is no strong technical reason to shard the data among multiple servers.

A better approach, in this case, is to have a series of read replicas to help with the read load. This method is much simpler to develop and maintain. For this reason, scaling the geospatial index table through replicas is recommended.

Caching

Before introducing a cache layer we have to ask ourselves, do we really need a cache layer?

It is not immediately obvious that caching is a solid win:

- The workload is read-heavy, and the dataset is relatively small. The data could fit in the working set of any modern database server. Therefore, the queries are not I/O bound and they should run almost as fast as an in-memory cache.
- If read performance is a bottleneck, we can add database read replicas to improve the read throughput.

Be mindful when discussing caching with the interviewer, as it will require careful benchmarking and cost analysis. If you find out that caching does fit the business requirements, then you can proceed with discussions about caching strategy.

Cache key

The most straightforward cache key choice is the location coordinates (latitude and longitude) of the user. However, this choice has a few issues:

- Location coordinates returned from mobile phones are not accurate as they are just the best estimation [32]. Even if you don't move, the results might be slightly different each time you fetch coordinates on your phone.
- A user can move from one location to another, causing location coordinates to change slightly. For most applications, this change is not meaningful.

Therefore, location coordinates are not a good cache key. Ideally, small changes in loca-

tion should still map to the same cache key. The geohash/quadtree solution mentioned earlier handles this problem well because all businesses within a grid map to the same geohash.

Types of data to cache

As shown in Table 1.12, there are two types of data that can be cached to improve the overall performance of the system:

Key	Value
geohash	List of business IDs in the grid
business_id	Business object

Table 1.12: Key-value pairs in cache

List of business IDs in a grid

Since business data is relatively stable, we precompute the list of business IDs for a given geohash and store it in a key-value store such as Redis. Let's take a look at a concrete example of getting nearby businesses with caching enabled.

1. Get the list of business IDs for a given geohash.

```
SELECT business_id FROM geohash_index WHERE geohash LIKE `{:
geohash}%`
```

2. Store the result in the Redis cache if cache misses.

```
public List<String> getNearbyBusinessIds(String geohash) {
  String cacheKey = hash(geohash);
  List<string> listOfBusinessIds = Redis.get(cacheKey);
  if (listOfBusinessIDs == null) {
    listOfBusinessIds = Run the select SQL query above;
    Cache.set(cacheKey, listOfBusinessIds, "1d");
  }
  return listOfBusinessIds;
}
```

When a new business is added, edited, or deleted, the database is updated and the cache invalidated. Since the volume of those operations is relatively small and no locking mechanism is needed for the geohash approach, update operations are easy to deal with.

According to the requirements, a user can choose the following 4 radii on the client: 500m, 1km, 2km, and 5km. Those radii are mapped to geohash lengths of 4, 5, 5, and 6, respectively. To quickly fetch nearby businesses for different radii, we cache data in Redis on all three precisions (geohash_4, geohash_5, and geohash_6).

As mentioned earlier, we have 200 million businesses and each business belongs to 1 grid in a given precision. Therefore the total memory required is:

- Storage for Redis values: 8 bytes $\times$ 200 million $\times$ 3 precisions = $\sim$ 5GB
- Storage for Redis keys: negligible

- Total memory required: $\sim$ 5GB

We can get away with one modern Redis server from the memory usage perspective, but to ensure high availability and reduce cross continent latency, we deploy the Redis cluster across the globe. Given the estimated data size, we can have the same copy of cache data deployed globally. We call this Redis cache "Geohash" in our final architecture diagram (Figure 1.21).

Business data needed to render pages on the client

This type of data is quite straightforward to cache. The key is the `business_id` and the value is the business object which contains the business name, address, image URLs, etc. We call this Redis cache "Business info" in our final architecture diagram (Figure 1.21).

Region and availability zones

We deploy a location-based service to multiple regions and availability zones as shown in Figure 1.20. This has a few advantages:

- Makes users physically "closer" to the system. Users from the US West are connected to the data centers in that region, and users from Europe are connected with data centers in Europe.
- Gives us the flexibility to spread the traffic evenly across the population. Some regions such as Japan and Korea have high population densities. It might be wise to put them in separate regions, or even deploy location-based services in multiple availability zones to spread the load.
- Privacy laws. Some countries may require user data to be used and stored locally. In this case, we could set up a region in that country and employ DNS routing to restrict all requests from the country to only that region.

Figure 1.20: Deploy LBS "closer" to the user

Follow-up question: filter results by time or business type

The interviewer might ask a follow-up question: how to return businesses that are open now, or only return businesses that are restaurants?

Candidate: When the world is divided into small grids with geohash or quadtree, the number of businesses returned from the search result is relatively small. Therefore, it is acceptable to return business IDs first, hydrate business objects, and filter them based on opening time or business type. This solution assumes opening time and business type are stored in the business table.

Final design diagram

Putting everything together, we come up with the following design diagram.

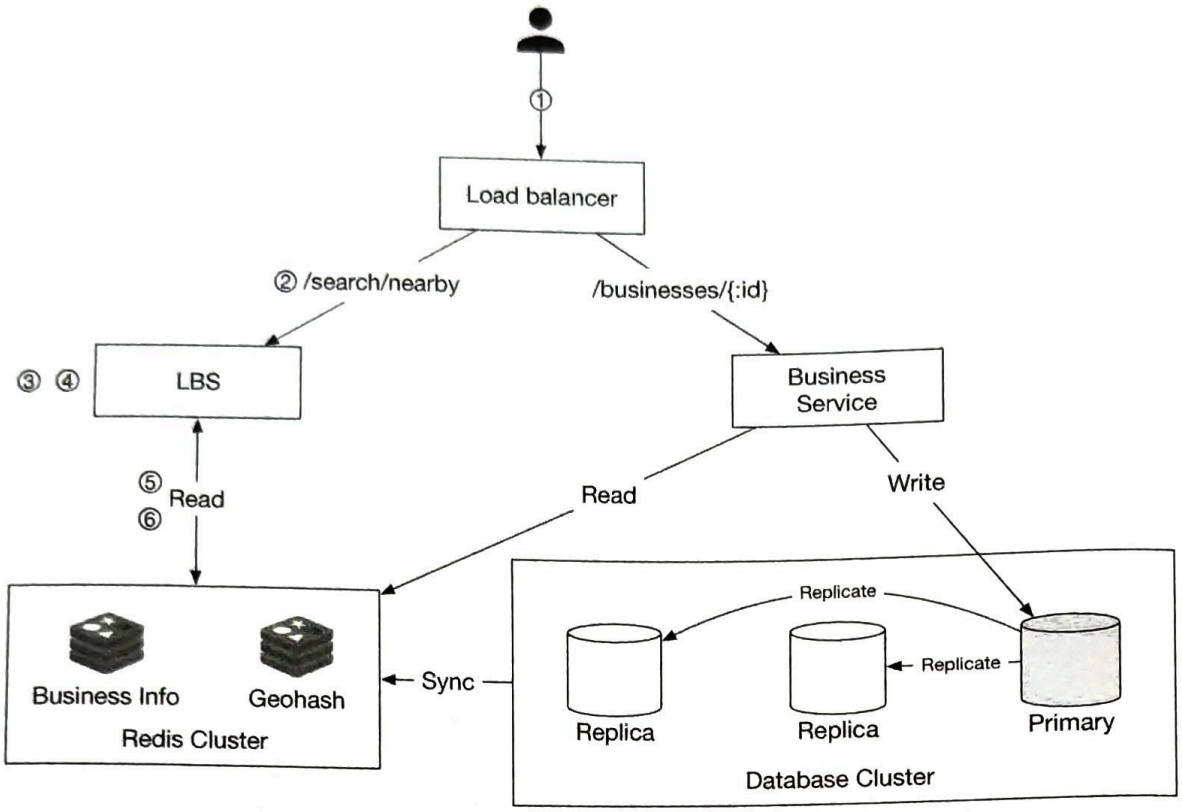


Figure 1.21: Design diagram

Get nearby businesses

1. You try to find restaurants within 500 meters on Yelp. The client sends the user location (latitude = 37.776720, longitude = −122.416730) and radius (500m) to the load balancer.
2. The load balancer forwards the request to the LBS.
3. Based on the user location and radius info, the LBS finds the geohash length that matches the search. By checking Table 1.5, 500m map to geohash length = 6.
4. LBS calculates neighboring geohashes and adds them to the list. The result looks like this:
   `list_of_geohashes = [my_geohash, neighbor1_geohash, neighbor2_geohash, ..., neighbor8_geohash]`.
5. For each geohash in `list_of_geohashes`, LBS calls the "Geohash" Redis server to fetch corresponding business IDs. Calls to fetch business IDs for each geohash can be made in parallel to reduce latency.
6. Based on the list of business IDs returned, LBS fetches fully hydrated business information from the "Business info" Redis server, then calculates distances between a user and businesses, ranks them, and returns the result to the client.

View, update, add or delete a business

All business-related APIs are separated from the LBS. To view the detailed information about a business, the business service first checks if the data is stored in the "Business info" Redis cache. If it is, cached data will be returned to the client. If not, data is fetched from the database cluster and then stored in the Redis cache, allowing subsequent requests to get results from the cache directly.

Since we have an upfront business agreement that newly added/updated businesses will be effective the next day, cached business data is updated by a nightly job.

Step 4 - Wrap Up

In this chapter, we have presented the design for proximity service. The system is a typical LBS that leverages geospatial indexing. We discussed several indexing options:

- Two-dimensional search
- Evenly divided grid
- Geohash
- Quadtree
- Google S2

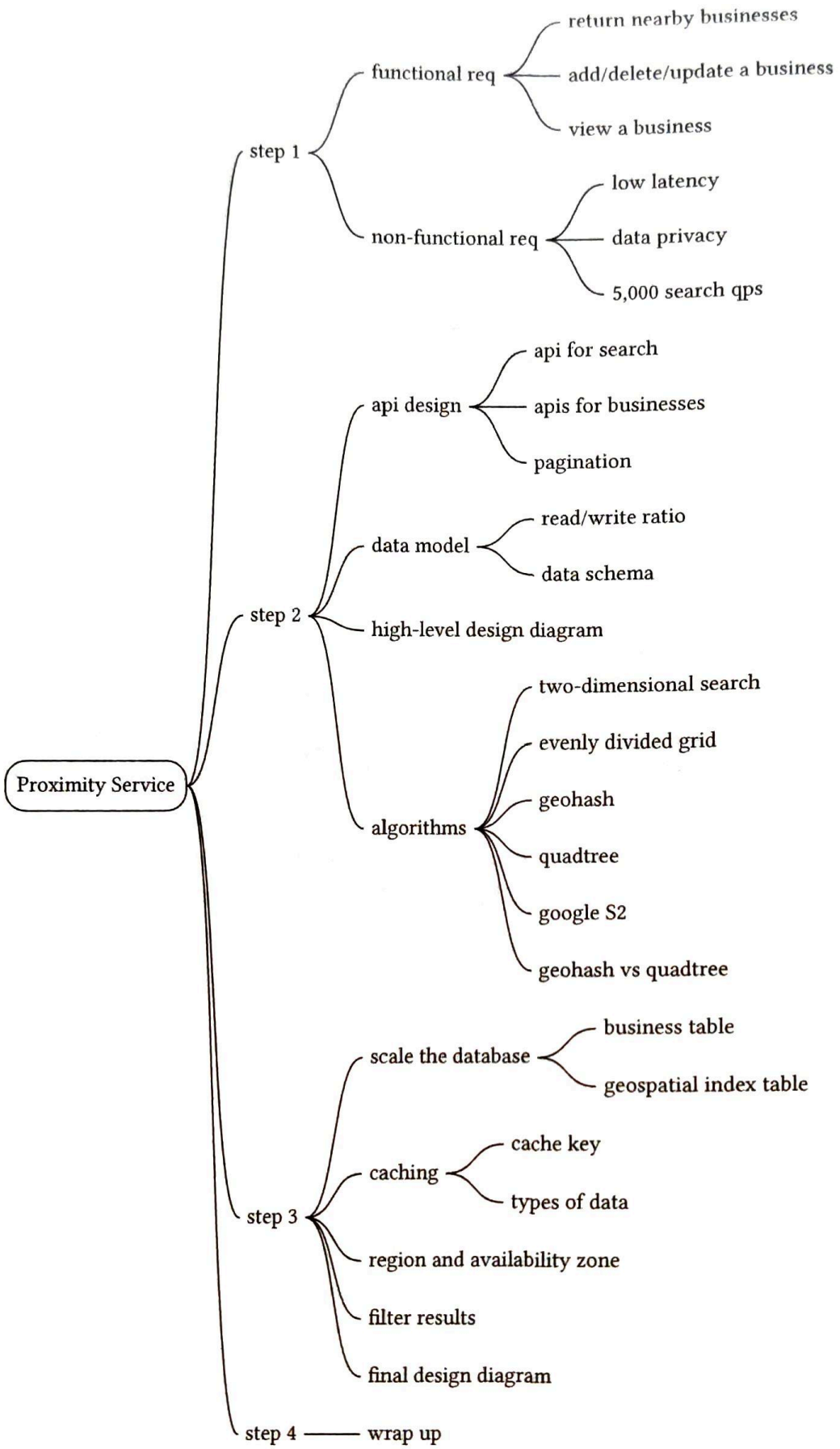
Geohash, quadtree, and S2 are widely used by different tech companies. We choose geohash as an example to show how a geospatial index works.

In the deep dive, we discussed why caching is effective in reducing the latency, what should be cached and how to use cache to retrieve nearby businesses fast. We also discussed how to scale the database with replication and sharding.

We then looked at deploying LBS in different regions and availability zones to improve availability, to make users physically closer to the servers, and to comply better with local privacy laws.

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

Chapter Summary



Reference Material

[1] Yelp. https://www.yelp.com/.

[2] Map tiles by Stamen Design. http://maps.stamen.com/.

[3] OpenStreetMap. https://www.openstreetmap.org.

[4] GDPR. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation.

[5] CCPA. https://en.wikipedia.org/wiki/California_Consumer_Privacy_Act.

[6] Pagination in the REST API. https://developer.atlassian.com/server/confluence/pagination-in-the-rest-api/.

[7] Google places API. https://developers.google.com/maps/documentation/places/web-service/search.

[8] Yelp business endpoints. https://www.yelp.com/developers/documentation/v3/business_search.

[9] Regions and Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[10] Redis GEOHASH. https://redis.io/commands/GEOHASH.

[11] POSTGIS. https://postgis.net/.

[12] Cartesian tiers. http://www.nsshutdown.com/projects/lucene/whitepaper/localluc ene_v2.html.

[13] R-tree. https://en.wikipedia.org/wiki/R-tree.

[14] Global map in a Geographic Coordinate Reference System. https://bit.ly/3DsjAwg.

[15] Base32. https://en.wikipedia.org/wiki/Base32.

[16] Geohash grid aggregation. https://bit.ly/3kKl4e6.

[17] Geohash. https://www.movable-type.co.uk/scripts/geohash.html.

[18] Quadtree. https://en.wikipedia.org/wiki/Quadtree.

[19] How many leaves has a quadtree. https://stackoverflow.com/questions/35976444/how-many-leaves-has-a-quadtree.

[20] Blue green deployment. https://martinfowler.com/bliki/BlueGreenDeployment.html.

[21] Improved Location Caching with Quadtrees. https://engblog.yext.com/post/geolocation-caching.

[22] S2. https://s2geometry.io/.

[23] Hilbert curve. https://en.wikipedia.org/wiki/Hilbert_curve.

[24] Hilbert mapping. http://bit-player.org/extras/hilbert/hilbert-mapping.html.

[25] Geo-fence. https://en.wikipedia.org/wiki/Geo-fence.

[26] Region cover. https://s2.sidewalklabs.com/regioncoverer/.

[27] Bing map. https://bit.ly/30ytSfG.

[28] MongoDB. https://docs.mongodb.com/manual/tutorial/build-a-2d-index/.

[29] Geospatial Indexing: The 10 Million QPS Redis Architecture Powering Lyft. https://www.youtube.com/watch?v=cSFWlF96Sds&t=2155s.

[30] Geo Shape Type. https://www.elastic.co/guide/en/elasticsearch/reference/1.6/mapping-geo-shape-type.html.

[31] Geosharded Recommendations Part 1: Sharding Approach. https://medium.com/tinder-engineering/geosharded-recommendations-part-1-sharding-approach-d5d54e0ec77a.

[32] Get the last known location. https://developer.android.com/training/location/retrieve-current#Challenges.

2 Nearby Friends

In this chapter, we design a scalable backend system for a new mobile app feature called "Nearby Friends". For an opt-in user who grants permission to access their location, the mobile client presents a list of friends who are geographically nearby. If you are looking for a real-world example, please refer to this article [1] about a similar feature in the Facebook app.

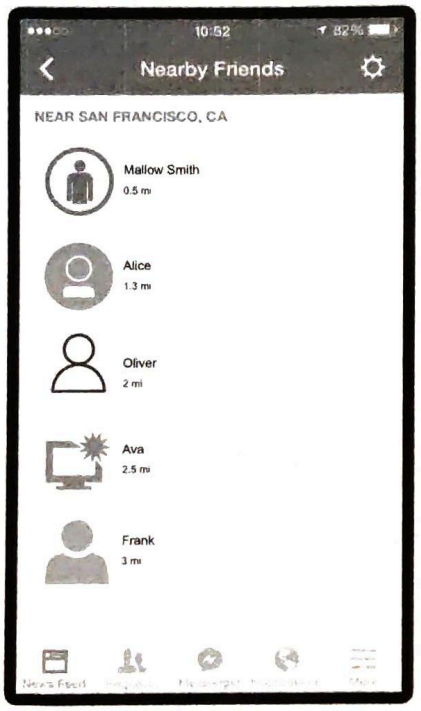


Figure 2.1: Facebook's nearby friends

If you read Chapter 1 Proximity Service, you may wonder why we need a separate chapter for designing "nearby friends" since it looks similar to proximity services. If you think carefully though, you will find major differences. In proximity services, the addresses for businesses are static as their locations do not change, while in "nearby friends", data is more dynamic because user locations change frequently.

Step 1 - Understand the Problem and Establish Design Scope

Any backend system at the Facebook scale is complicated. Before starting with the design, we need to ask clarification questions to narrow down the scope.

Candidate: How geographically close is considered to be "nearby"?
Interviewer: 5 miles. This number should be configurable.

Candidate: Can I assume the distance is calculated as the straight-line distance between two users? In real life, there could be, for example, a river in between the users, resulting in a longer travel distance.
Interviewer: Yes, that's a reasonable assumption.

Candidate: How many users does the app have? Can I assume 1 billion users and 10% of them use the nearby friends feature?
Interviewer: Yes, that's a reasonable assumption.

Candidate: Do we need to store location history?
Interviewer: Yes, location history can be valuable for different purposes such as machine learning.

Candidate: Could we assume if a friend is inactive for more than 10 minutes, that friend will disappear from the nearby friend list? Or should we display the last known location?
Interviewer: We can assume inactive friends will no longer be shown.

Candidate: Do we need to worry about privacy and data laws such as GDPR or CCPA?
Interviewer: Good question. For simplicity, don't worry about it for now.

Functional requirements

- Users should be able to see nearby friends on their mobile apps. Each entry in the nearby friend list has a distance and a timestamp indicating when the distance was last updated.
- Nearby friend lists should be updated every few seconds.

Non-functional requirements

- Low latency. It's important to receive location updates from friends without too much delay.
- Reliability. The system needs to be reliable overall, but occasional data point loss is acceptable.
- Eventual consistency. The location data store doesn't need strong consistency. A few seconds delay in receiving location data in different replicas is acceptable.

Back-of-the-envelope estimation

Let's do a back-of-the-envelope estimation to determine the potential scale and challenges our solution will need to address. Some constraints and assumptions are listed below:

- Nearby friends are defined as friends whose locations are within a 5-mile radius.
- The location refresh interval is 30 seconds. The reason for this is that human walking speed is slow (average $3 \sim 4$ miles per hour). The distance traveled in 30 seconds does not make a significant difference on the "nearby friends" feature.
- On average, 100 million users use the "nearby friends" feature every day.
- Assume the number of concurrent users is 10% of DAU (Daily Active Users), so the number of concurrent users is 10 million.
- On average, a user has 400 friends. Assume all of them use the "nearby friends" feature.
- The app displays 20 nearby friends per page and may load more nearby friends upon request.

Calculate QPS
• 100 million DAU • Concurrent users: $10\% \times 100 \text{ million} = 10 \text{ million}$ • Users report their locations every 30 seconds. • Location update QPS = $\frac{10 \text{ million}}{30} = \sim 334{,}000$

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

In this section, we will discuss the following:

- High-level design
- API design
- Data model

In other chapters, we usually discuss API design and data model before the high-level design. However, for this problem, the communication protocol between client and server might not be a straightforward HTTP protocol, as we need to push location data to all friends. Without understanding the high-level design, it's difficult to know what the APIs look like. Therefore, we discuss the high-level design first.

High-level design

At a high level, this problem calls for a design with efficient message passing. Conceptually, a user would like to receive location updates from every active friend nearby. It could in theory be done purely peer-to-peer, that is, a user could maintain a persistent connection to every other active friend in the vicinity (Figure 2.2).

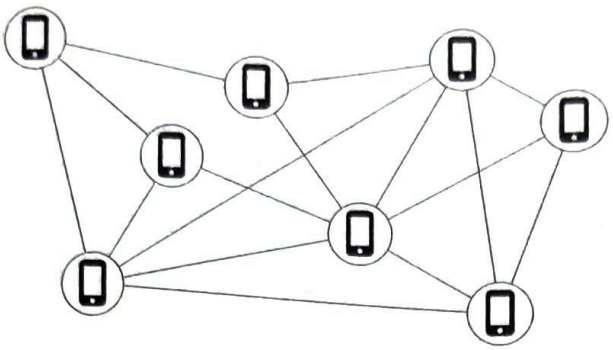


Figure 2.2: Peer-to-peer

This solution is not practical for a mobile device with sometimes flaky connections and a tight power consumption budget, but the idea sheds some light on the general design direction.

A more practical design would have a shared backend and look like this:

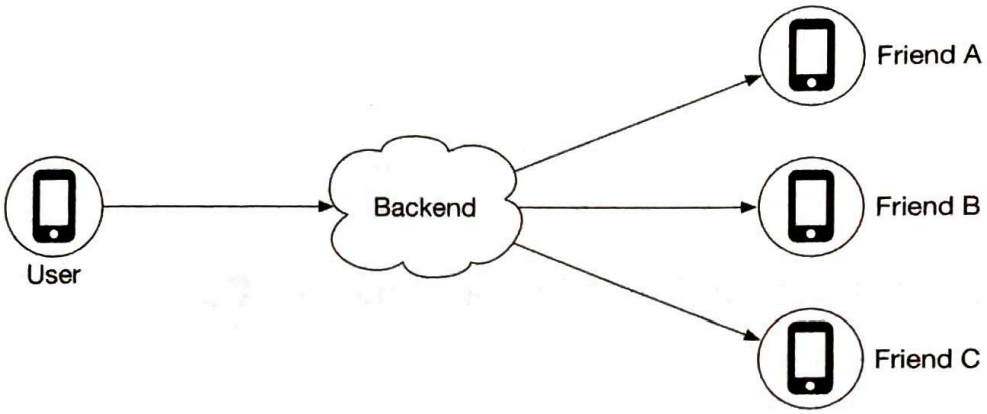


Figure 2.3: Shared backend

What are the responsibilities of the backend in Figure 2.3?

- Receive location updates from all active users.
- For each location update, find all the active friends who should receive it and forward it to those users' devices.
- If the distance between two users is over a certain threshold, do not forward it to the recipient's device.

This sounds pretty simple. What is the issue? Well, to do this at scale is not easy. We have 10 million active users. With each user updating the location information every 30 seconds, there are 334K updates per second. If on average each user has 400 friends, and we further assume that roughly 10% of those friends are online and nearby, every second the backend forwards 334K × 400 × 10% = 14 million location updates per second. That is a lot of updates to forward.

Proposed design

We will first come up with a high-level design for the backend at a lower scale. Later in the deep dive section, we will optimize the design for scale.

Figure 2.4 shows the basic design that should satisfy the functional requirements. Let's go over each component in the design.

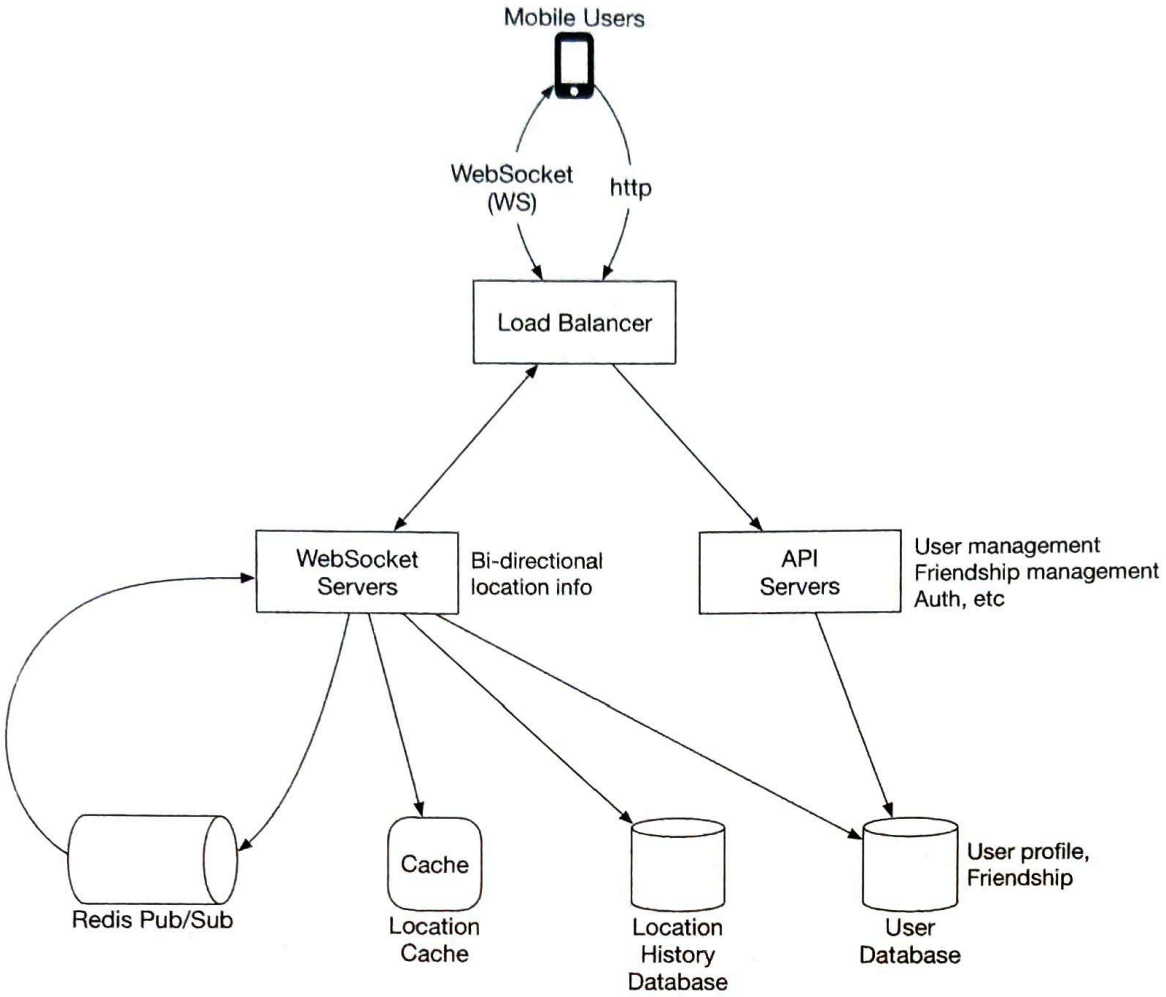


Figure 2.4: High-level design

Load balancer

The load balancer sits in front of the RESTful API servers and the stateful, bi-directional WebSocket servers. It distributes traffic across those servers to spread out load evenly.

RESTful API servers

This is a cluster of stateless HTTP servers that handles the typical request/response traffic. The API request flow is highlighted in Figure 2.5. This API layer handles auxiliary tasks like adding/removing friends, updating user profiles, etc. These are very common and we will not go into more detail.

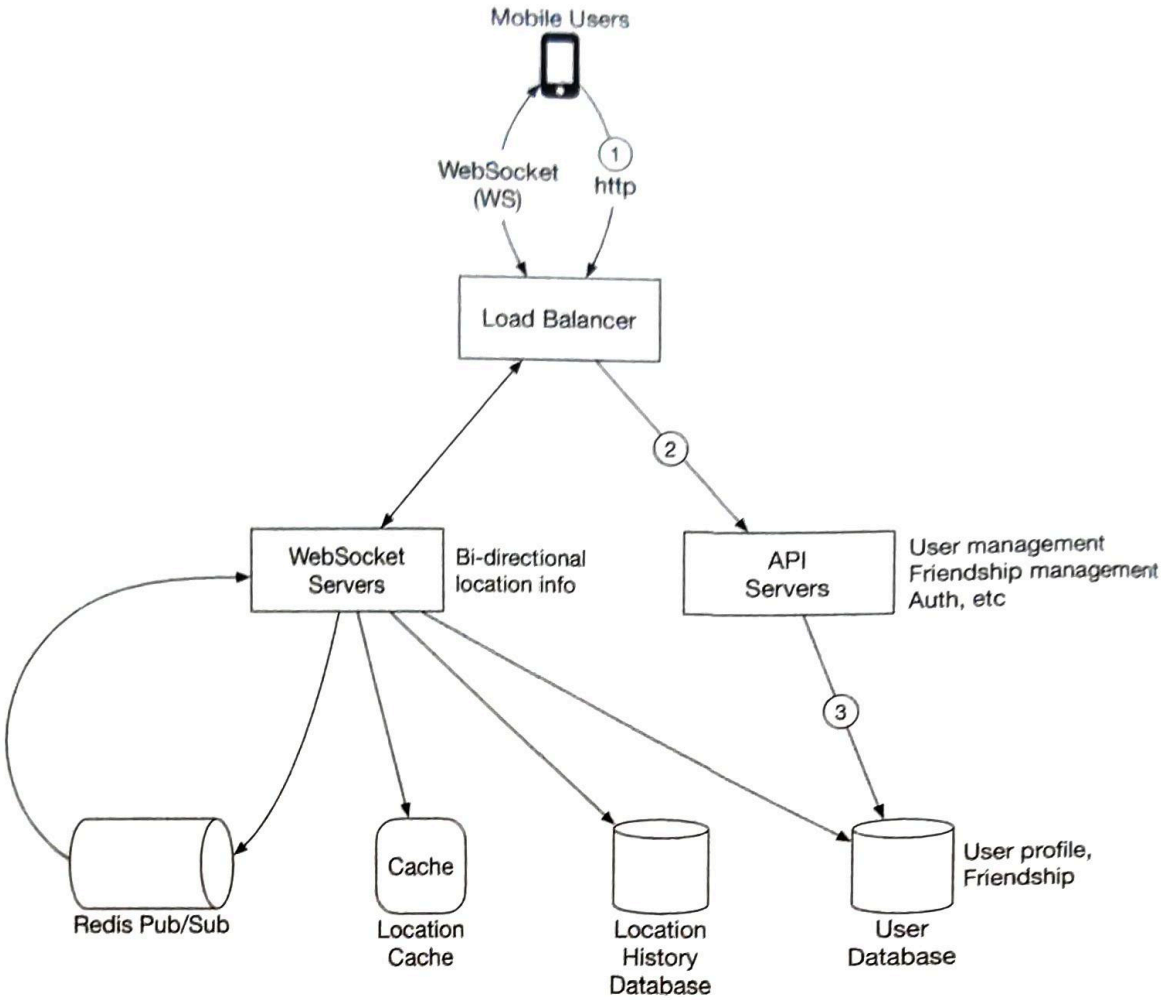


Figure 2.5: RESTful API request flow

WebSocket servers

This is a cluster of stateful servers that handles the near real-time update of friends' locations. Each client maintains one persistent WebSocket connection to one of these servers. When there is a location update from a friend who is within the search radius, the update is sent on this connection to the client.

Another major responsibility of the WebSocket servers is to handle client initialization for the "nearby friends" feature. It seeds the mobile client with the locations of all nearby online friends. We will discuss how this is done in more detail later.

Note "WebSocket connection" and "WebSocket connection handler" are interchangeable in this chapter.

Redis location cache

Redis is used to store the most recent location data for each active user. There is a Time to Live (TTL) set on each entry in the cache. When the TTL expires, the user is no longer active and the location data is expunged from the cache. Every update refreshes the TTL. Other KV stores that support TTL could also be used.

User database

The user database stores user data and user friendship data. Either a relational database or a NoSQL database can be used for this.

Location history database

This database stores users' historical location data. It is not directly related to the "nearby friends" feature.

Redis Pub/Sub server

Redis Pub/Sub [2] is a very lightweight message bus. Channels in Redis Pub/Sub are very cheap to create. A modern Redis server with GBs of memory could hold millions of channels (also called topics). Figure 2.6 shows how Redis Pub/Sub works.

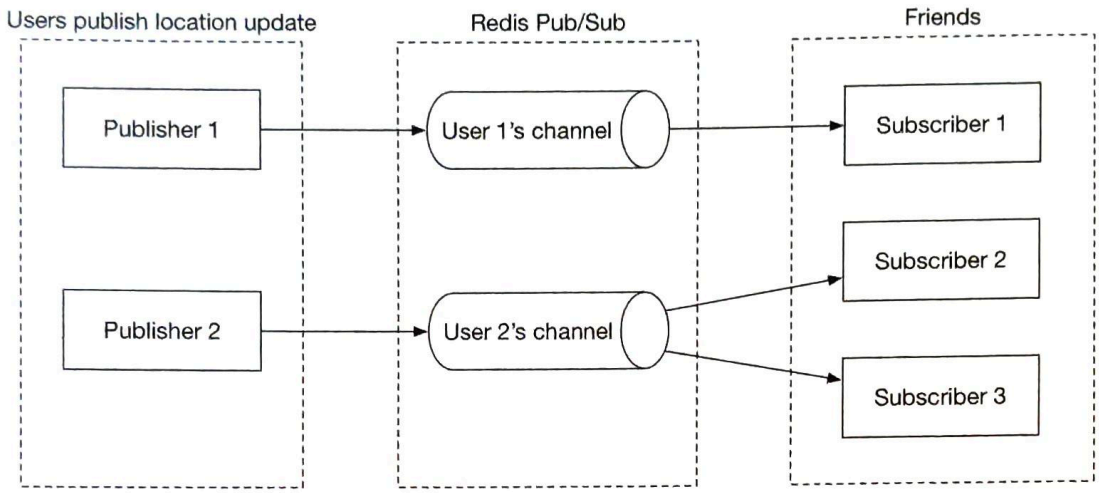


Figure 2.6: Redis Pub/Sub

In this design, location updates received via the WebSocket server are published to the user's own channel in the Redis Pub/Sub server. A dedicated WebSocket connection handler for each active friend subscribes to the channel. When there is a location update, the WebSocket handler function gets invoked, and for each active friend, the function recomputes the distance. If the new distance is within the search radius, the new location and timestamp are sent via the WebSocket connection to the friend's client. Other message buses with lightweight channels could also be used.

Now that we understand what each component does, let's examine what happens when a user's location changes from the system's perspective.

Periodic location update

The mobile client sends periodic location updates over the persistent WebSocket connection. The flow is shown in Figure 2.7.

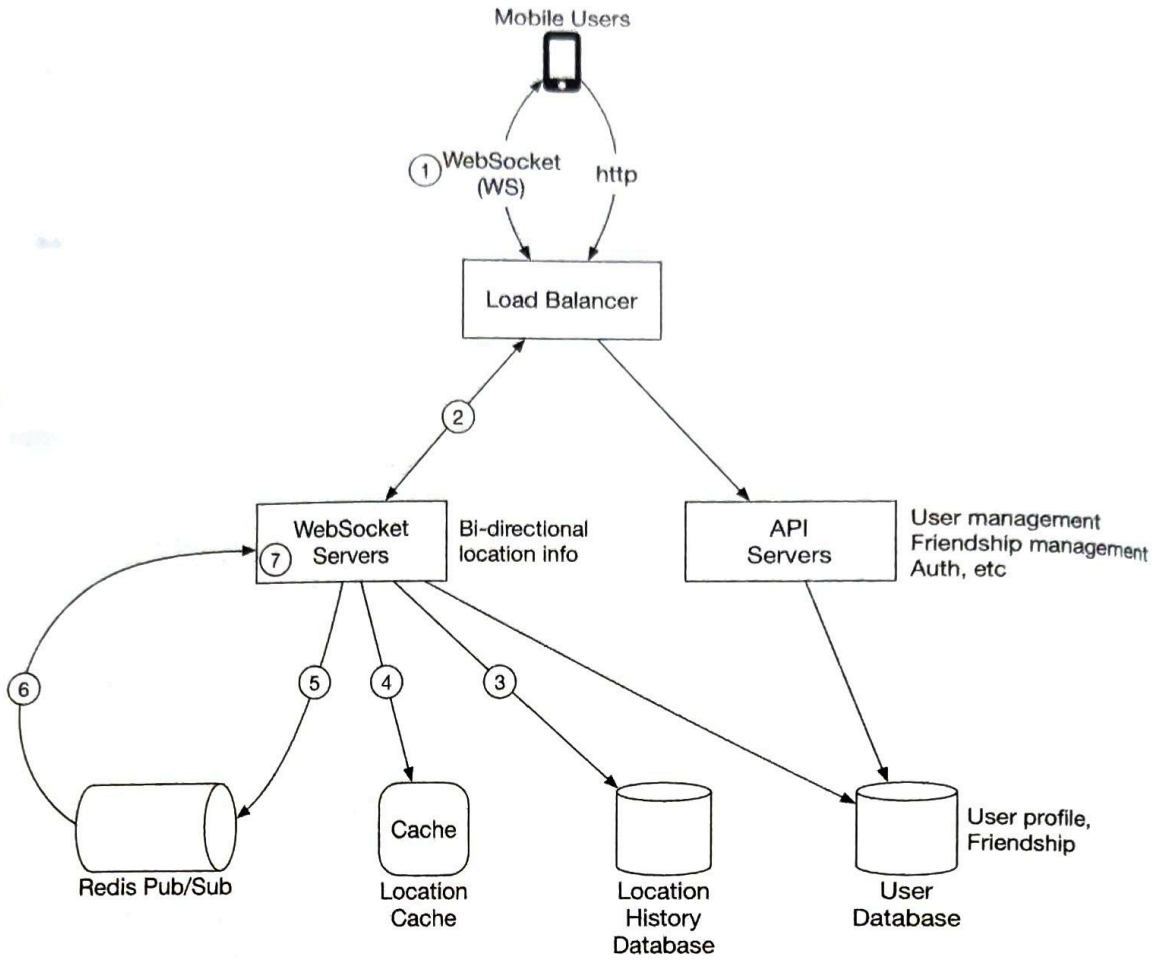


Figure 2.7: Periodic location update

1. The mobile client sends a location update to the load balancer.
2. The load balancer forwards the location update to the persistent connection on the WebSocket server for that client.
3. The WebSocket server saves the location data to the location history database.
4. The WebSocket server updates the new location in the location cache. The update refreshes the TTL. The WebSocket server also saves the new location in a variable in the user's WebSocket connection handler for subsequent distance calculations.
5. The WebSocket server publishes the new location to the user's channel in the Redis Pub/Sub server. Steps 3 to 5 can be executed in parallel.
6. When Redis Pub/Sub receives a location update on a channel, it broadcasts the update to all the subscribers (WebSocket connection handlers). In this case, the subscribers are all the online friends of the user sending the update. For each subscriber (i.e., for each of the user's friends), its WebSocket connection handler would receive the user location update.
7. On receiving the message, the WebSocket server, on which the connection handler lives, computes the distance between the user sending the new location (the location

data is in the message) and the subscriber (the location data is stored in a variable with the WebSocket connection handler for the subscriber).

8. This step is not drawn on the diagram. If the distance does not exceed the search radius, the new location and the last updated timestamp are sent to the subscriber's client. Otherwise, the update is dropped.

Since understanding this flow is extremely important, let's examine it again with a concrete example, as shown in Figure 2.8. Before we start, let's make a few assumptions.

- User 1's friends: User 2, User 3, and User 4.
- User 5's friends: User 4 and User 6.

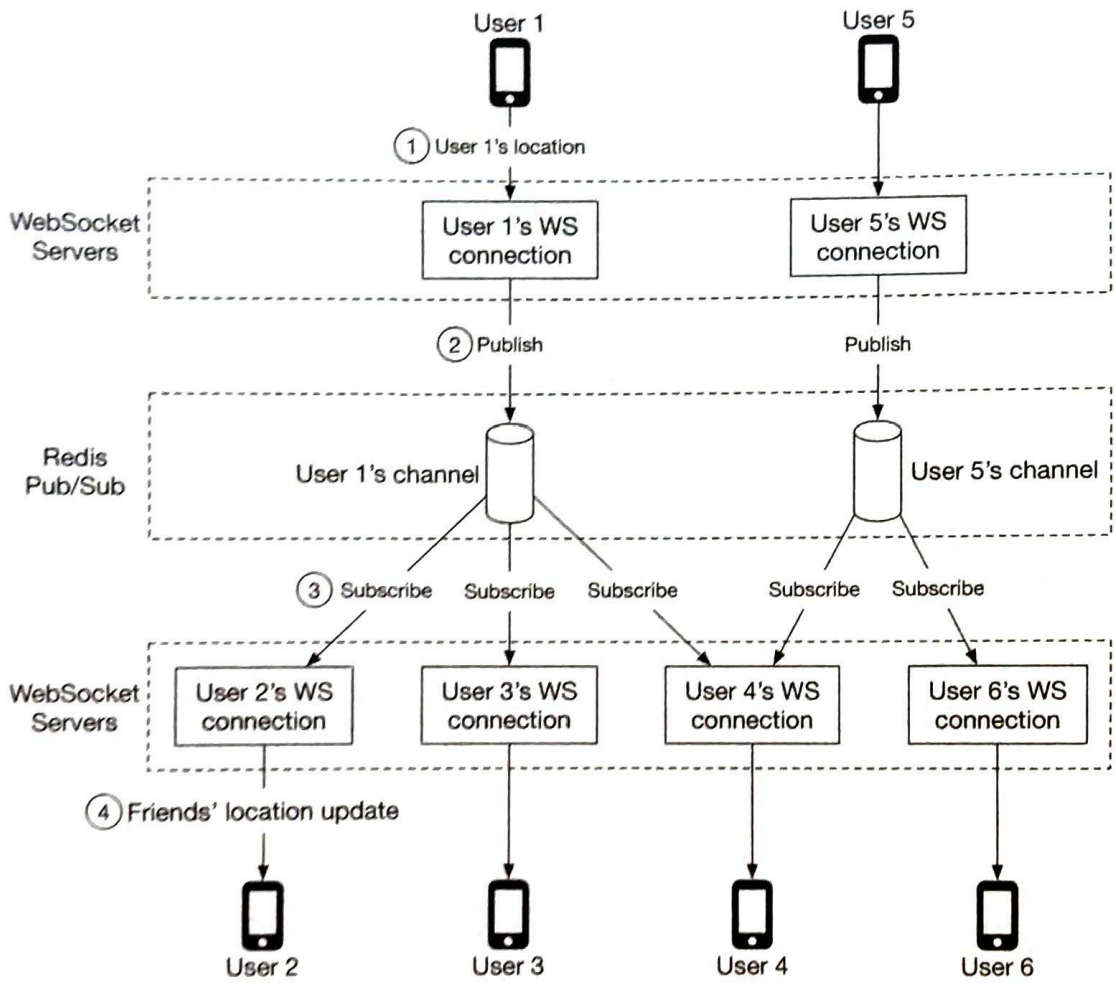


Figure 2.8: Send location update to friends

1. When User 1's location changes, their location update is sent to the WebSocket server which holds User 1's connection.
2. The location is published to User 1's channel in Redis Pub/Sub server.
3. Redis Pub/Sub server broadcasts the location update to all subscribers. In this case,

subscribers are WebSocket connection handlers (User 1's friends).

4. If the distance between the user sending the location (User 1) and the subscriber (User 2) doesn't exceed the search radius, the new location is sent to the client (User 2).

This computation is repeated for every subscriber to the channel. Since there are 400 friends on average, and we assume that 10% of those friends are online and nearby, there are about 40 location updates to forward for each user's location update.

API design

Now that we have created a high-level design, let's list APIs needed.

WebSocket: Users send and receive location updates through the WebSocket protocol. At the minimum, we need the following APIs.

1. Periodic location update
Request: Client sends latitude, longitude, and timestamp.
Response: Nothing.

2. Client receives location updates
Data sent: Friend location data and timestamp.

3. WebSocket initialization
Request: Client sends latitude, longitude, and timestamp.
Response: Client receives friends' location data.

4. Subscribe to a new friend
Request: WebSocket server sends friend ID.
Response: Friend's latest latitude, longitude, and timestamp.

5. Unsubscribe a friend
Request: WebSocket server sends friend ID.
Response. Nothing.

HTTP requests: the API servers handle tasks like adding/removing friends, updating user profiles, etc. These are very common and we will not go into detail here.

Data model

Another important element to discuss is the data model. We already talked about the User DB in the high-level design, so let's focus on the location cache and location history database.

Location cache

The location cache stores the latest locations of all active users who have had the nearby friends feature turned on. We use Redis for this cache. The key/value of the cache is shown in Table 2.1.

key	value
user_id	{latitude, longitude, timestamp}

Table 2.1: Location cache

Why don't we use a database to store location data?

The "nearby friends" feature only cares about the **current** location of a user. Therefore, we only need to store one location per user. Redis is an excellent choice because it provides super-fast read and write operations. It supports TTL, which we use to auto-purge users from the cache who are no longer active. The current locations do not need to be durably stored. If the Redis instance goes down, we could replace it with an empty new instance and let the cache be filled as new location updates stream in. The active users could miss location updates from friends for an update cycle or two while the new cache warms. It is an acceptable tradeoff. In the deep dive section, we will discuss ways to lessen the impact on users when the cache gets replaced.

Location history database

The location history database stores users' historical location data and the schema looks like this:

user_id	latitude	longitude	timestamp

We need a database that handles the heavy-write workload well and can be horizontally scaled. Cassandra is a good candidate. We could also use a relational database. However, with a relational database, the historical data would not fit in a single instance so we need to shard that data. The most basic approach is to shard by user ID. This sharding scheme ensures that load is evenly distributed among all the shards, and operationally, it is easy to maintain.

Step 3 - Design Deep Dive

The high-level design we created in the previous section works in most cases, but it will likely break at our scale. In this section, we work together to uncover the bottlenecks as we increase the scale, and along the way work on solutions to eliminate those bottlenecks.

How well does each component scale?

API servers

The methods to scale the RESTful API tiers are well understood. These are stateless servers, and there are many ways to auto-scale the clusters based on CPU usage, load, or I/O. We will not go into detail here.

WebSocket servers

For the WebSocket cluster, it is not difficult to auto-scale based on usage. However, the WebSocket servers are stateful, so care must be taken when removing existing nodes. Before a node can be removed, all existing connections should be allowed to drain. To

achieve that, we can mark a node as "draining" at the load balancer so that no new WebSocket connections will be routed to the draining server. Once all the existing connections are closed (or after a reasonably long wait), the server is then removed.

Releasing a new version of the application software on a WebSocket server requires the same level of care.

It is worth noting that effective auto-scaling of stateful servers is the job of a good load balancer. Most cloud load balancers handle this job very well.

Client initialization

The mobile client on startup establishes a persistent WebSocket connection with one of the WebSocket server instances. Each connection is long-running. Most modern languages are capable of maintaining many long-running connections with a reasonably small memory footprint.

When a WebSocket connection is initialized, the client sends the initial location of the user, and the server performs the following tasks in the WebSocket connection handler.

1. It updates the user's location in the location cache.
2. It saves the location in a variable of the connection handler for subsequent calculations.
3. It loads all the user's friends from the user database.
4. It makes a batched request to the location cache to fetch the locations for all the friends. Note that because we set a TTL on each entry in the location cache to match our inactivity timeout period, if a friend is inactive then their location will not be in the location cache.
5. For each location returned by the cache, the server computes the distance between the user and the friend at that location. If the distance is within the search radius, the friend's profile, location, and last updated timestamp are returned over the WebSocket connection to the client.
6. For each friend, the server subscribes to the friend's channel in the Redis Pub/Sub server. We will explain our use of Redis Pub/Sub shortly. Since creating a new channel is cheap, the user subscribes to all active and inactive friends. The inactive friends will take up a small amount of memory on the Redis Pub/Sub server, but they will not consume any CPU or I/O (since they do not publish updates) until they come online.
7. It sends the user's current location to the user's channel in the Redis Pub/Sub server.

User database

The user database holds two distinct sets of data: user profiles (user ID, username, profile URL, etc.) and friendships. These datasets at our design scale will likely not fit in a single relational database instance. The good news is that the data is horizontally scal-

able by sharding based on user ID. Relational database sharding is a very common technique.

As a side note, at the scale we are designing for, the user and friendship datasets will likely be managed by a dedicated team and be available via an internal API. In this scenario, the WebSocket servers will use the internal API instead of querying the database directly to fetch user and friendship-related data. Whether accessing via API or direct database queries, it does not make much difference in terms of functionality or performance.

Location cache

We choose Redis to cache the most recent locations of all the active users. As mentioned earlier, we also set a TTL on each key. The TTL is renewed upon every location update. This puts a cap on the maximum amount of memory used. With 10 million active users at peak, and with each location taking no more than 100 bytes, a single modern Redis server with many GBs of memory should be able to easily hold the location information for all users.

However, with 10 million active users roughly updating every 30 seconds, the Redis server will have to handle 334K updates per second. That is likely a little too high, even for a modern high-end server. Luckily, this cache data is easy to shard. The location data for each user is independent, and we can evenly spread the load among several Redis servers by sharding the location data based on user ID.

To improve availability, we could replicate the location data on each shard to a standby node. If the primary node goes down, the standby could be quickly promoted to minimize downtime.

Redis Pub/Sub server

The Pub/Sub server is used as a routing layer to direct messages (location updates) from one user to all the online friends. As mentioned earlier, we choose Redis Pub/Sub because it is very lightweight to create new channels. A new channel is created when someone subscribes to it. If a message is published to a channel that has no subscribers, the message is dropped, placing very little load on the server. When a channel is created, Redis uses a small amount of memory to maintain a hash table and a linked list [3] to track the subscribers. If there is no update on a channel when a user is offline, no CPU cycles are used after a channel is created. We take advantage of this in our design in the following ways:

1. We assign a unique channel to every user who uses the "nearby friends" feature. A user would, upon app initialization, subscribe to each friend's channel, whether the friend is online or not. This simplifies the design since the backend does not need to handle subscribing to a friend's channel when the friend becomes active, or handling unsubscribing when the friend becomes inactive.
2. The tradeoff is that the design would use more memory. As we will see later, memory use is unlikely to be the bottleneck. Trading higher memory use for a simpler architecture is worth it in this case.

How many Redis Pub/Sub servers do we need?

Let's do some math on memory and CPU usage.

Memory usage

Assuming a channel is allocated for each user who uses the nearby friends feature, we need 100 million channels (1 billion × 10%). Assuming that on average a user has 100 active friends using this feature (this includes friends who are nearby, or not), and it takes about 20 bytes of pointers in the internal hash table and linked list to track each subscriber, it will need about 200GB (100 million × 20 bytes × 100 friends / 10^9 = 200GB) to hold all the channels. For a modern server with 100GB of memory, we will need about 2 Redis Pub/Sub servers to hold all the channels.

CPU usage

As previously calculated, the Pub/Sub server pushes about 14 million updates per second to subscribers. Even though it is not easy to estimate with any accuracy how many messages a modern Redis server could push a second without actual benchmarking, it is safe to assume that a single Redis server will not be able to handle that load. Let's pick a conservative number and assume that a modern server with a gigabit network could handle about 100,000 subscriber pushes per second. Given how small our location update messages are, this number is likely to be conservative. Using this conservative estimate, we will need to distribute the load among 14 million / 100,000 = 140 Redis servers. Again, this number is likely too conservative, and the actual number of servers could be much lower.

From the math, we conclude that:

- The bottleneck of Redis Pub/Sub server is the CPU usage, not the memory usage.
- To support our scale, we need a distributed Redis Pub/Sub cluster.

Distributed Redis Pub/Sub server cluster

How do we distribute the channels to hundreds of Redis servers? The good news is that the channels are independent of each other. This makes it relatively easy to spread the channels among multiple Pub/Sub servers by sharding, based on the publisher's user ID. Practically speaking though, with hundreds of Pub/Sub servers, we should go into a bit more detail on how this is done so that operationally it is somewhat manageable, as servers inevitably go down from time to time.

Here, we introduce a service discovery component to our design. There are many service discovery packages available, with etcd [4] and ZooKeeper [5] among the most popular ones. Our need for the service discovery component is very basic. We need these two features:

1. The ability to keep a list of servers in the service discovery component, and a simple UI or API to update it. Fundamentally, service discovery is a small key-value store for holding configuration data. Using Figure 2.9 as an example, the key and value for

the hash ring could look like this:

Key: `/config/pub_sub_ring`
Value: `["p_1", "p_2", "p_3", "p_4"]`

2. The ability for clients (in this case, the WebSocket servers) to subscribe to any updates to the "Value" (Redis Pub/Sub servers).

Under the "Key" mentioned in point 1, we store a hash ring of all the active Redis Pub/Sub servers in the service discovery component (See the consistent hashing chapter in Volume 1 of the System Design Interview book or [6] on details of a hash ring). The hash ring is used by the publishers and subscribers of the Redis Pub/Sub servers to determine the Pub/Sub server to talk to for each channel. For example, channel 2 lives in Redis Pub/Sub server 1 in Figure 2.9.

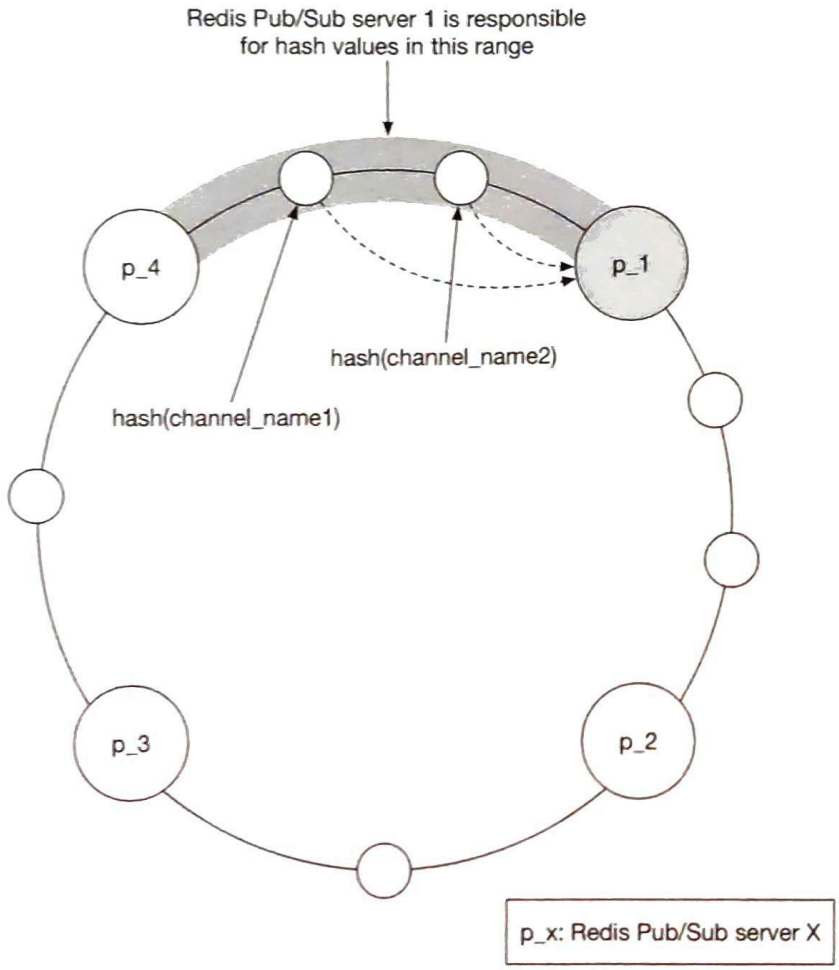


Figure 2.9: Consistent hashing

Figure 2.10 shows what happens when a WebSocket server publishes a location update to a user's channel.

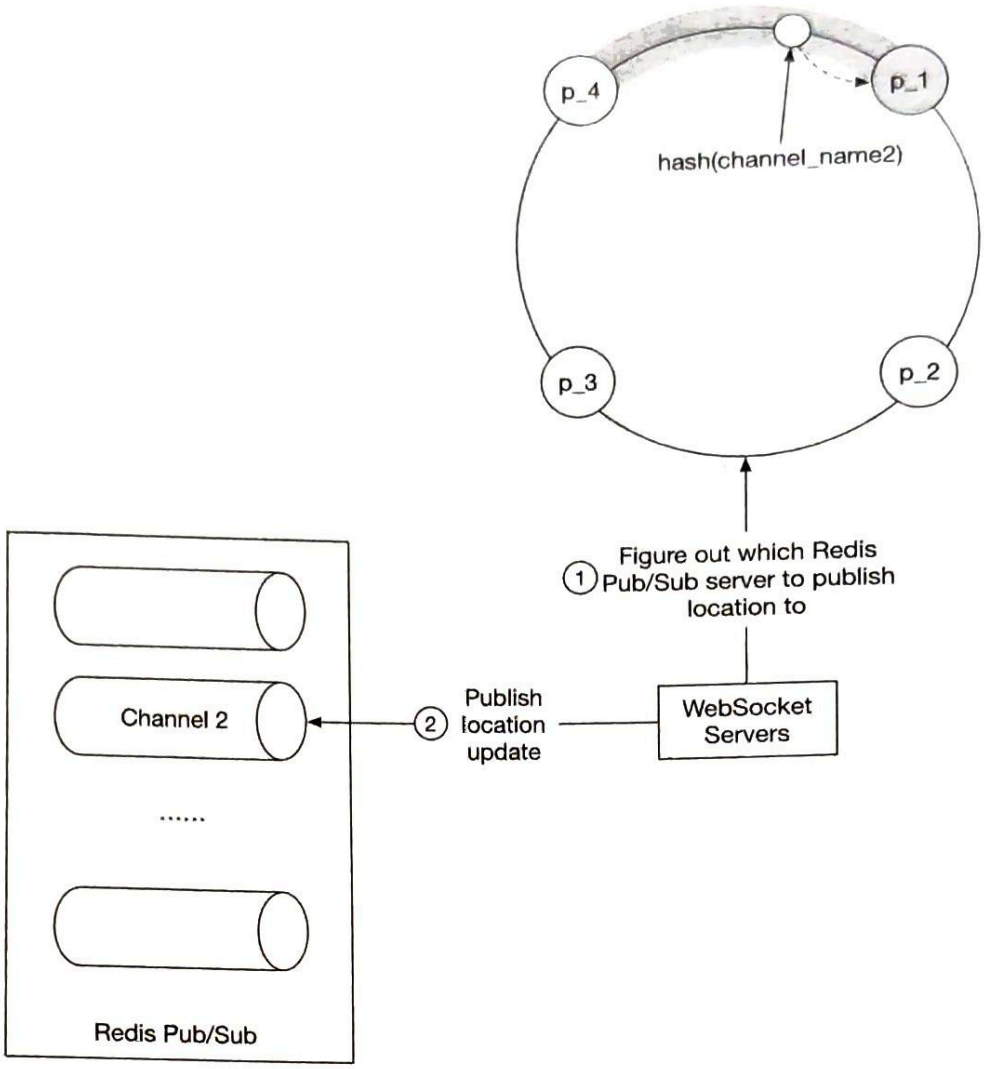


Figure 2.10: Figure out the correct Redis Pub/Sub server

1. The WebSocket server consults the hash ring to determine the Redis Pub/Sub server to write to. The source of truth is stored in service discovery, but for efficiency, a copy of the hash ring could be cached on each WebSocket server. The WebSocket server subscribes to any updates on the hash ring to keep its local in-memory copy up to date.
2. WebSocket server publishes the location update to the user's channel on that Redis Pub/Sub server.

Subscribing to a channel for location updates uses the same mechanism.

Scaling considerations for Redis Pub/Sub servers

How should we scale the Redis Pub/Sub server cluster? Should we scale it up and down daily, based on traffic patterns? This is a very common practice for stateless servers because it is low risk and saves costs. To answer these questions, let's examine some of the properties of the Redis Pub/Sub server cluster.

1. The messages sent on a Pub/Sub channel are not persisted in memory or on disk. They are sent to all subscribers of the channel and removed immediately after. If there are no subscribers, the messages are just dropped. In this sense, the data going through the Pub/Sub channel is stateless.
2. However, there are indeed states stored in the Pub/Sub servers for the channels. Specifically, the subscriber list for each channel is a key piece of the states tracked by the Pub/Sub servers. If a channel is moved, which could happen when the channel's Pub/Sub server is replaced, or if a new server is added or an old server removed on the hash ring, then every subscriber to the moved channel must know about it, so they could unsubscribe from the channel on the old server and resubscribe to the replacement channel on the new server. In this sense, a Pub/Sub server is stateful, and coordination with all subscribers to the server must be orchestrated to minimize service interruptions.

For these reasons, we should treat the Redis Pub/Sub cluster more like a stateful cluster, similar to how we would handle a storage cluster. With stateful clusters, scaling up or down has some operational overhead and risks, so it should be done with careful planning. The cluster is normally over-provisioned to make sure it can handle daily peak traffic with some comfortable headroom to avoid unnecessary resizing of the cluster.

When we inevitably have to scale, be mindful of these potential issues:

- When we resize a cluster, many channels will be moved to different servers on the hash ring. When the service discovery component notifies all the WebSocket servers of the hash ring update, there will be a ton of resubscription requests.
- During these mass resubscription events, some location updates might be missed by the clients. Although occasional misses are acceptable for our design, we should minimize the occurrences.
- Because of the potential interruptions, resizing should be done when usage is at its lowest in the day.

How is resizing actually done? It is quite simple. Follow these steps:

- Determine the new ring size, and if scaling up, provision enough new servers.
- Update the keys of the hash ring with the new content.
- Monitor your dashboard. There should be some spike in CPU usage in the WebSocket cluster.

Using the hash ring from Figure 2.9 above, if we were to add 2 new nodes, say, p_5, and p_6, the hash ring would be updated like this:

```
Old: ["p_1", "p_2", "p_3", "p_4"]
New: ["p_1", "p_2", "p_3", "p_4", "p_5", "p_6"]
```

Operational considerations for Redis Pub/Sub servers

The operational risk of replacing an existing Redis Pub/Sub server is much, much lower. It does not cause a large number of channels to be moved. Only the channels on the server being replaced will need to be handled. This is good because servers inevitably go down and need to be replaced regularly.

When a Pub/Sub server goes down, the monitoring software should alert the on-call operator. Precisely how the monitoring software monitors the health of a Pub/Sub server is beyond the scope of this chapter, so it is not covered. The on-call operator updates the hash ring key in service discovery to replace the dead node with a fresh standby node. The WebSocket servers are notified about the update and each one then notifies its connection handlers to re-subscribe to the channels on the new Pub/Sub server. Each WebSocket handler keeps a list of all channels it has subscribed to, and upon receiving the notification from the server, it checks each channel against the hash ring to determine if a channel needs to be re-subscribed on a new server.

Using the hash ring from Figure 2.9 above, if `p_1` went down, and we replace it with `p1_new`, the hash ring would be updated like so:

Old: `["p_1", "p_2", "p_3", "p_4"]`

New: `["p_1_new", "p_2", "p_3", "p_4"]`

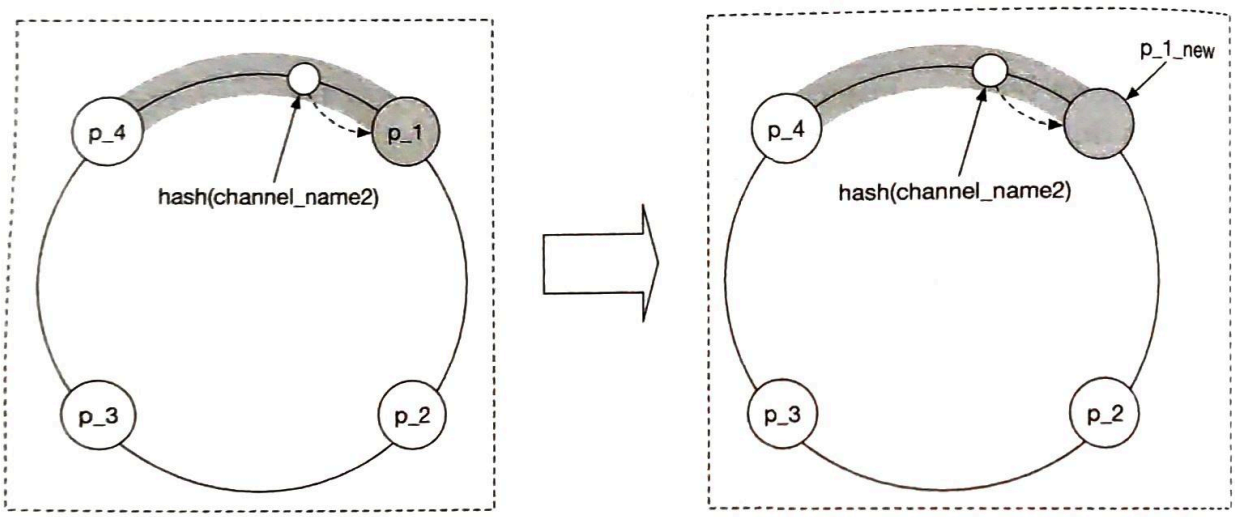


Figure 2.11: Replace Pub/Sub serverver

Adding/removing friends

What should the client do when the user adds or removes a friend? When a new friend is added, the client's WebSocket connection handler on the server needs to be notified, so it can subscribe to the new friend's Pub/Sub channel.

Since the "nearby friends" feature is within the ecosystem of a larger app, we can assume that the "nearby friends" feature could register a callback on the mobile client whenever a new friend is added. The callback, upon invocation, sends a message to the WebSocket server to subscribe to the new friend's Pub/Sub channel. The WebSocket server also

returns a message containing the new friend's latest location and timestamp, if they are active.

Likewise, the client could register a callback in the application whenever a friend is removed. The callback would send a message to the WebSocket server to unsubscribe from the friend's Pub/Sub channel.

This subscribe/unsubscribe callback could also be used whenever a friend has opted in or out of the location update.

Users with many friends

It is worth discussing whether a user with many friends could cause performance hotspots in our design. We assume here that there is a hard cap on the number of friends. (Facebook has a cap of 5,000 friends, for example). Friendships are bi-directional. We are not talking about a follower model in which a celebrity could have millions of followers.

In a scenario with thousands of friends, the Pub/Sub subscribers will be scattered among the many WebSocket servers in the cluster. The update load would be spread among them and it's unlikely to cause any hotspots.

The user would place a bit more load on the Pub/Sub server where their channel lives. Since there are over 100 Pub/Sub servers, these "whale" users would be spread out among the Pub/Sub servers and the incremental load should not overwhelm any single one.

Nearby random person

You might call this section an extra credit, as it's not in the initial functional requirements. What if the interviewer wants to update the design to show random people who opted-in to location-sharing?

One way to do this while leveraging our design is to add a pool of Pub/Sub channels by geohash. (See Chapter 1 Proximity Service for details on geohash). As shown in Figure 2.12, an area is divided into four geohash grids and a channel is created for each grid.

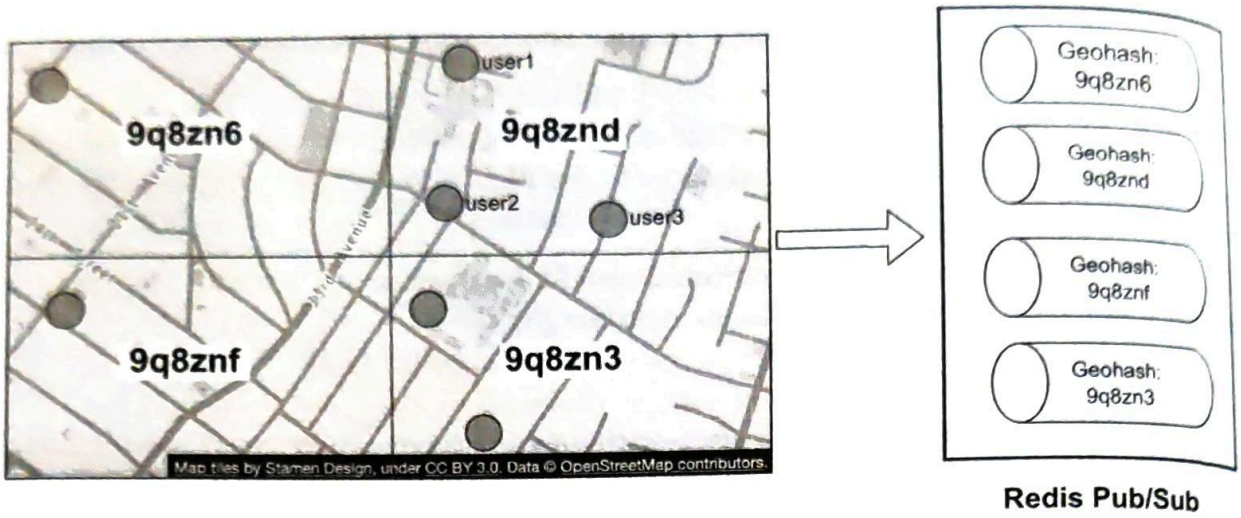


Figure 2.12: Redis Pub/Sub channels

Anyone within the grid subscribes to the same channel. Let's take grid 9q8znd for example as shown in Figure 2.13.

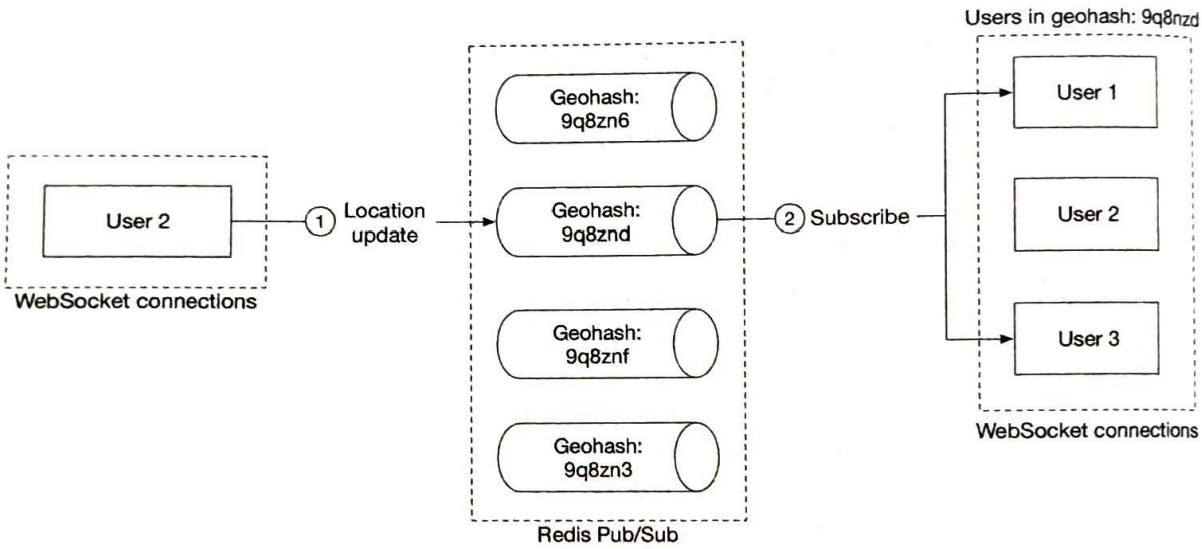


Figure 2.13: Publish location update to random nearby person

1. Here, when user 2 updates their location, the WebSocket connection handler computes the user's geohash ID and sends the location to the channel for that geohash.
2. Anyone nearby who subscribes to the channel (exclude the sender) will receive a location update message.

To handle people who are close to the border of a geohash grid, every client could subscribe to the geohash the user is in and the eight surrounding geohash grids. An example with all 9 geohash grids highlighted is shown in Figure 2.14.

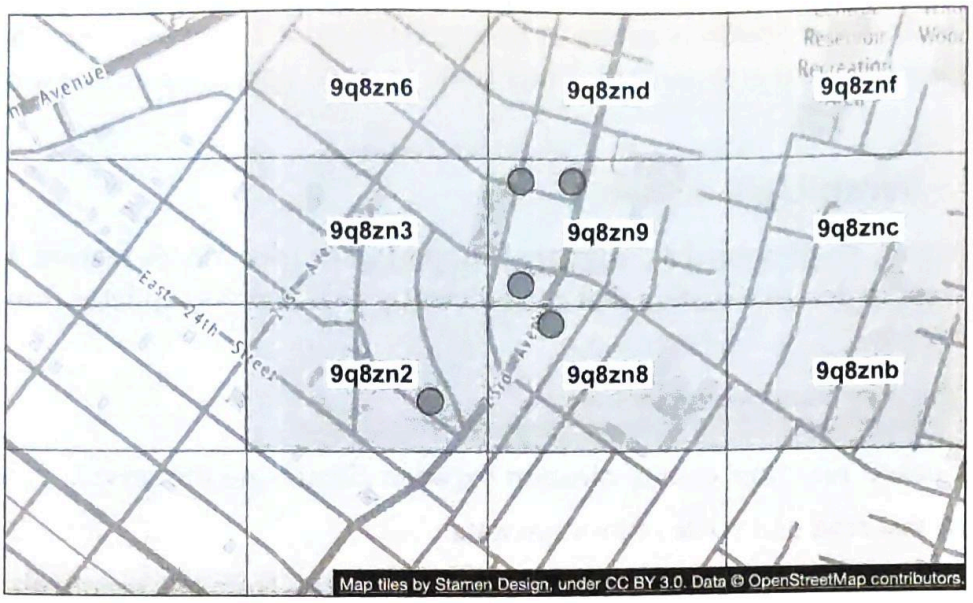


Figure 2.14: Nine geohash grids

Alternative to Redis Pub/Sub

Is there any good alternative to using Redis Pub/Sub as the routing layer? The answer is a resounding yes. Erlang [7] is a great solution for this particular problem. We would argue that Erlang is a better solution than the Redis Pub/Sub proposed above. However, Erlang is quite a niche, and hiring good Erlang programmers is hard. But if your team has Erlang expertise, this is a great option.

So, why Erlang? Erlang is a general programming language and runtime environment built for highly distributed and concurrent applications. When we say Erlang here, we specifically talk about the Erlang ecosystem itself. This includes the language component (Erlang or Elixir [8]) and the runtime environment and libraries (the Erlang virtual machine called BEAM [9] and the Erlang runtime libraries called OTP [10]).

The power of Erlang lies in its lightweight processes. An Erlang process is an entity running on the BEAM VM. It is several orders of magnitude cheaper to create than a Linux process. A minimal Erlang process takes about 300 bytes, and we can have millions of these processes on a single modern server. If there is no work to do in an Erlang process, it just sits there without using any CPU cycles at all. In other words, it is extremely cheap to model each of the 10 million active users in our design as an individual Erlang process.

Erlang is also very easy to distribute among many Erlang servers. The operational overhead is very low, and there are great tools to support debugging live production issues, safely. The deployment tools are also very strong.

How would we use Erlang in our design? We would implement the WebSocket service in Erlang, and also replace the entire cluster of Redis Pub/Sub with a distributed Erlang application. In this application, each user is modeled as an Erlang process. The user process would receive updates from the WebSocket server when a user's location is updated by the client. The user process also subscribes to updates from the Erlang processes of the

user's friends. Subscription is native in Erlang/OTP and it's easy to build. This forms a mesh of connections that would efficiently route location updates from one user to many friends.

Step 4 - Wrap Up

In this chapter, we presented a design that supports a nearby friends feature. Conceptually, we want to design a system that can efficiently pass location updates from one user to their friends.

Some of the core components include:

- WebSocket: real-time communication between clients and the server.
- Redis: fast read and write of location data.
- Redis Pub/Sub: routing layer to direct location updates from one user to all the online friends.

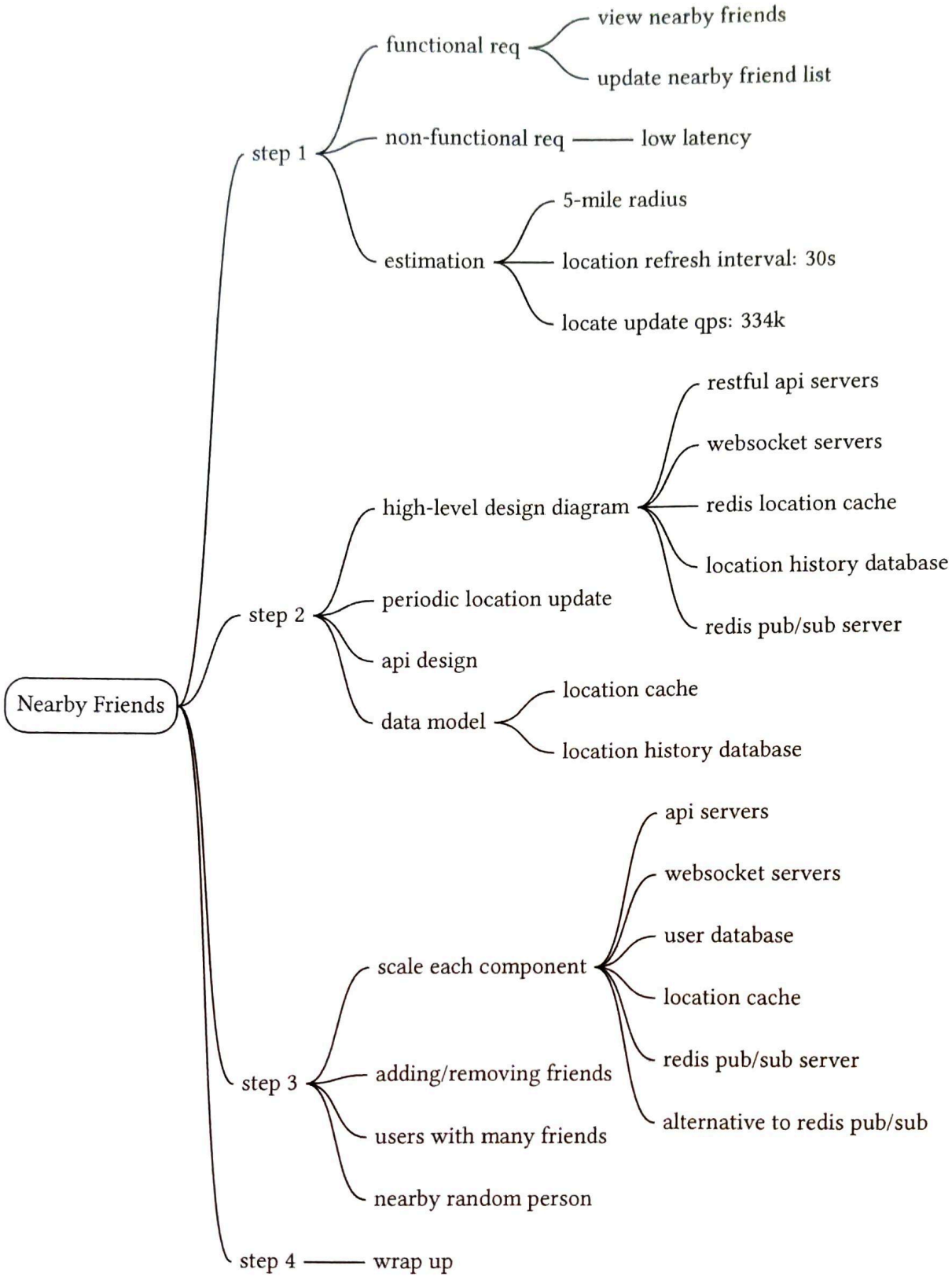
We first came up with a high-level design at a lower scale and then discussed challenges that arise as the scale increases. We explored how to scale the following:

- RESTful API servers
- WebSocket servers
- Data layer
- Redis Pub/Sub servers
- Alternative to Redis Pub/Sub

Finally, we discussed potential bottlenecks when a user has many friends and we proposed a design for the "nearby random person" feature.

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

Chapter Summary



Reference Material

[1] Facebook Launches "Nearby Friends". https://techcrunch.com/2014/04/17/facebook-nearby-friends/.

[2] Redis Pub/Sub. https://redis.io/topics/pubsub.

[3] Redis Pub/Sub under the hood. https://making.pusher.com/redis-pubsub-under-the-hood/.

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

[5] ZooKeeper. https://zookeeper.apache.org/.

[6] Consistent hashingones. https://www.toptal.com/big-data/consistent-hashing.

[7] Erlang. https://www.erlang.org/.

[8] Elixir. https://elixir-lang.org/.

[9] A brief introduction to BEAM. https://www.erlang.org/blog/a-brief-beam-primer/.

[10] OTP. https://www.erlang.org/doc/design_principles/des_princ.html.

3 Google Maps

In this chapter, we design a simple version of Google Maps. Before we proceed to the system design, let's learn a bit about Google Maps. Google started Project Google Maps in 2005 and developed a web mapping service. It provides many services such as satellite imagery, street maps, real-time traffic conditions, and route planning [1].

Google Maps helps users find directions and navigate to their destination. As of March 2021, Google Maps had one billion daily active users, 99% coverage of the world, and 25 million updates daily of accurate and real-time location information [2]. Given the enormous complexity of Google Maps, it is important to nail down which features our version of it supports.

Step 1 - Understand the Problem and Establish Design Scope

The interaction between the interviewer and the candidate could look like this:

Candidate: How many daily active users are we expecting?
Interviewer: 1 billion DAU.

Candidate: Which features should we focus on? Direction, navigation, and estimated time of arrival (ETA)?
Interviewer: Let's focus on location update, navigation, ETA, and map rendering.

Candidate: How large is the road data? Can we assume we have access to it?
Interviewer: Great questions. Yes, let's assume we obtained the road data from different sources. It is terabytes (TBs) of raw data.

Candidate: Should our system take traffic conditions into consideration?
Interviewer: Yes, traffic conditions are very important for accurate time estimation.

Candidate: How about different travel modes such as driving, walking, bus, etc?
Interviewer: We should be able to support different travel modes.

Candidate: Should it support multi-stop directions?
Interviewer: It is good to allow a user to define multiple stops, but let's not focus on it.

Candidate: How about business places and photos? How many photos are we expecting?
Interviewer: I am happy you asked and considered these. We do not need to design those.

In the rest of the chapter, we focus on three key features. The main devices that we need to support are mobile phones.

- User location update.
- Navigation service, including ETA service.
- Map rendering.

Non-functional requirements and constraints

- Accuracy: Users should not be given the wrong directions.
- Smooth navigation: On the client-side, users should experience very smooth map rendering.
- Data and battery usage: The client should use as little data and battery as possible. This is very important for mobile devices.
- General availability and scalability requirements.

Before jumping into the design, we will briefly introduce some basic concepts and terminologies that are helpful in designing Google Maps.

Map 101

Positioning system

The world is a sphere that rotates on its axis. At the very top, there is the north pole, and the very bottom is the south pole.

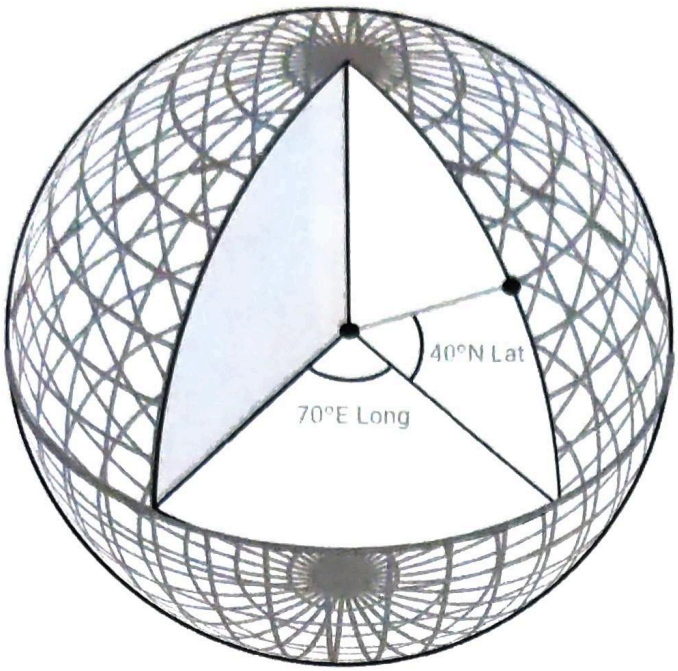


Figure 3.1: Latitude and longitude (source: [3])

Lat (Latitude): denotes how far north or south we are

Long (Longitude): denotes how far east or west we are

Going from 3D to 2D

The process of translating the points from a 3D globe to a 2D plane is called "Map Projection".

There are different ways to do map projection, and each comes with its own strengths and limitations. Almost all of them distort the actual geometry. Below we can see some examples.

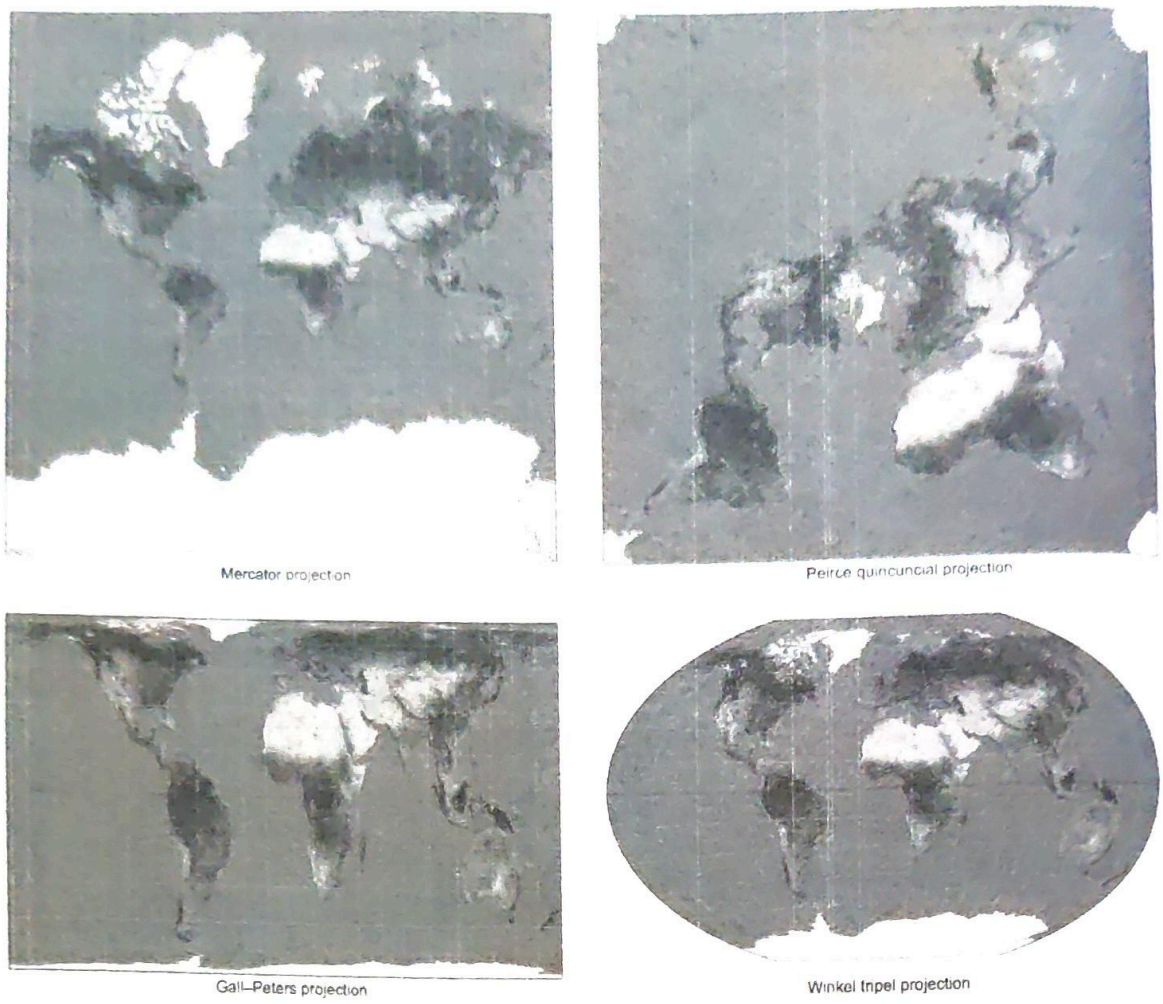


Figure 3.2: Map projections (source: Wikipedia [4] [5] [6] [7])

Google Maps selected a modified version of Mercator projection called Web Mercator. For more details on positioning systems and projections, please refer to [3].

Geocoding

Geocoding is the process of converting addresses to geographic coordinates. For instance, "1600 Amphitheatre Parkway, Mountain View, CA" is geocoded to a latitude/longitude pair of (latitude 37.423021, longitude −122.083739).

In the other direction, the conversion from the latitude/longitude pair to the actual human-readable address is called reverse geocoding.

One way to geocode is interpolation [8]. This method leverages the data from different sources such as geographic information systems (GIS) where the street network is mapped to the geographic coordinate space.

Geohashing

Geohashing is an encoding system that encodes a geographic area into a short string of letters and digits. At its core, it depicts the earth as a flattened surface and recursively divides the grids into sub-grids, which can be square or rectangular. We represent each grid with a string of numbers between 0 to 3 that are created recursively.

Let's assume the initial flattened surface is of size 20,000km × 10,000km. After the first division, we would have 4 grids of size 10,000km × 5,000km. We represent them as 00, 01, 10, and 11 as shown in Figure 3.3. We further divide each grid into 4 grids and use the same naming strategy. Each sub-grid is now of size 5,000km × 2,500km. We recursively divide the grids until each grid reaches a certain size threshold.

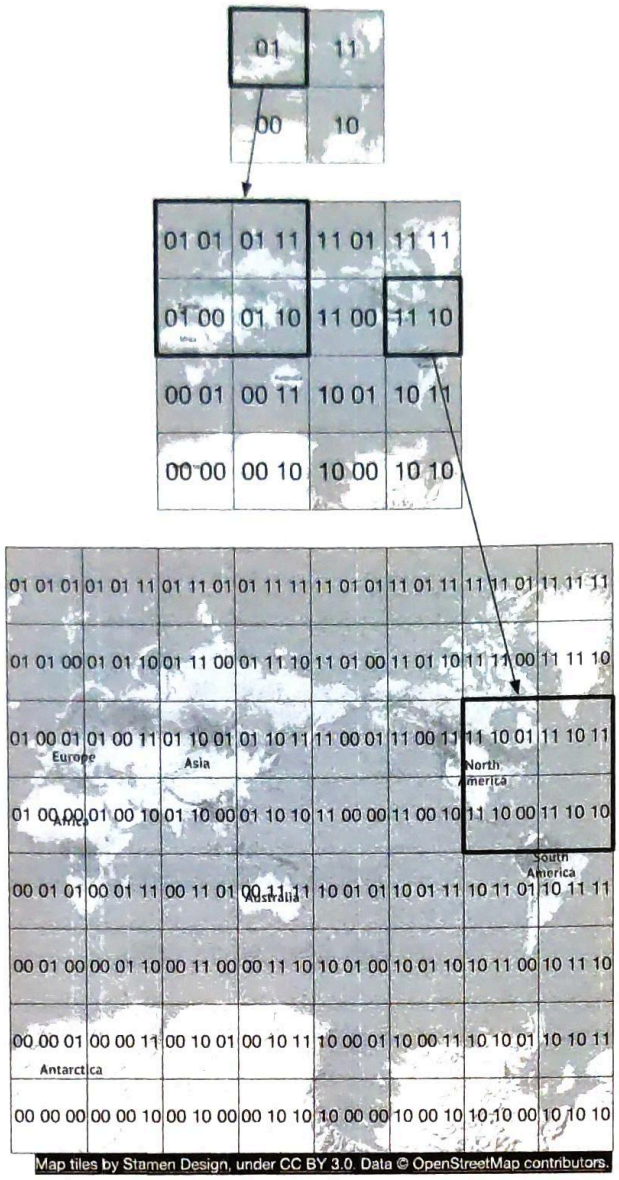


Figure 3.3: Geohashing

Geohashing has many uses. In our design, we use geohashing for map tiling. For more details on geohashing and its benefits, please refer to [9].

Map rendering

We won't go into a lot of detail about map rendering here, but it is worth mentioning the basics. One foundational concept in map rendering is tiling. Instead of rendering the entire map as one large custom image, the world is broken up into smaller tiles. The client only downloads the relevant tiles for the area the user is in and stitches them together like a mosaic for display.

There are distinct sets of tiles at different zoom levels. The client chooses the set of tiles appropriate for the zoom level of the map viewport on the client. This provides the right level of map details without consuming excess bandwidth. To illustrate with an extreme example, when the client is zoomed all the way out to show the entire world, we don't want to have to download hundreds of thousands of tiles for a very high zoom level. All the details would go to waste. Instead, the client would download one tile at the lowest zoom level, which represents the entire world with a single 256×256 pixel image.

Road data processing for navigation algorithms

Most routing algorithms are variations of Dijkstra's or A* pathfinding algorithms. The exact algorithm choice is a complex topic and we won't go into much detail in this chapter. What is important to note is that all these algorithms operate on a graph data structure, where intersections are nodes and roads are edges of the graph. See Figure 3.4 for an example:

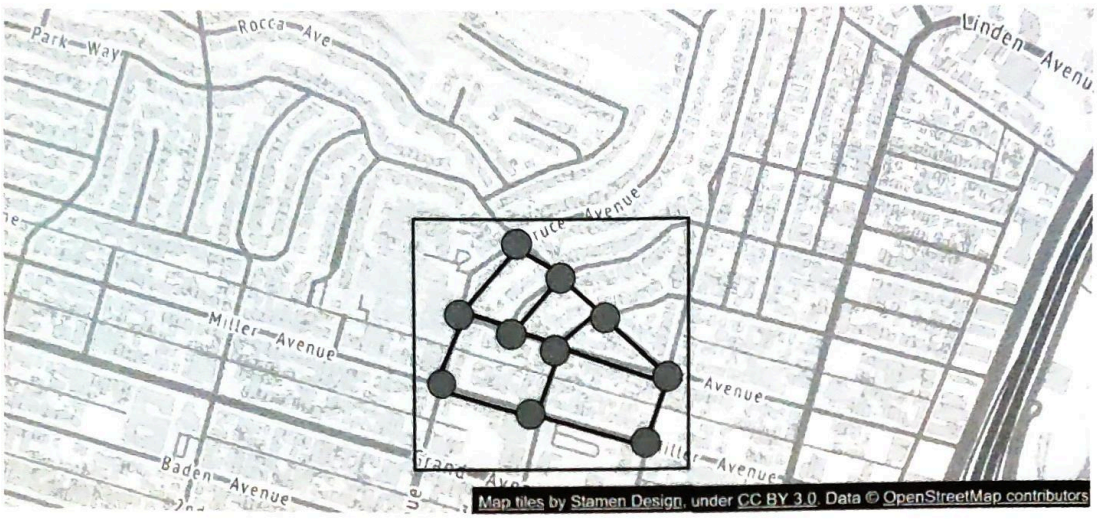


Figure 3.4: Map as a graph

The pathfinding performance for most of these algorithms is extremely sensitive to the size of the graph. Representing the entire world of road networks as a single graph would consume too much memory and is likely too large for any of these algorithms to run efficiently. The graph needs to be broken up into manageable units for these algorithms to work at our design scale.

One way to break up road networks around the world is very similar to the tiling concept we discussed for map rendering. By employing a similar subdivision technique as geohashing, we divide the world into small grids. For each grid, we convert the roads within the grid into a small graph data structure that consists of the nodes (intersections) and edges (roads) inside the geographical area covered by the grid. We call these grids routing tiles. Each routing tile holds references to all the other tiles it connects to. This is how the routing algorithms can stitch together a bigger road graph as it traverses these interconnected routing tiles.

By breaking up road networks into routing tiles that can be loaded on demand, the rout-

ing algorithms can significantly reduce memory consumption and improve pathfinding performance by only consuming a small subset of the routing tiles at a time, and only loading additional tiles as needed.

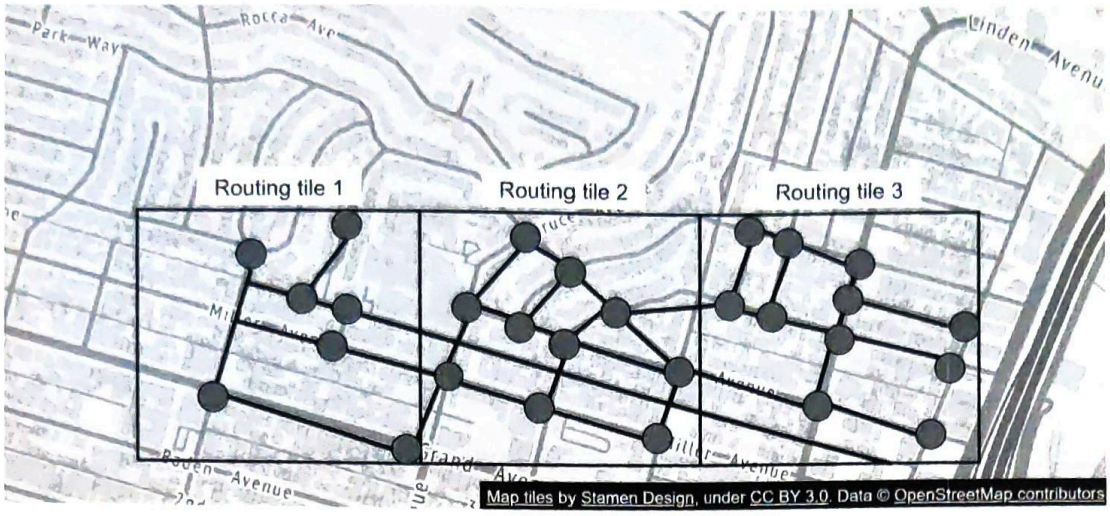


Figure 3.5: Routing tiles

Reminder
In Figure 3.5, we call these grids routing tiles. Routing tiles are similar to map tiles in that both are grids covering certain geographical areas. Map tiles are PNG images, while routing tiles are binary files of road data for the area covered by the tiles.

Hierarchical routing tiles

Efficient navigation routing also requires having road data at the right level of detail. For example, for cross country routing, it would be slow to run the routing algorithm against a highly detailed set of street-level routing tiles. The graph stitched together from these detailed routing tiles would likely be too large and consume too much memory.

There are typically three sets of routing tiles with different levels of detail. At the most detailed level, the routing tiles are small and contain only local roads. At the next level, the tiles are bigger and contain only arterial roads connecting districts together. At the lowest level of detail, the tiles cover large areas and contain only major highways connecting cities and states together. At each level, there could be edges connecting to tiles at a different zoom level. For example, for a freeway entrance from local street A to freeway F, there would be a reference from the node (street A) in the small tile to the node (freeway F) in the big tile. See Figure 3.6 for an example of routing tiles of varying sizes.

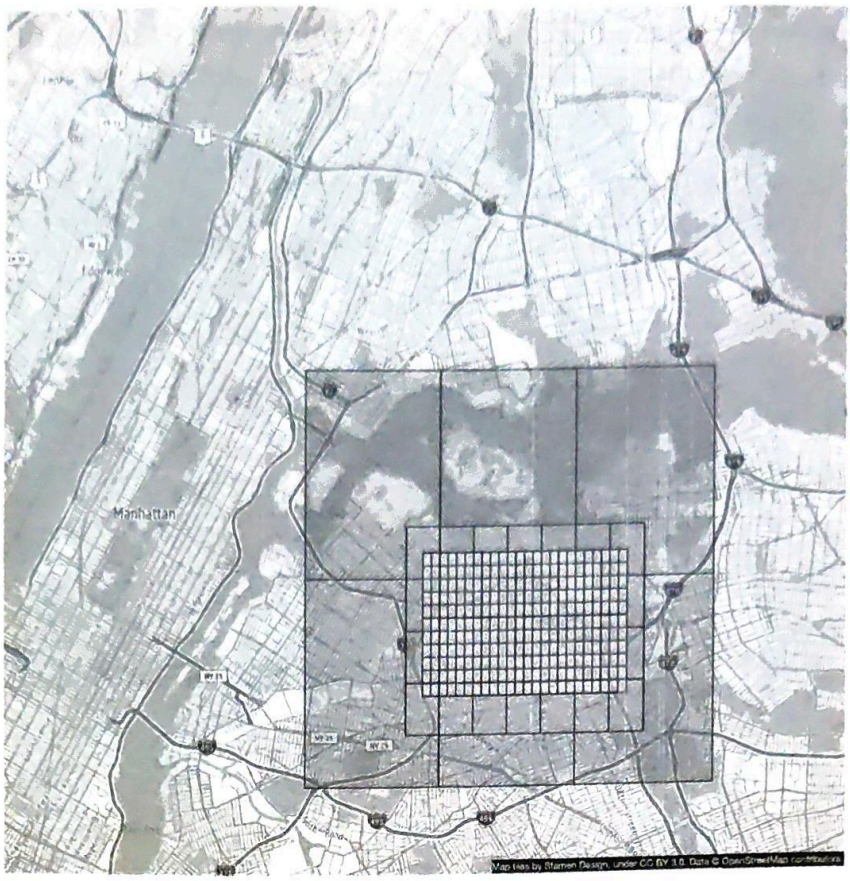


Figure 3.6: Routing tiles of varying sizes

Back-of-the-envelope estimation

Now that we understand the basics, let's do a back-of-the-envelope estimation. Since the focus of the design is mobile, data usage and battery consumption are two important factors to consider.

Before we dive into the estimation, here are some imperial/metric conversions for reference.

- 1 foot = 0.3048 meters
- 1 kilometer (km) = 0.6214 miles
- 1 km = 1,000 meters

Storage usage

We need to store three types of data.

- Map of the world: A detailed calculation is shown below.
- Metadata: Given that the metadata for each map tile could be negligible in size, we can skip the metadata in our computation.
- Road info: The interviewer told us there are TBs of road data from external sources. We transform this dataset into routing tiles, which are also likely to be terabytes in size.

Map of the world

We discussed the concept of map tiling in the "Map 101" section on page 60. There are many sets of map tiles, with one at each zoom level. To get an idea of the storage requirement for the entire collection of map tile images, it would be informative to estimate the size of the largest tile set at the highest zoom level first. At zoom level 21, there are about 4.3 trillion tiles (Table 3.1). Let's assume that each tile is a 256×256 pixel compressed PNG image, with the image size of about 100KB. The entire set at the highest zoom level would need about 4.4 trillion $\times$ 100KB = 440PB.

In Table 3.1, we show the progression of tile counts at every zoom level.

Zoom	Number of Tiles
0	1
1	4
2	16
3	64
4	256
5	1 024
6	4 096
7	16 384
8	65 536
9	262 144
10	1 048 576
11	4 194 304
12	16 777 216
13	67 108 864
14	268 435 456
15	1 073 741 824
16	4 294 967 296
17	17 179 869 184
18	68 719 476 736
19	274 877 906 944
20	1 099 511 627 776
21	4 398 046 511 104

Table 3.1: Zoom levels

However, keep in mind that about 90% of the world's surface is natural and mostly uninhabited areas like oceans, deserts, lakes, and mountains. Since these areas are highly compressible as images, we could conservatively reduce the storage estimate by 80 $\sim$ 90%. That would reduce the storage size to a range of 44 to 88PB. Let's pick a simple round number of 50PB.

Next, let's estimate how much storage each subsequent lower zoom level would take. At each lower zoom level, the number of tiles for both north-south and east-west directions drops by half. This results in a total reduction of the number of tiles by 4x, which drops

the storage size for the zoom level also by 4x. With the storage size reduced by 4x at each lower zoom level, the math for the total size is a series: $50 + \frac{50}{4} + \frac{50}{16} + \frac{50}{64} + \cdots =\sim 67PB$. This is just a rough estimate. It is good enough to know that we need roughly about 100PB to store all the map tiles at varying levels of detail.

Server throughput

To estimate the server throughput, let's review the types of requests we need to support. There are two main types of requests. The first is navigation requests. These are sent by the clients to initiate a navigation session. The second is location update requests. These are sent by the client as the user moves around during a navigation session. The location data is used by downstream services in many different ways. For example, location data is one of the inputs for live traffic data. We will cover the use cases of location data in the design deep dive section.

Now we can analyze the server throughput for navigation requests. Let's assume we have 1 billion DAU, and each user on average uses navigation for a total of 35 minutes per week. This translates to 35 billion minutes per week or 5 billion minutes per day.

One simple approach would be to send GPS coordinates every second, which results in 300 billion (5 billion minutes $\times$ 60) requests per day, or 3 million QPS $\left(\frac{300 \text{ billion requests}}{10^5} = 3 \text{ million}\right)$. However, the client may not need to send a GPS update every second. We can batch these on the client and send them at a much lower frequency (for example, every 15 seconds or 30 seconds) to reduce the write QPS. The actual frequency could depend on factors such as how fast the user moves. If they are stuck in traffic, a client can slow down the GPS updates. In our design, we assume GPS updates are batched and then sent to the server every 15 seconds. With this batched approach, the QPS is reduced to 200,000 $\left(\frac{3 \text{ million}}{15}\right)$.

Assume peak QPS is five times the average. Peak QPS for location updates = 200,000 $\times$ 5 = 1 million.

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

Now that we have more knowledge about Google Maps, we are ready to propose a high-level design (Figure 3.7).

High-level design

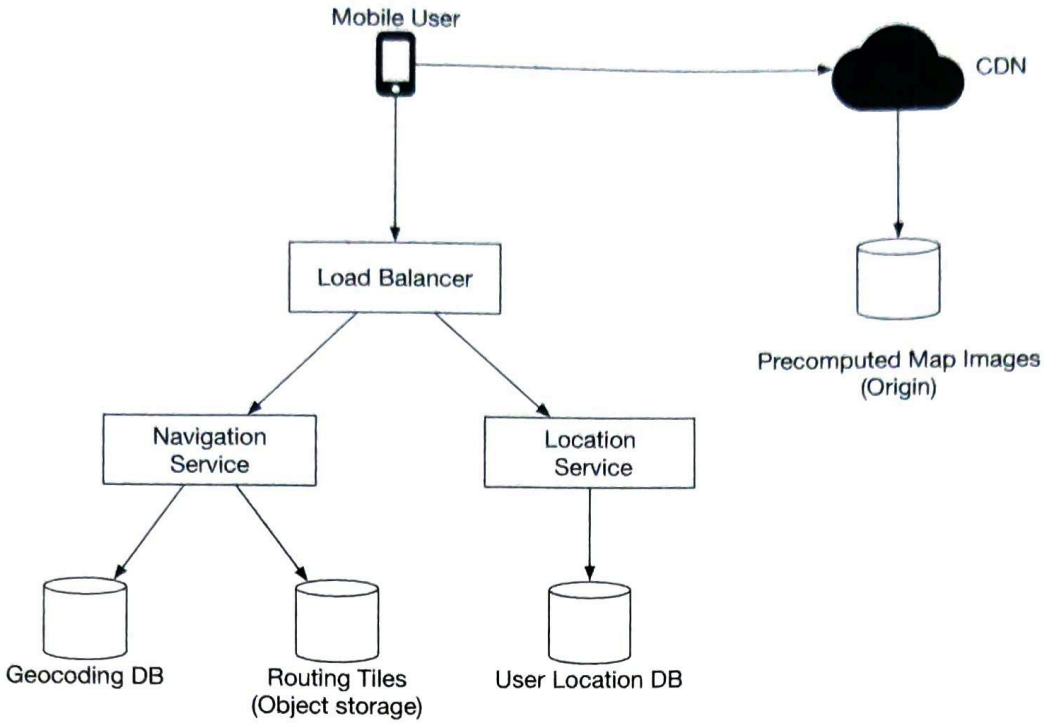


Figure 3.7: High-level design

The high-level design supports three features. Let's take a look at them one by one.

1. Location service
2. Navigation service
3. Map rendering

Location service

The location service is responsible for recording a user's location update. The architecture is shown in Figure 3.8.

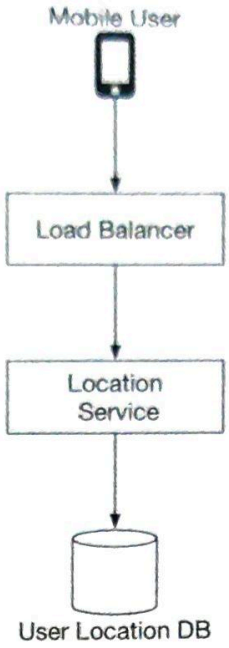


Figure 3.8: Location service

The basic design calls for the clients to send location updates every t seconds, where t is a configurable interval. The periodic updates have several benefits. First, we can leverage the streams of location data to improve our system over time. We can use the data to monitor live traffic, detect new or closed roads, and analyze user behavior to enable personalization, for example. Second, we can leverage the location data in near real-time to provide more accurate ETA estimates to the users and to reroute around traffic, if necessary.

But do we really need to send every location update to the server immediately? The answer is probably no. Location history can be buffered on the client and sent in batch to the server at a much lower frequency. For example, as shown in Figure 3.9, the location updates are recorded every second, but are only sent to the server as part of a batch every 15 seconds. This significantly reduces the total update traffic sent by all the clients.

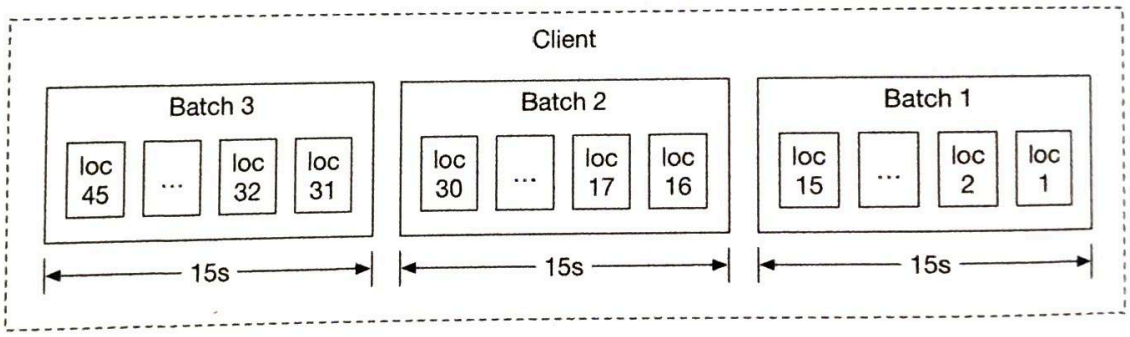


Figure 3.9: Batch requests

For a system like Google Maps, even when location updates are batched, the write volume is still very high. We need a database that is optimized for high write volume and is highly scalable, such as Cassandra. We may also need to log location data using a stream

processing engine such as Kafka for further processing. We discuss this in detail in the deep dive section.

What communication protocol might be a good fit here? HTTP with the keep-alive option [10] is a good choice because it is very efficient. The HTTP request might look like this:

```
POST /v1/locations
Parameters
locs: JSON encoded array of (latitude, longitude, timestamp)
  tuples.
```

Navigation service

This component is responsible for finding a reasonably fast route from point A to point B. We can tolerate a little bit of latency. The calculated route does not have to be the fastest, but accuracy is critical.

As shown in Figure 3.8, the user sends an HTTP request to the navigation service through a load balancer. The request includes origin and destination as the parameters. The API might look like this:

```
GET /v1/nav?origin=1355+market+street,SF&destination=
Disneyland
```

Here is an example of what the navigation result could look like:

```
{
  'distance': {'text':'0.2 mi', 'value': 259},
  'duration': {'text': '1 min', 'value': 83},
  'end_location': {'lat': 37.4038943, 'lng': -121.9410454},
  'html_instructions': 'Head <b>northeast</b> on <b>Brandon St
      </b> toward <b>Lumin Way</b><div style="font-size:0.9em">
      Restricted usage road</div>',
  'polyline': {'points': '_fhcFjbhgVuAwDsCal'},
  'start_location': {'lat': 37.4027165, 'lng': -121.9435809},
  'geocoded_waypoints': [
  {
    "geocoder_status" : "OK",
    "partial_match" : true,
    "place_id" : "ChIJwZNMti1fawwRO2aVVVX2yKg",
    "types" : [ "locality", "political" ]
  },
  {
    "geocoder_status" : "OK",
    "partial_match" : true,
    "place_id" : "ChIJ3aPgQGtXawwRLYeiBMUi7bM",
    "types" : [ "locality", "political" ]
  }
  ],
  'travel_mode': 'DRIVING'
}
```

Please refer to [11] for more details on Google Maps' official APIs.

So far we have not taken reroute and traffic changes into consideration. Those problems are tackled by the Adaptive ETA service in the deep dive section.

Map rendering

As we discussed in the back-of-the-envelope estimation, the entire collection of map tiles at various zoom levels is about a hundred petabytes in size. It is not practical to hold the entire dataset on the client. The map tiles must be fetched on-demand from the server based on the client's location and the zoom level of the client viewport.

When should the client fetch new map tiles from the server? Here are some scenarios:

- The user is zooming and panning the map viewpoint on the client to explore their surroundings.
- During navigation, the user moves out of the current map tile into a nearby tile.

We are dealing with a lot of data. Let's see how we could serve these map tiles from the server efficiently.

Option 1

The server builds the map tiles on the fly, based on the client location and zoom level of the client viewport. Considering that there is an infinite number of location and zoom level combinations, generating map tiles dynamically has a few severe disadvantages:

- It puts a huge load on the server cluster to generate every map tile dynamically.
- Since the map tiles are dynamically generated, it is hard to take advantage of caching.

Option 2

Another option is to serve a pre-generated set of map tiles at each zoom level. The map tiles are static, with each tile covering a fixed rectangular grid using a subdivision scheme like geohashing. Each tile is therefore represented by its geohash. In other words, there is a unique geohash associated with each grid. When a client needs a map tile, it first determines the map tile collection to use based on its zoom level. It then computes the map tile URL by converting its location to the geohash at the appropriate zoom level.

These static, pre-generated images are served by a CDN as shown in Figure 3.10.

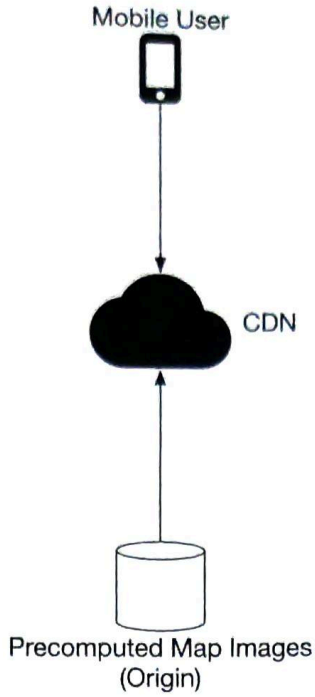


Figure 3.10: Pre-generated images are served by a CDN

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.

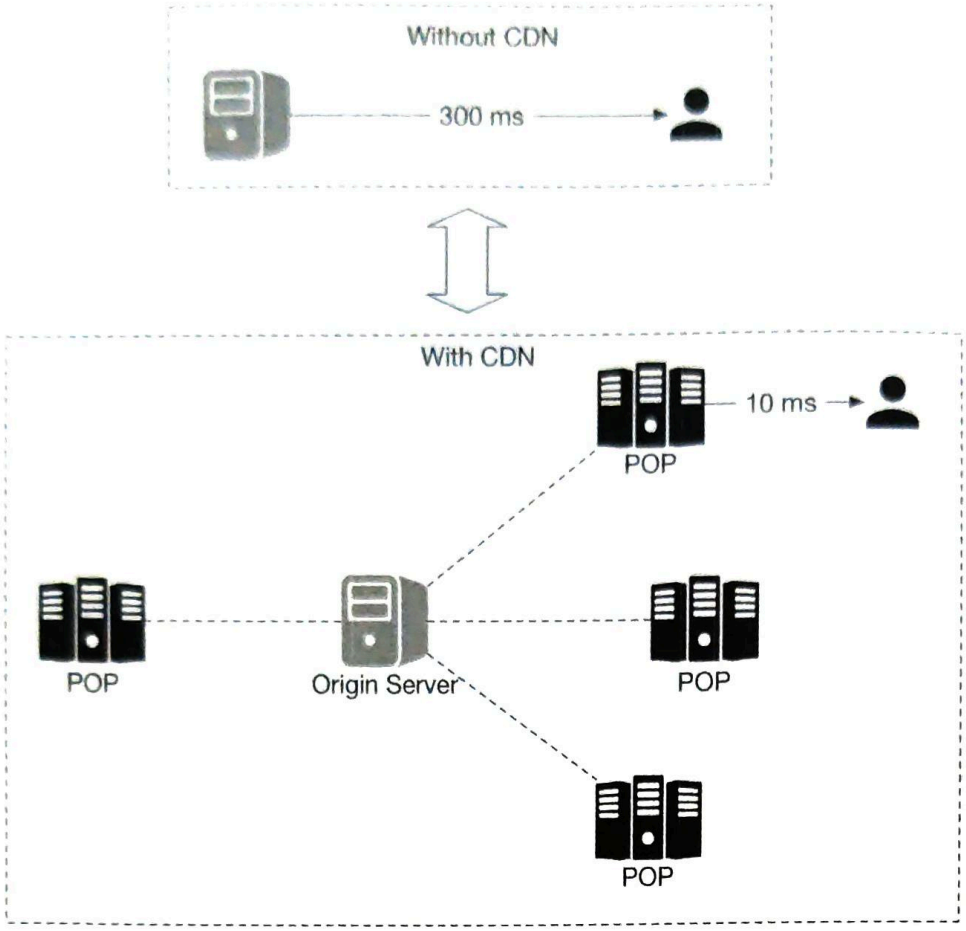


Figure 3.11: Without CDN vs with CDN

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 × 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 × 1km, we need 25 images or 2.5MB (25 × 100KB) of data. Therefore, if the speed is 30km/h, we need 75MB (30 × 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.

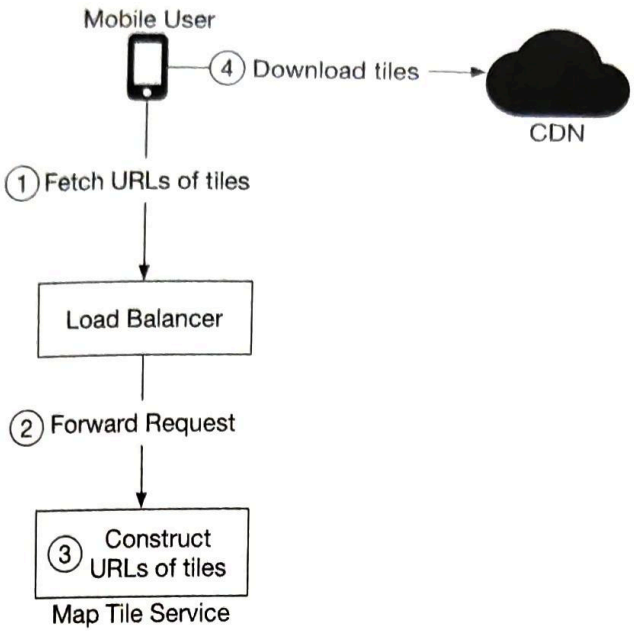


Figure 3.12: Map rendering

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:

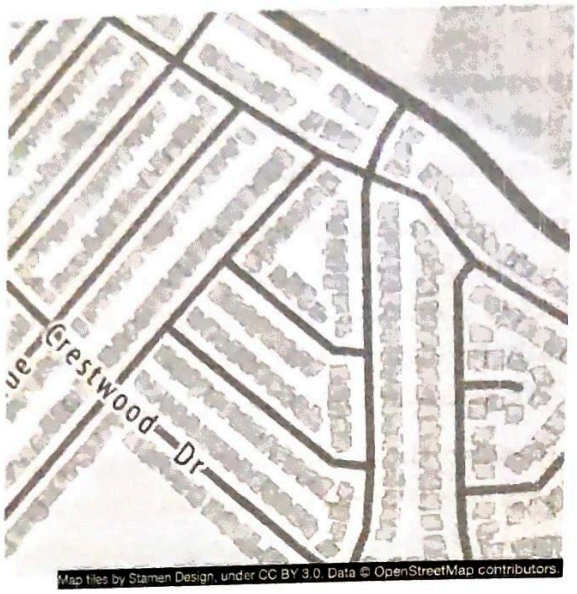


Figure 3.13: Precomputed tiles

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.

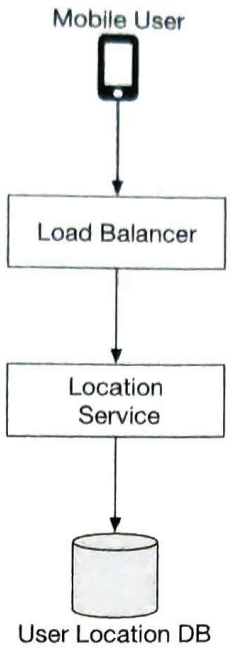


Figure 3.14: User location database

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.

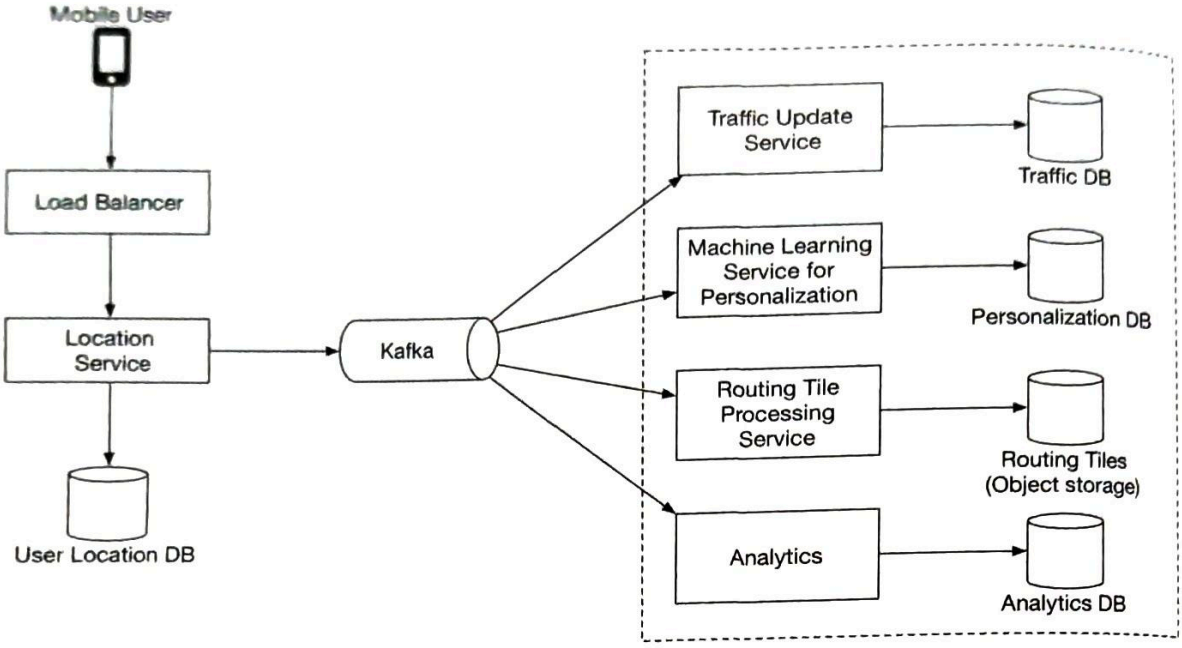


Figure 3.15: Location data is used by other services

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.

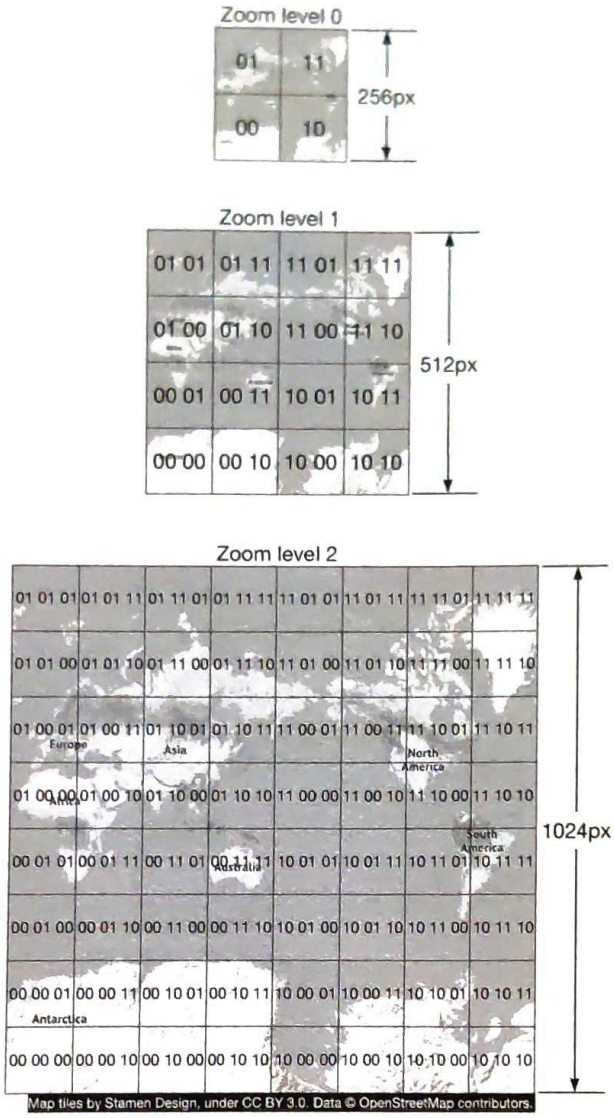


Figure 3.16: Zoom levels

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.

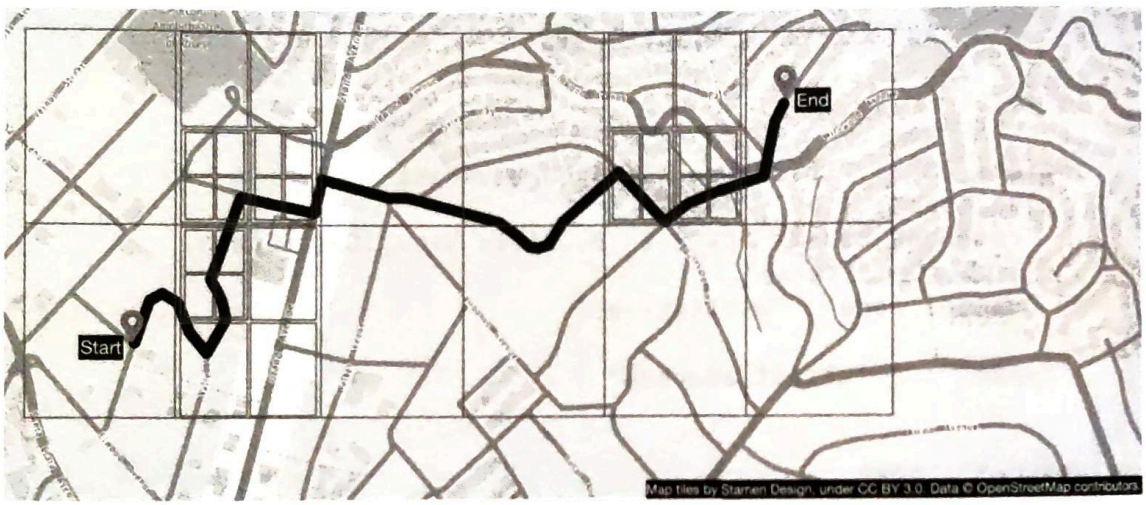


Figure 3.18: Graph traversal

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`.

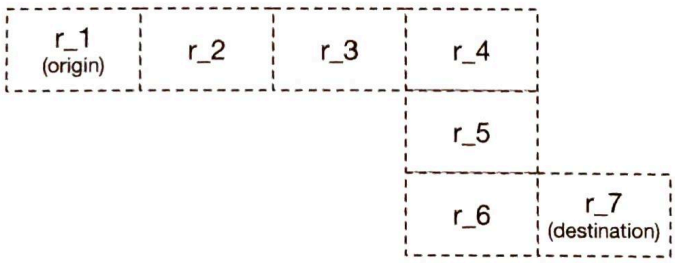


Figure 3.19: Navigation route

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)), ...
```

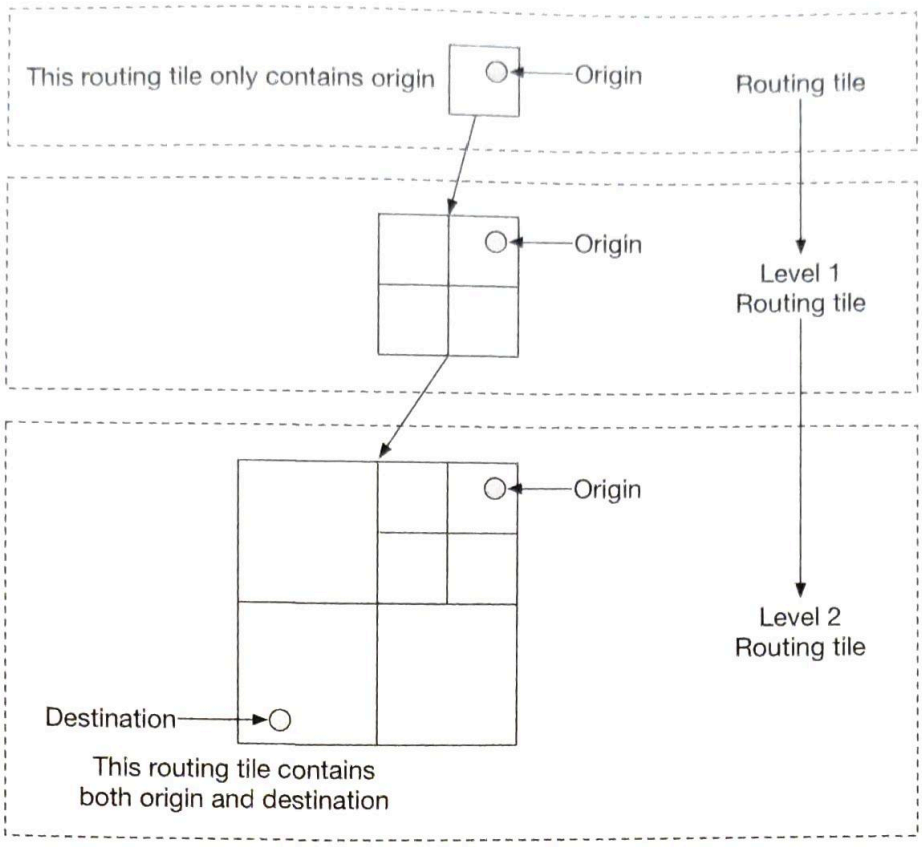


Figure 3.20: Build routing tiles

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.

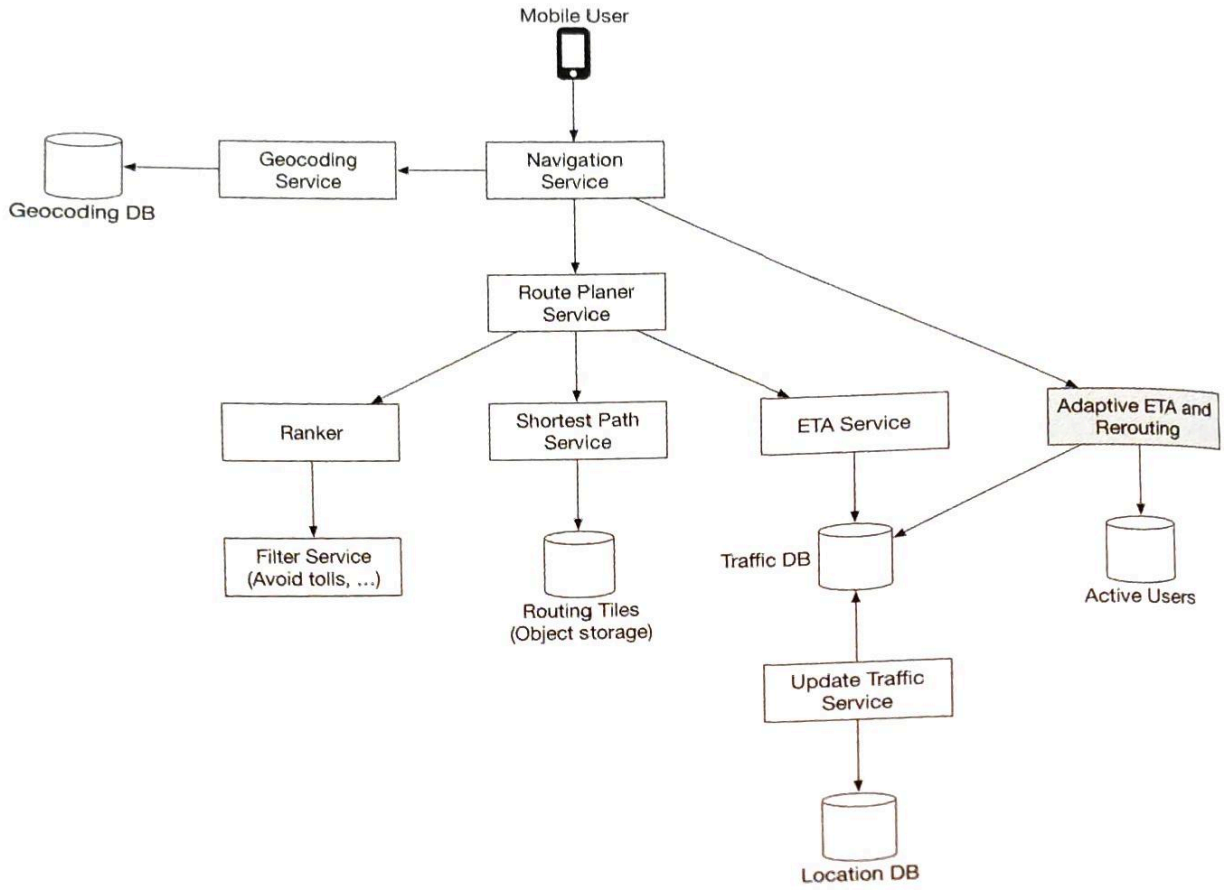


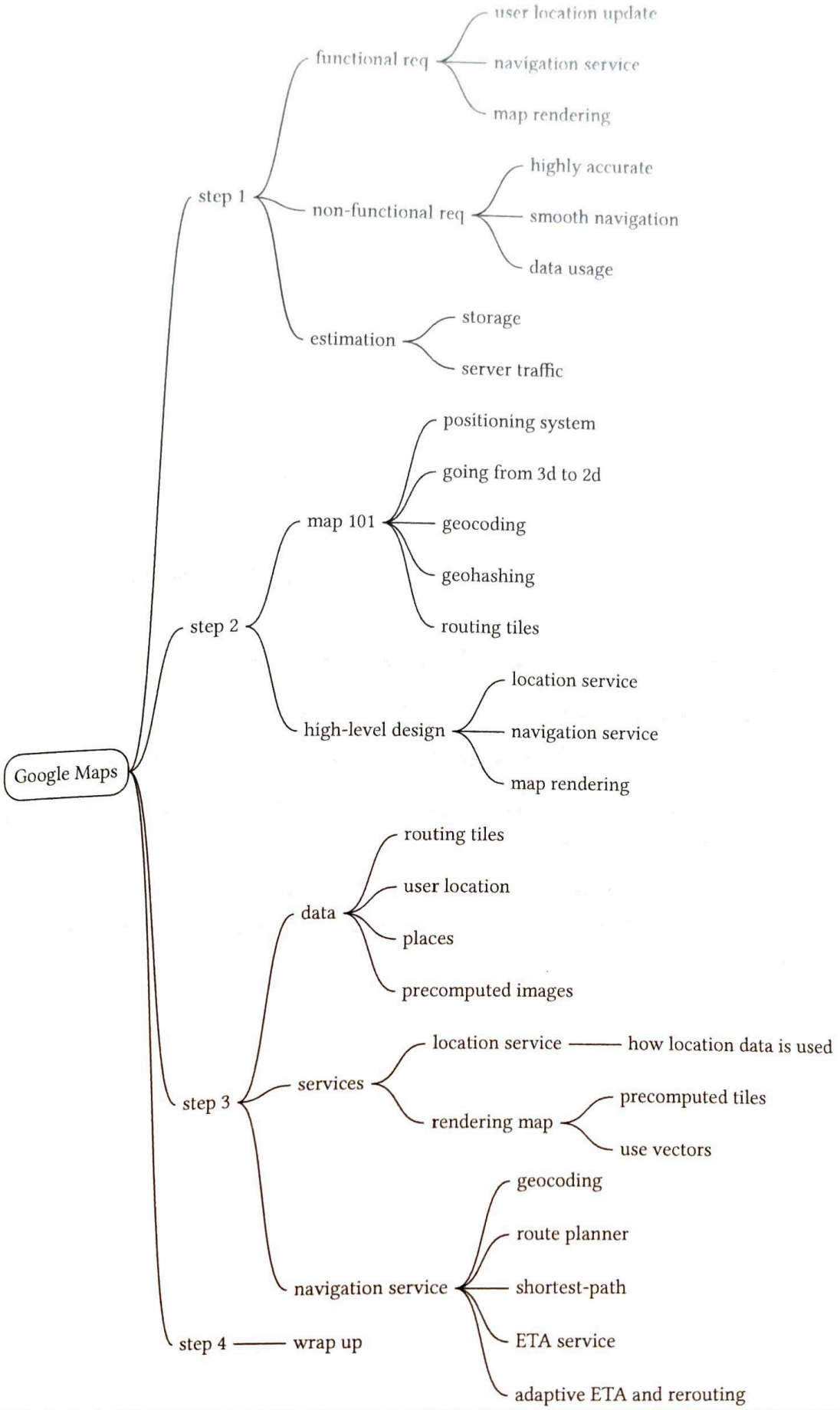
Figure 3.21: Final design

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.

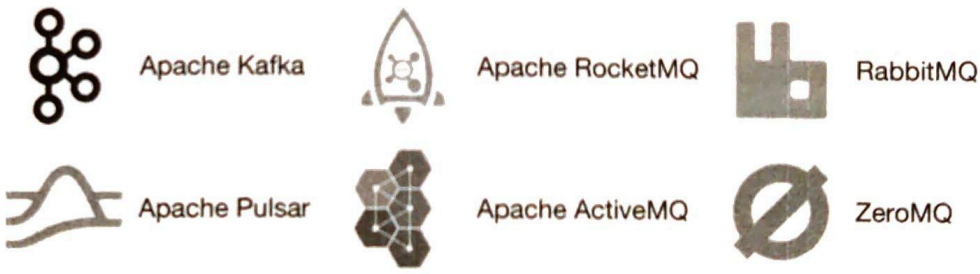


Figure 4.1: Popular distributed message queues

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

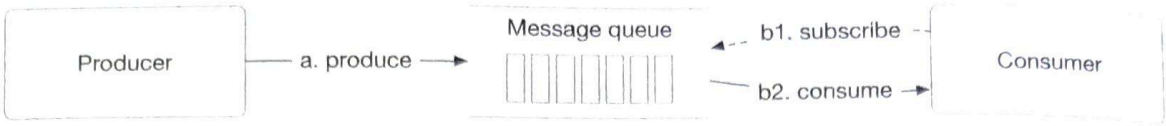


Figure 4.2: Key components in a message queue

- 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.

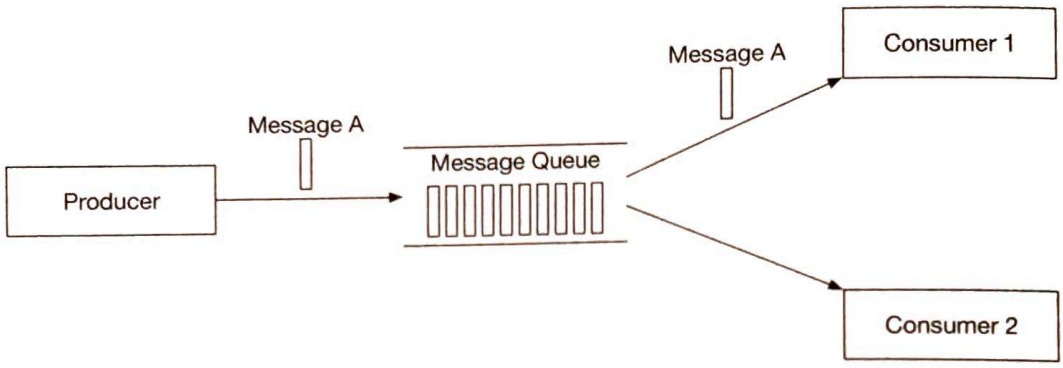


Figure 4.3: Point-to-point model

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.

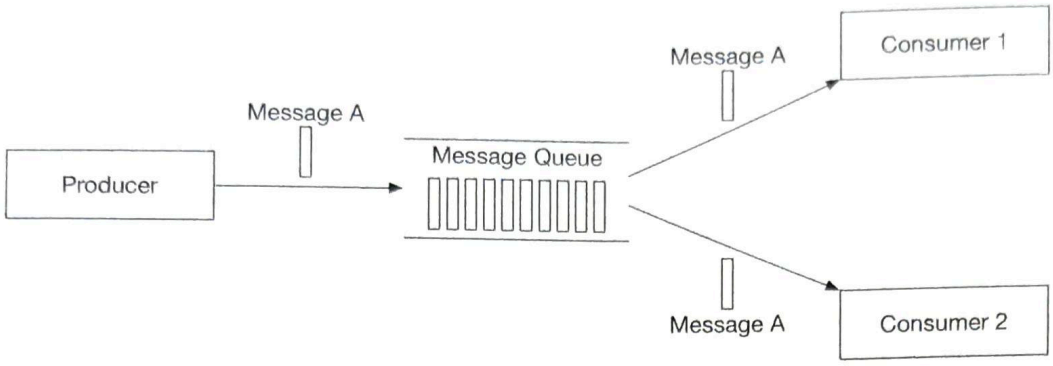


Figure 4.4: Publish-subscribe model

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.

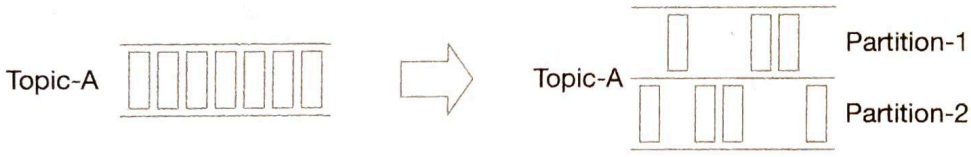


Figure 4.5: Partitions

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.

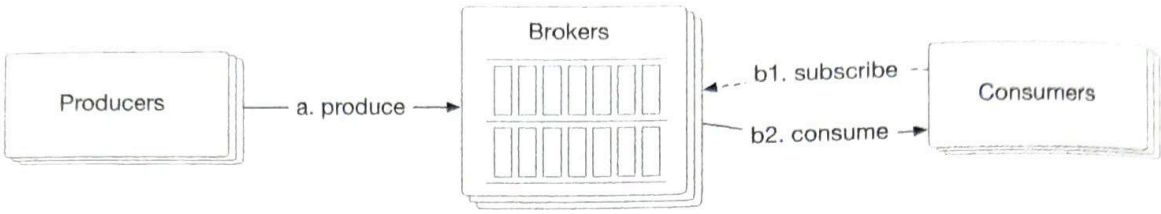


Figure 4.6: Message queue cluster

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.

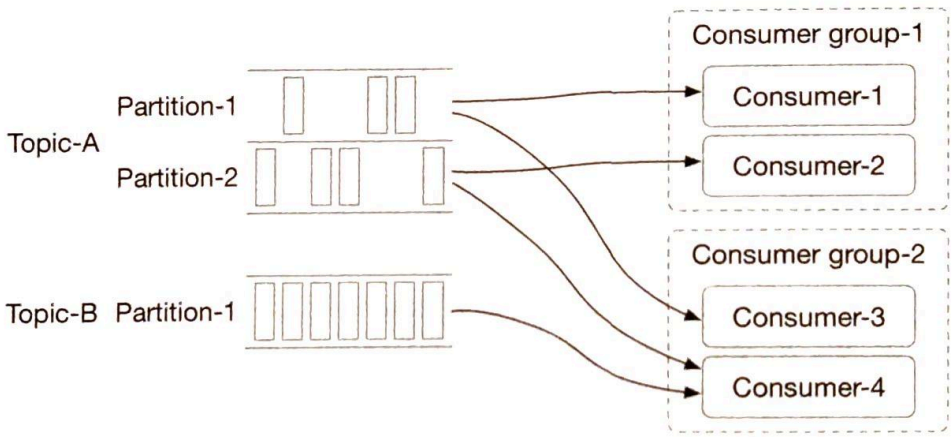


Figure 4.7: Consumer groups

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.

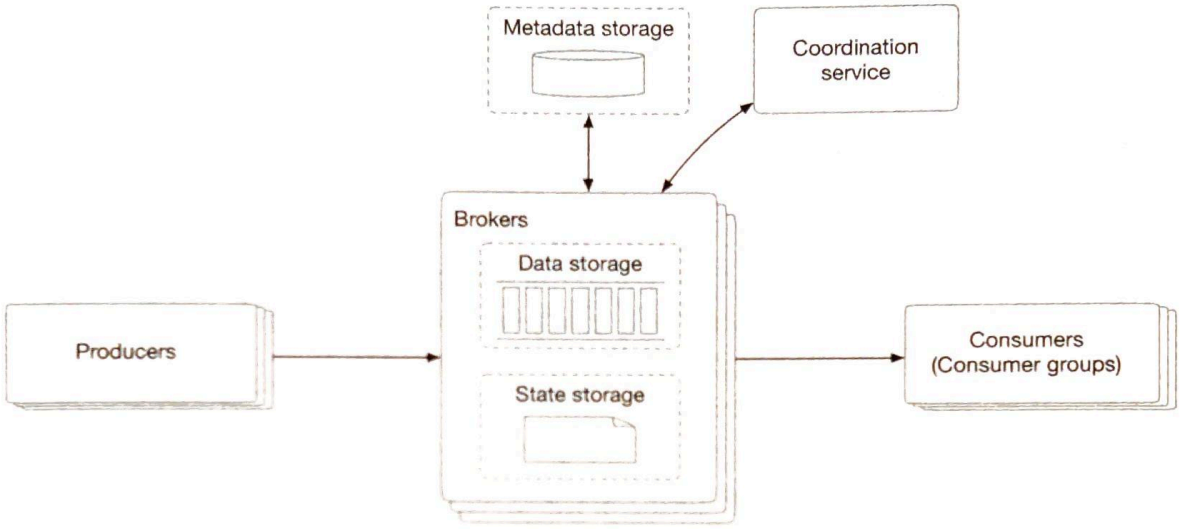


Figure 4.8: High-level design

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.

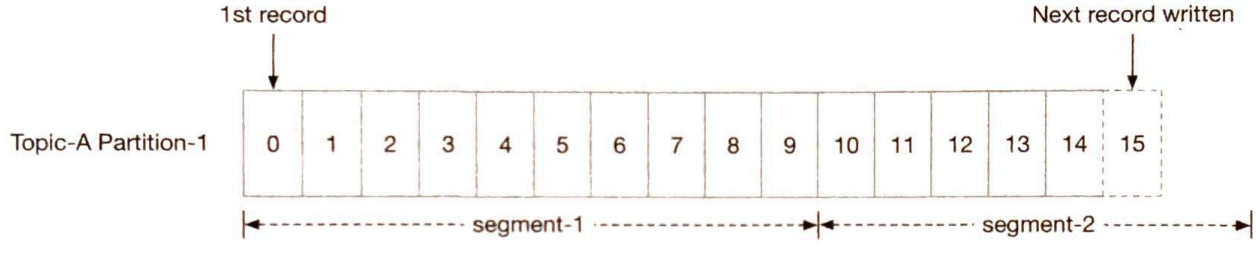


Figure 4.9: Append new messages

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

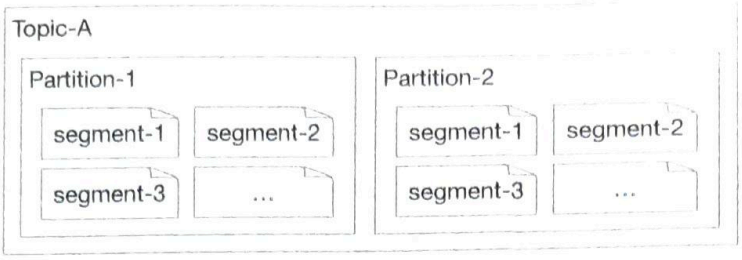


Figure 4.10: Data segment file distribution in topic partitions

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 messages 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.

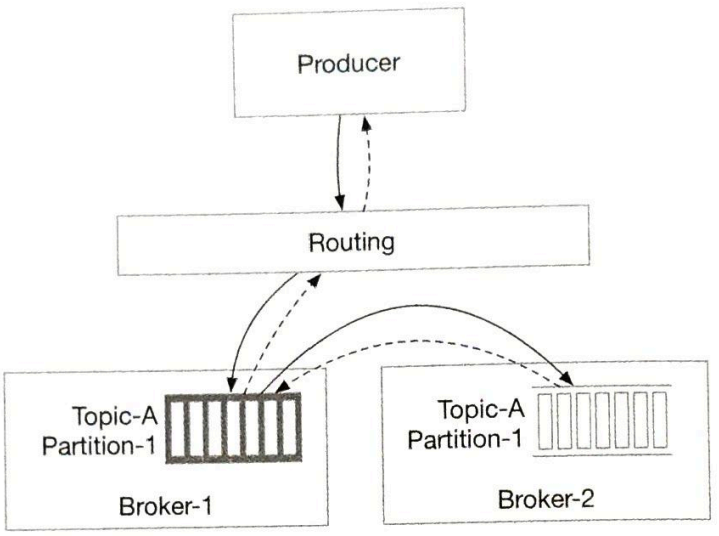


Figure 4.11: Routing layer

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.

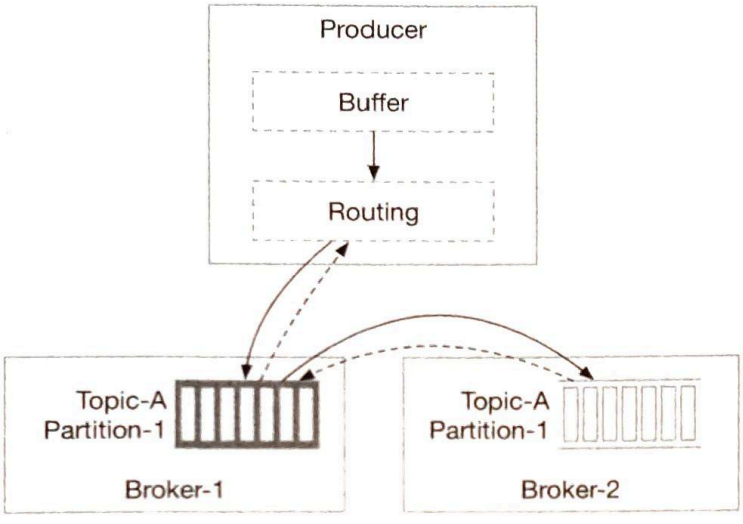


Figure 4.12: Producer with buffer and routing

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.

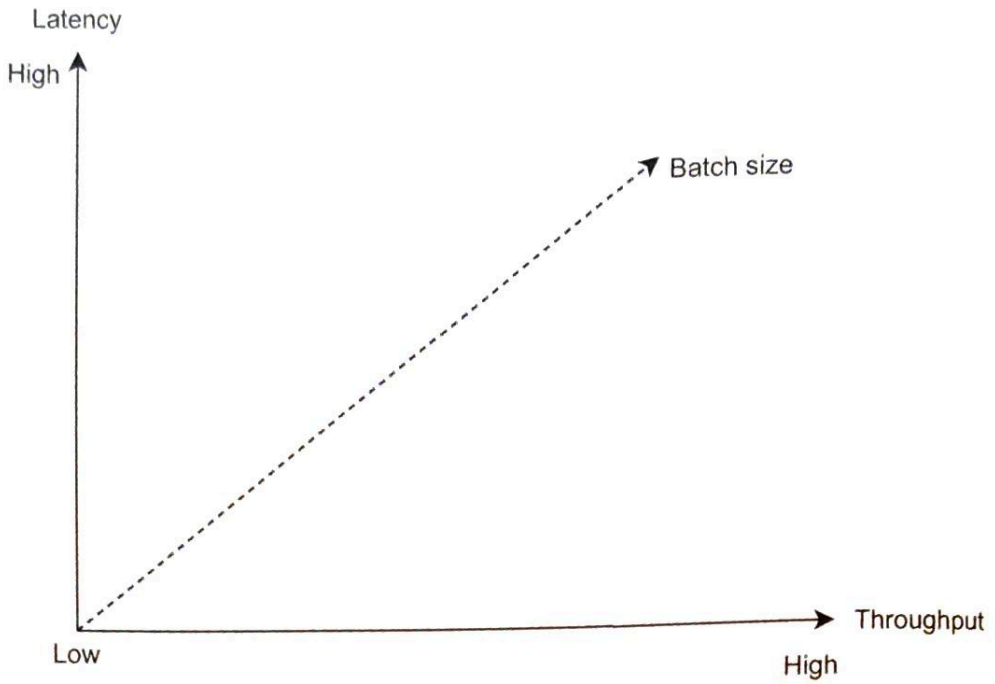


Figure 4.13: The choice of the batch size

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.

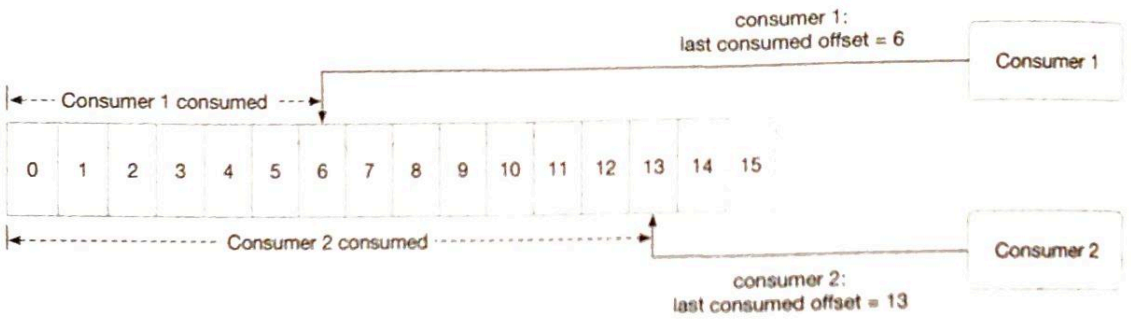


Figure 4.14: Consumer flow

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.

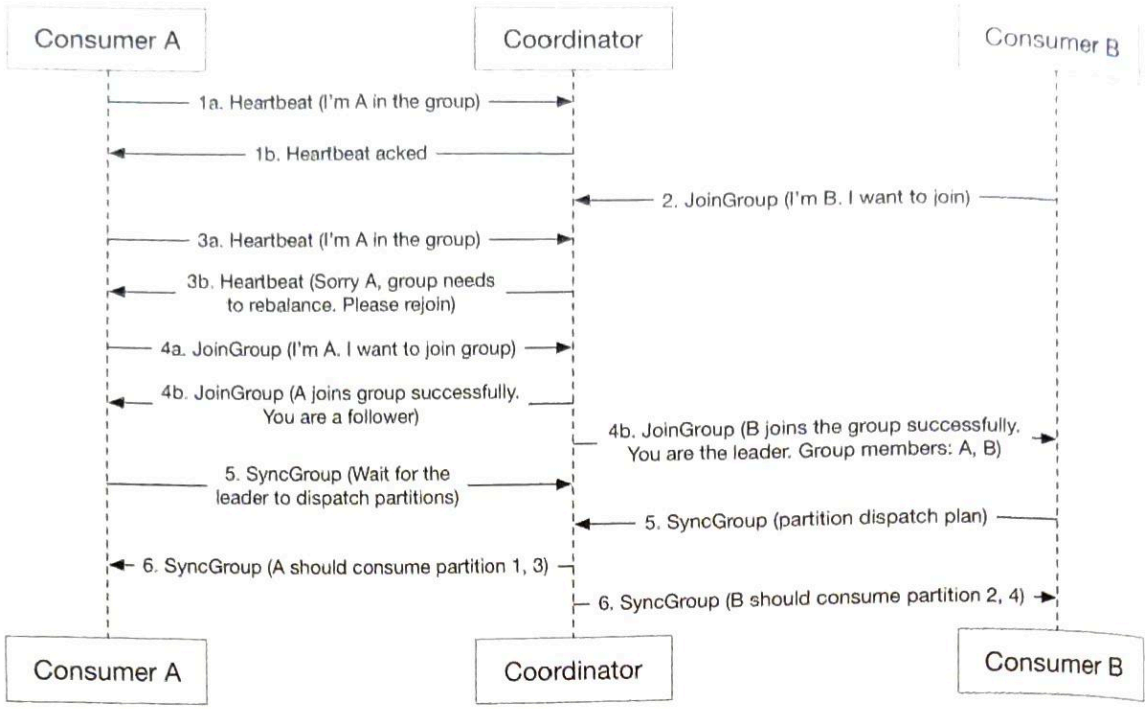


Figure 4.18: New consumer joins

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.

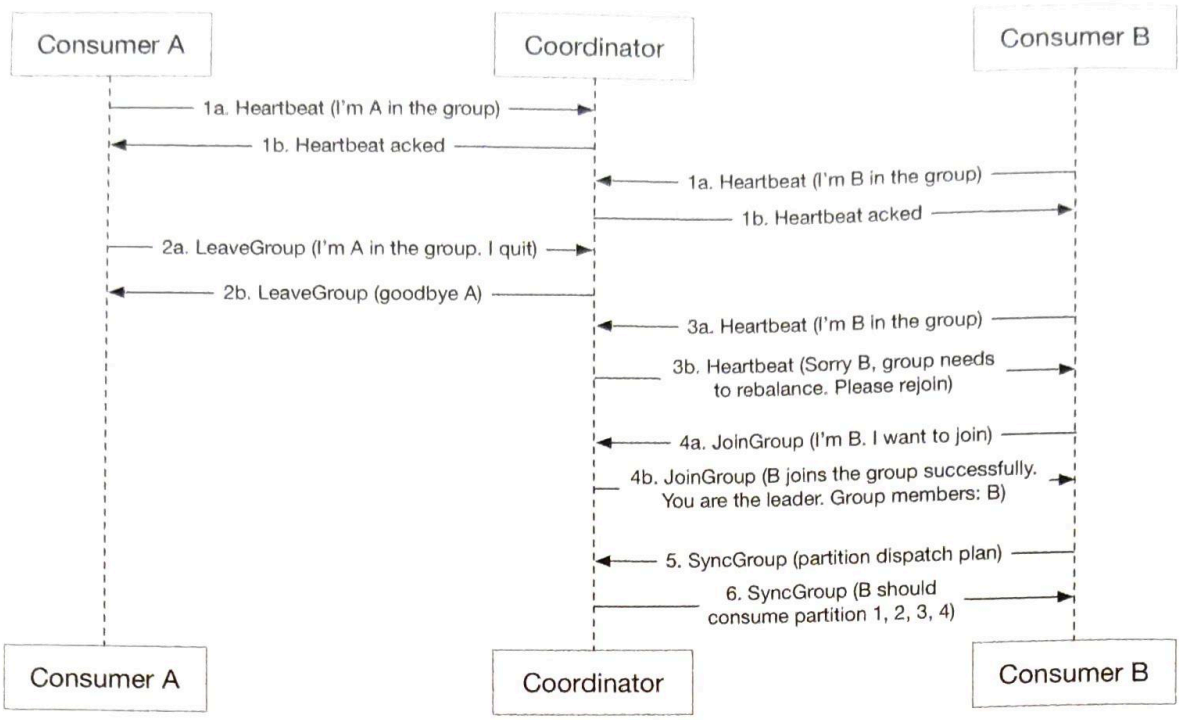


Figure 4.19: Existing consumer leaves

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.

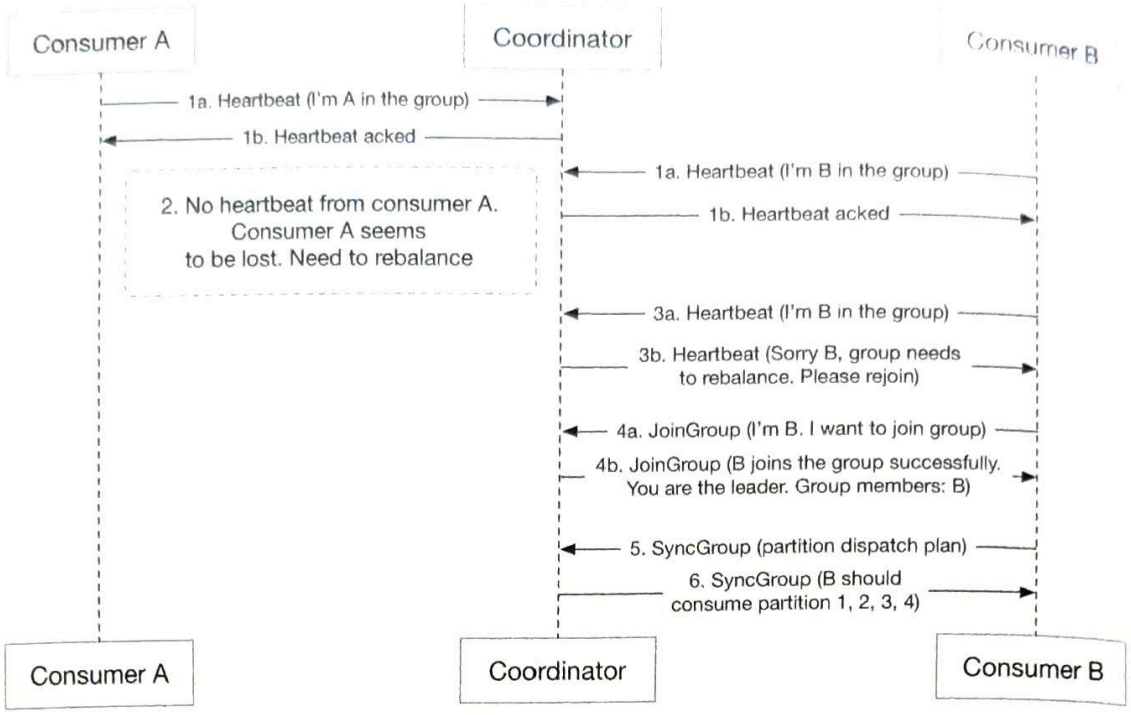


Figure 4.20: Existing consumer crashes

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.

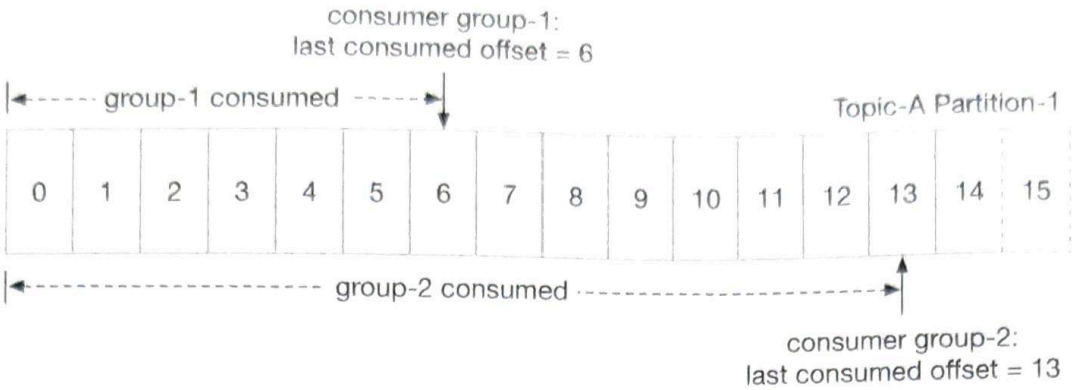


Figure 4.21: Last consumed offset of consumer groups

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.

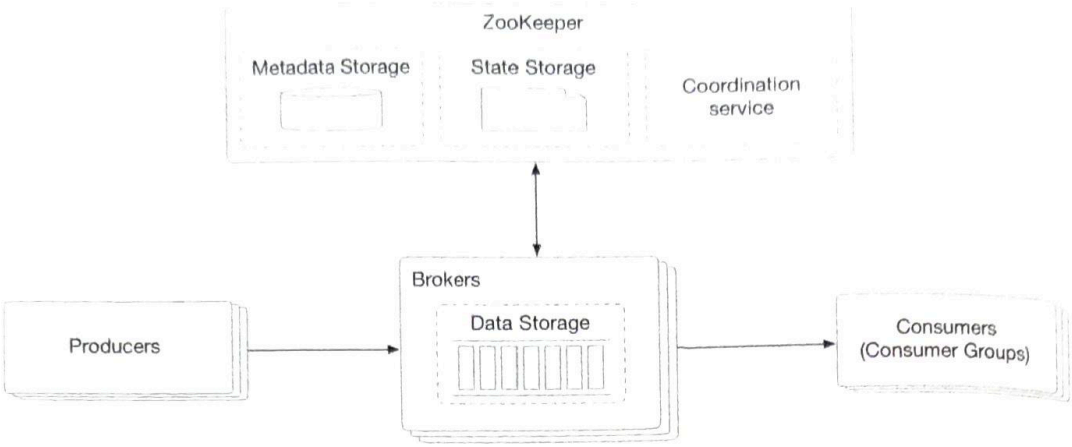


Figure 4.22: ZooKeeper

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.

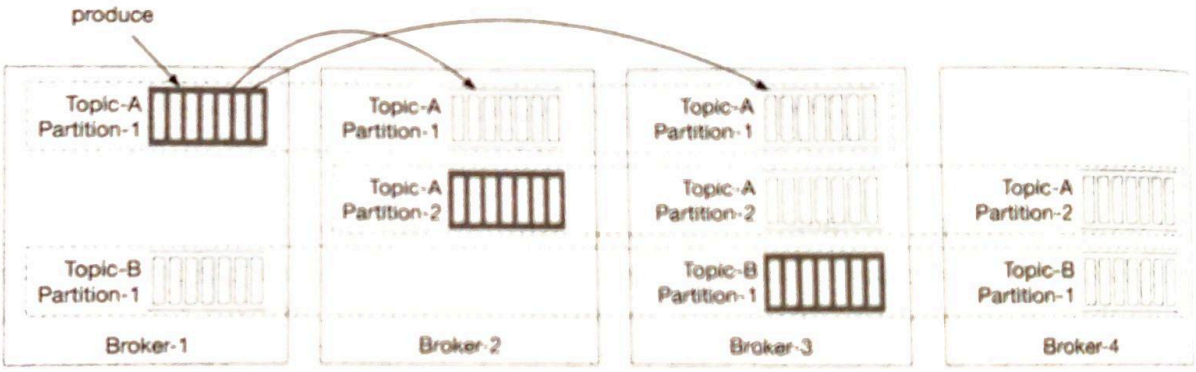


Figure 4.23: Replication

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.

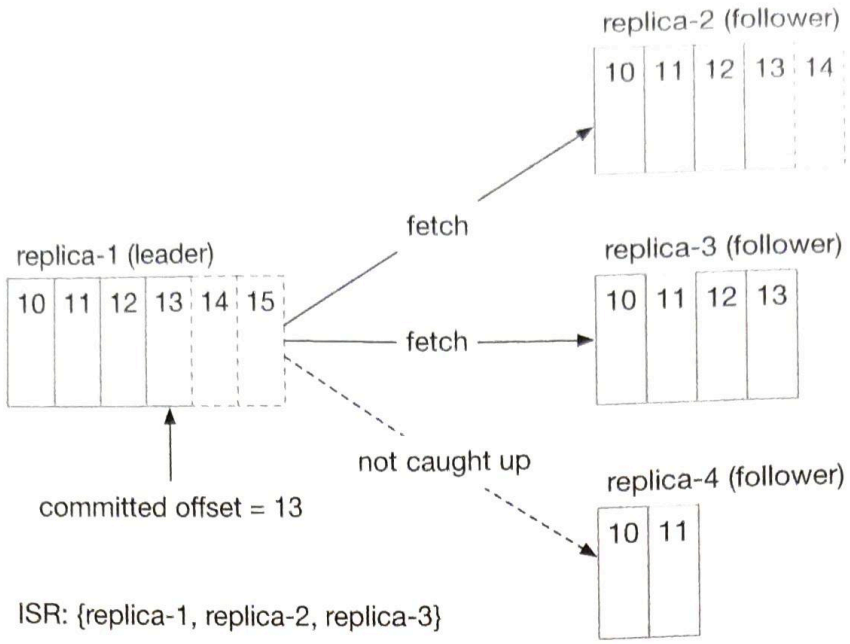


Figure 4.24: How ISR works

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.

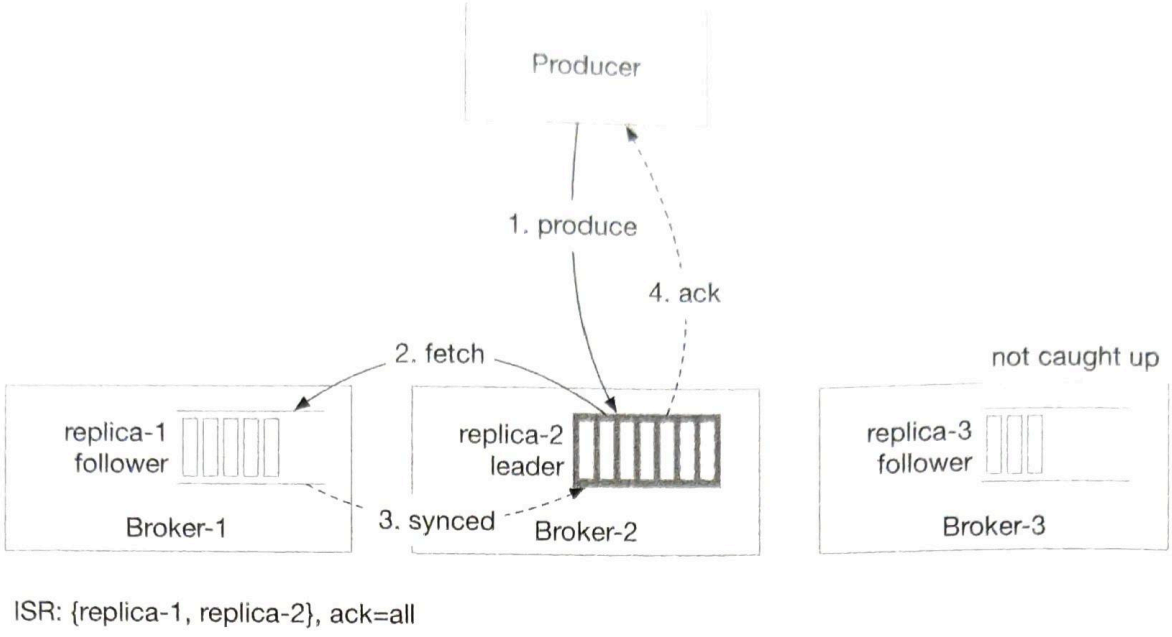


Figure 4.25: ACK=all

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.

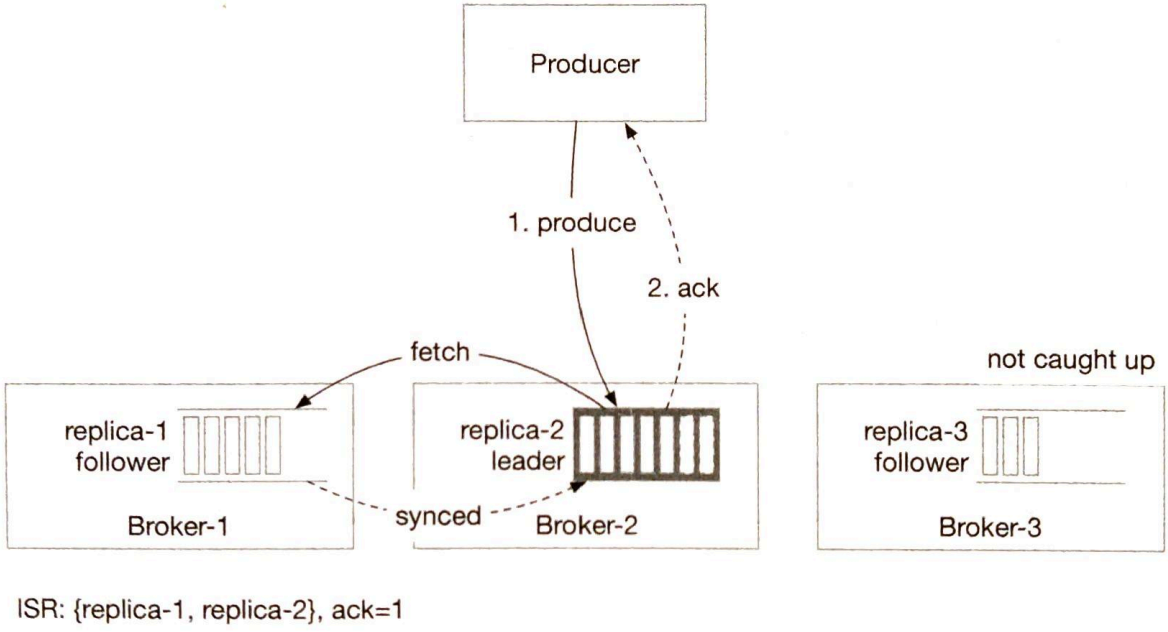


Figure 4.26: ACK=1

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.

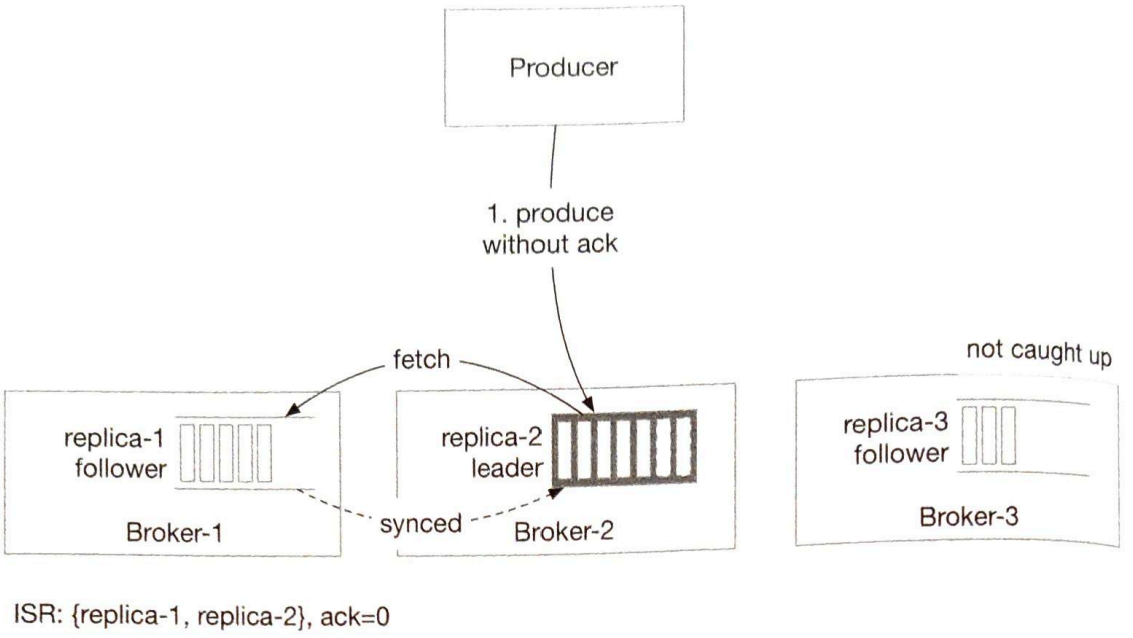


Figure 4.27: ACK=0

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.

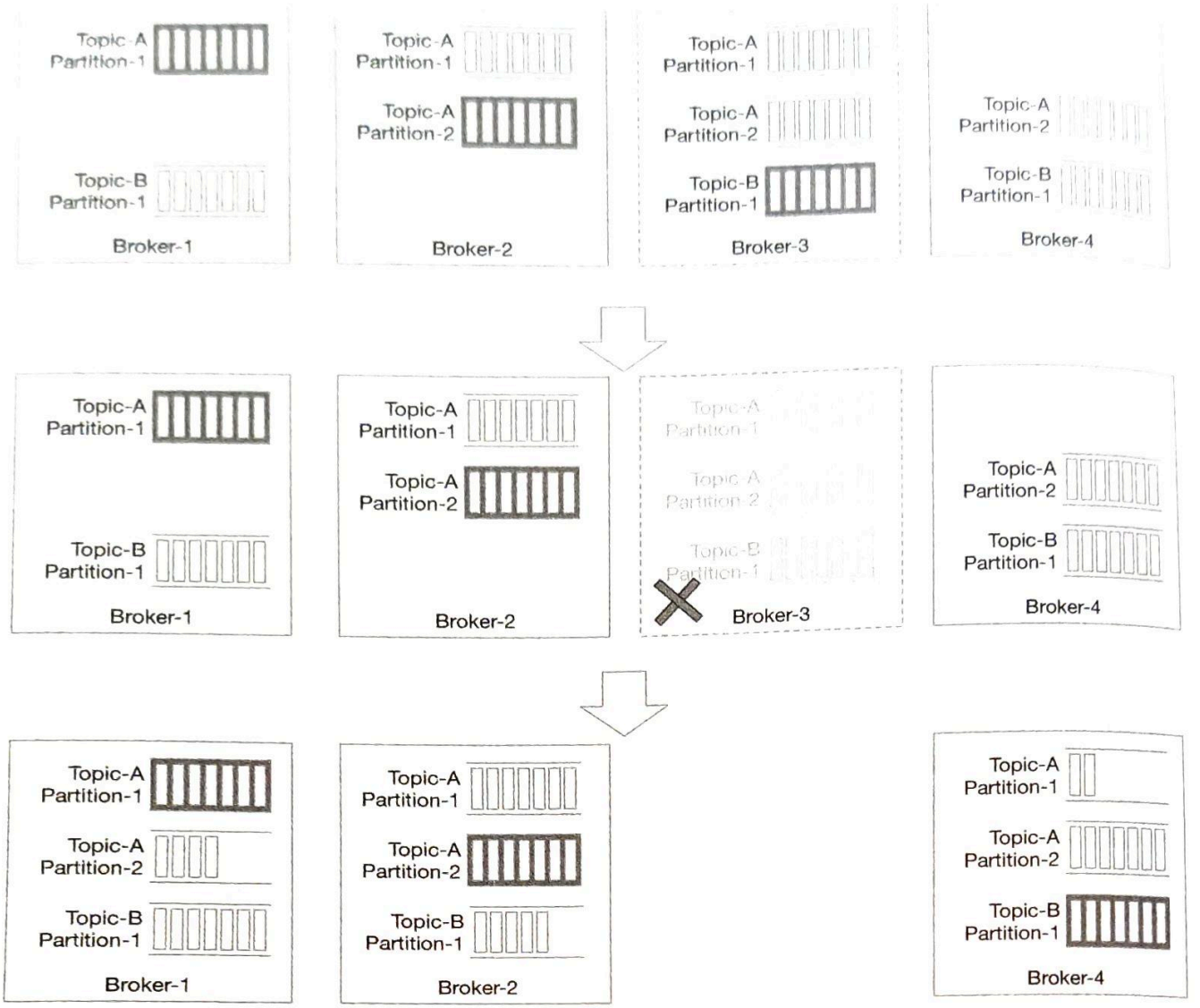


Figure 4.28: Broker node crashes

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.

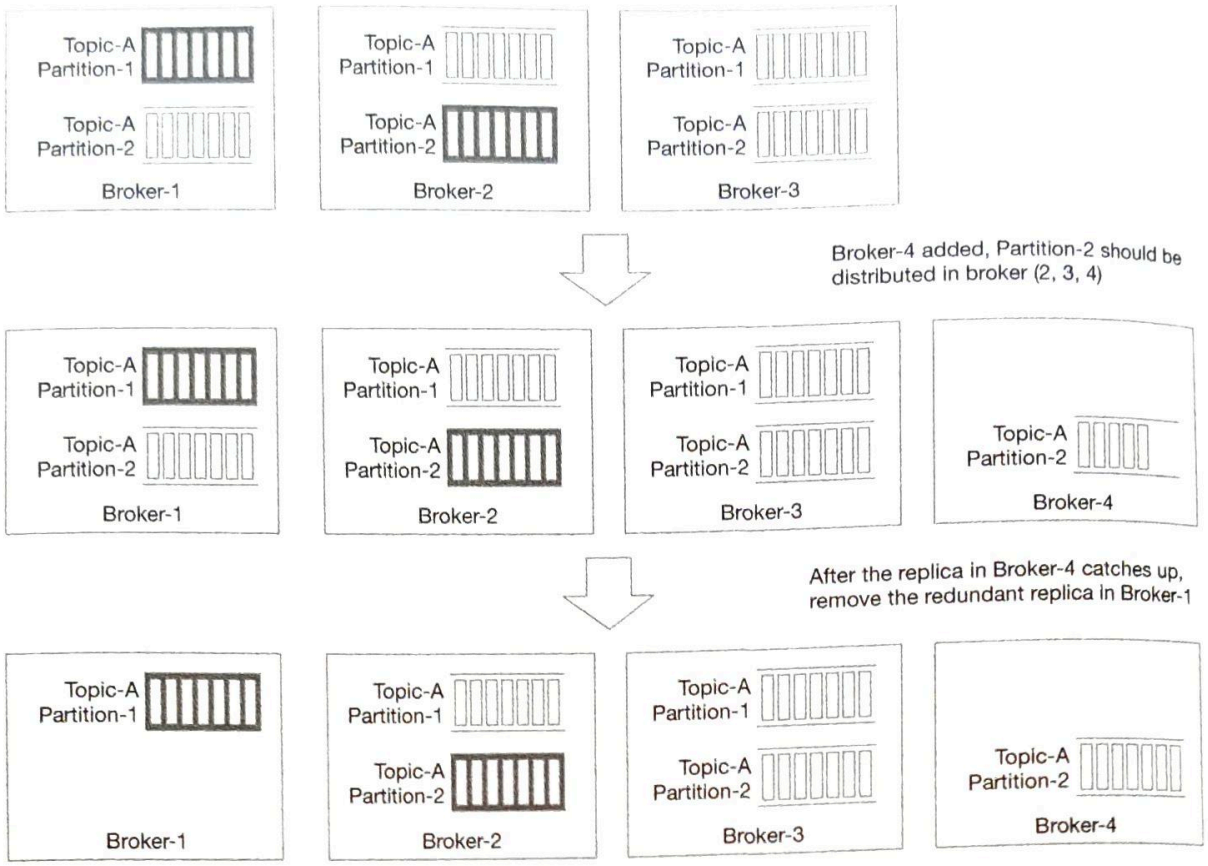


Figure 4.29: Add new broker node

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.

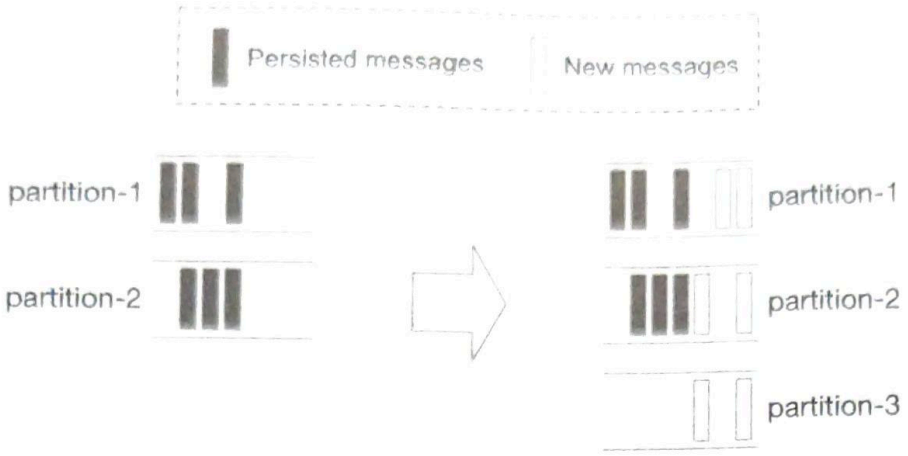


Figure 4.30: Partition increase

- 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.

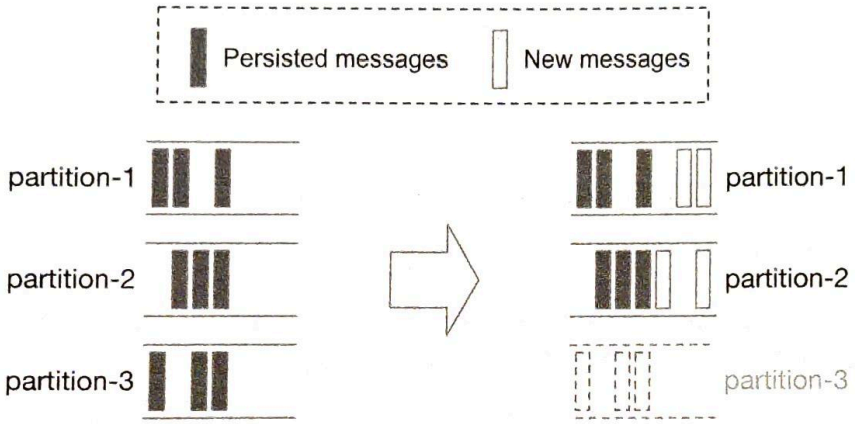


Figure 4.31: Partition decrease

- 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.

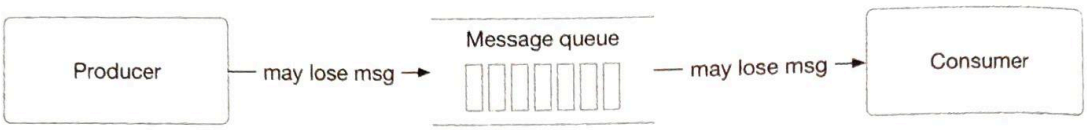


Figure 4.32: At-most once

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.

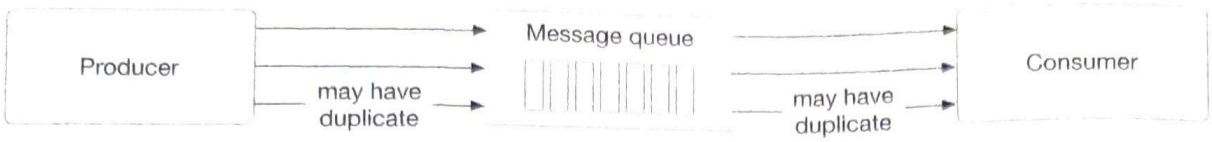


Figure 4.33: At-least once

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.

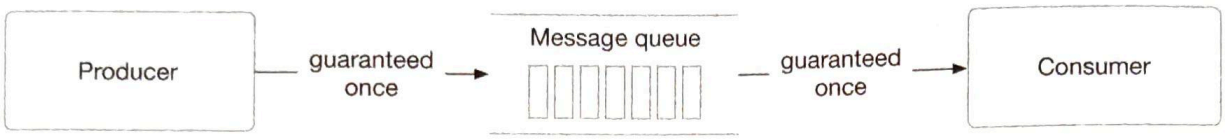


Figure 4.34: Exactly once

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].

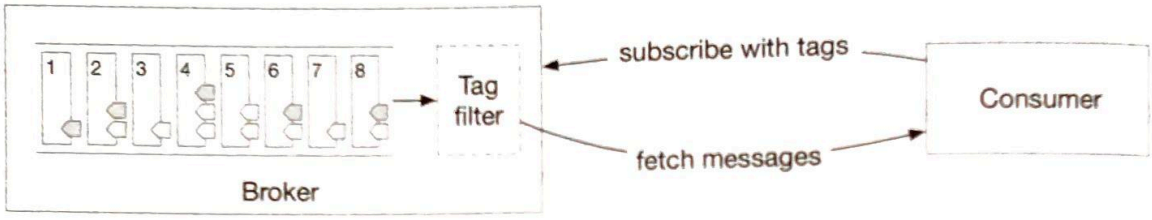


Figure 4.35: Message filtering by tags

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.

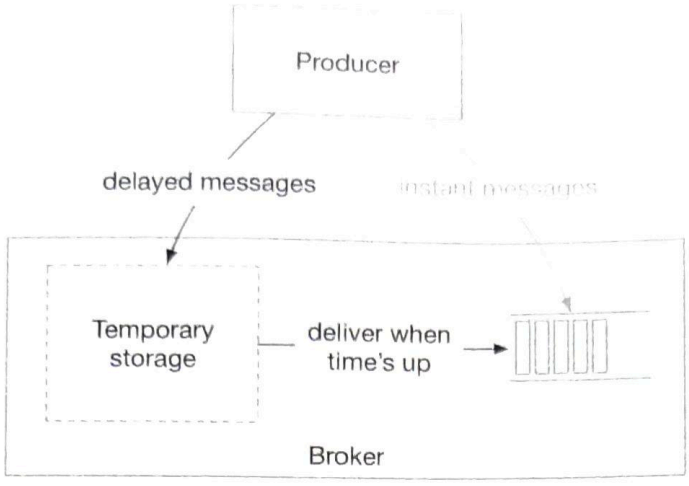


Figure 4.36: Delayed messages

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:

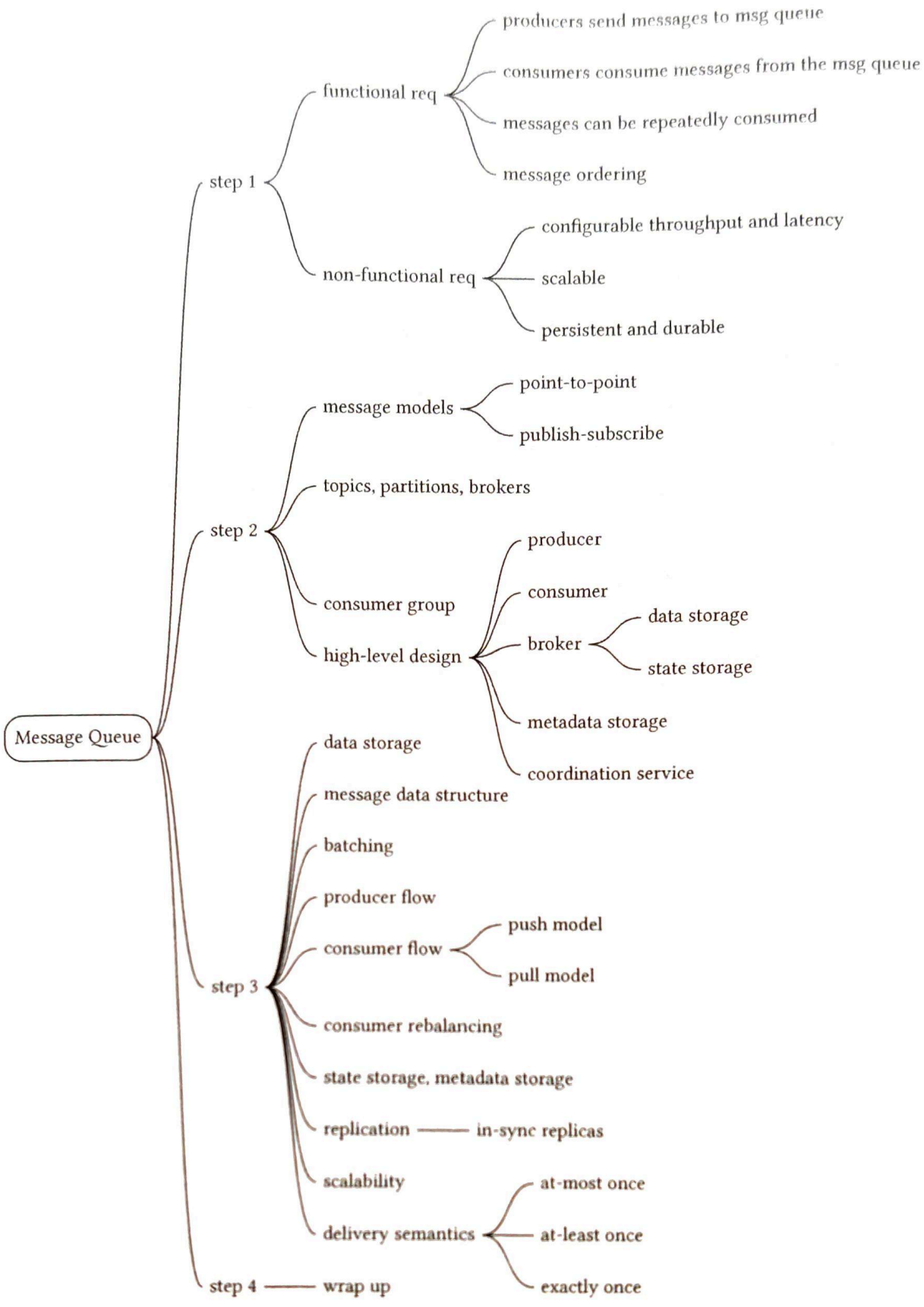
- Protocol: it defines rules, syntax, and APIs on how to exchange information and transfer data between different nodes. In a distributed message queue, the protocol should be able to:
  - Cover all the activities such as production, consumption, heartbeat, etc.
  - Effectively transport data with large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

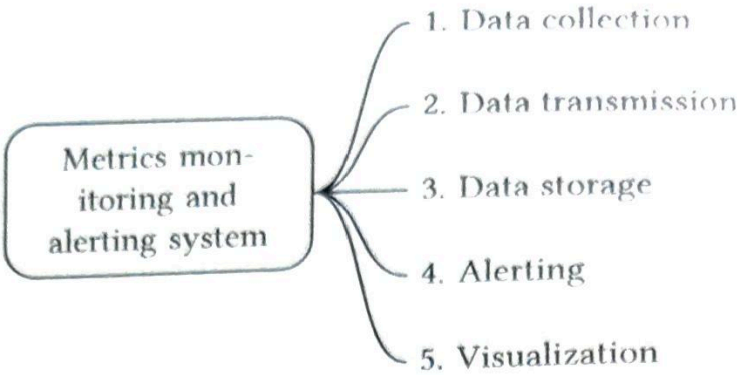


Figure 5.2: Five components of the system

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`?

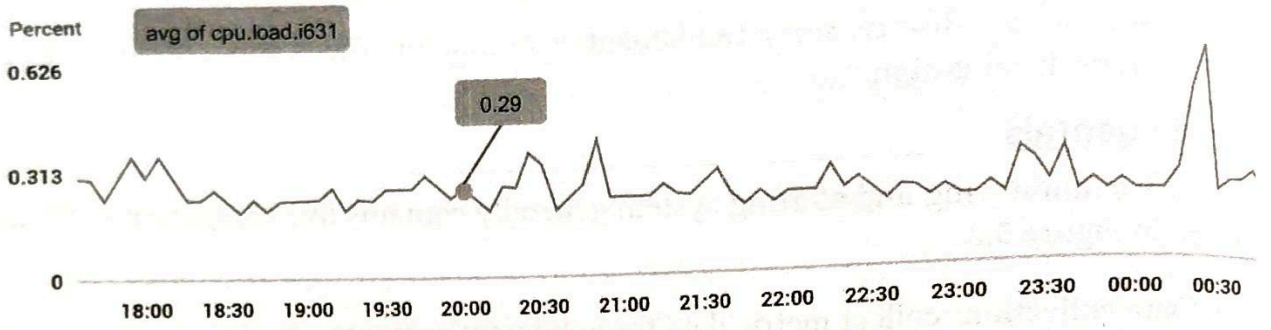


Figure 5.3: Popular metrics monitoring and alerting services

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.

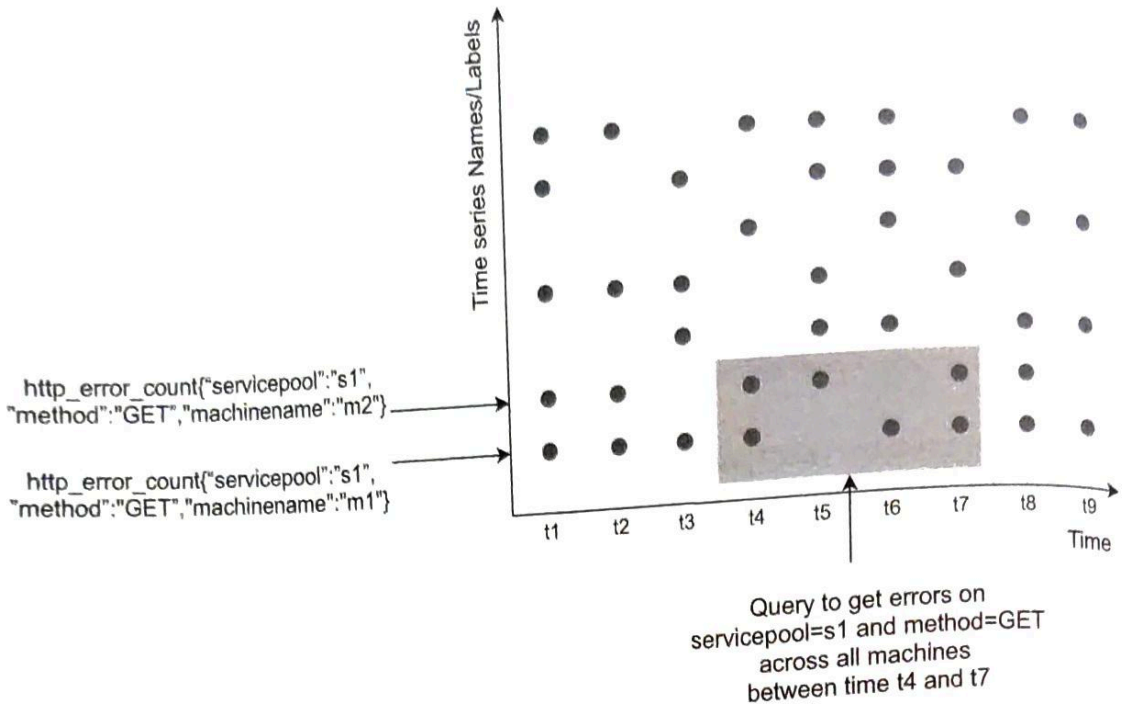


Figure 5.4: Data access pattern

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.

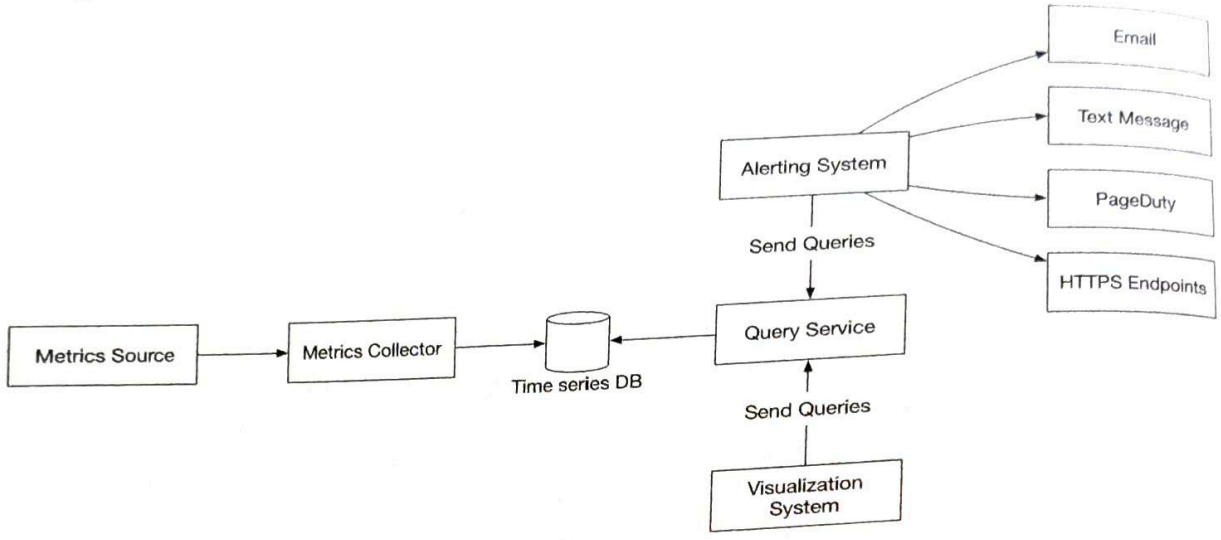


Figure 5.6: High-level design

- **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).

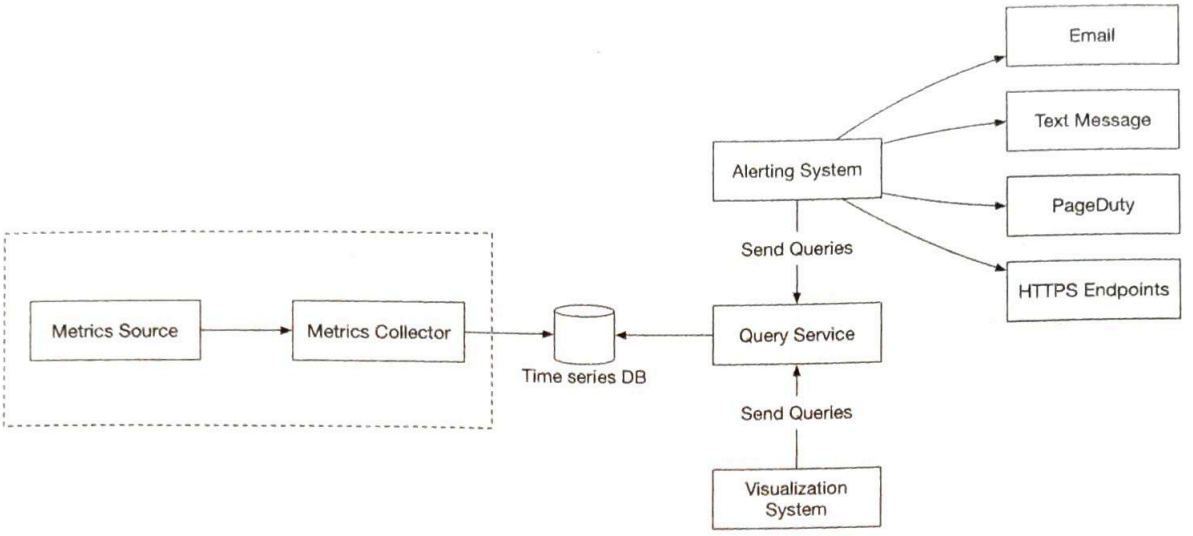


Figure 5.7: Metrics collection flow

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.

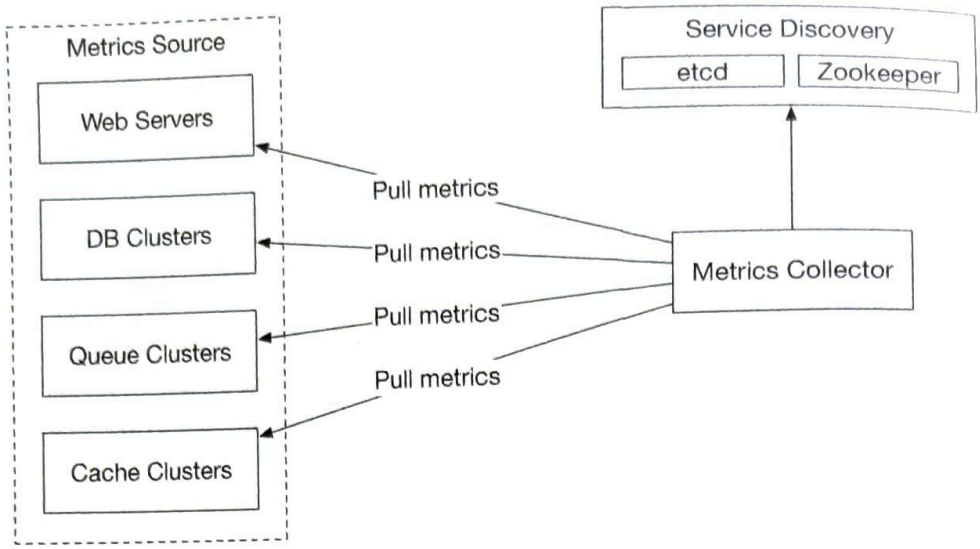


Figure 5.8: Pull model

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.

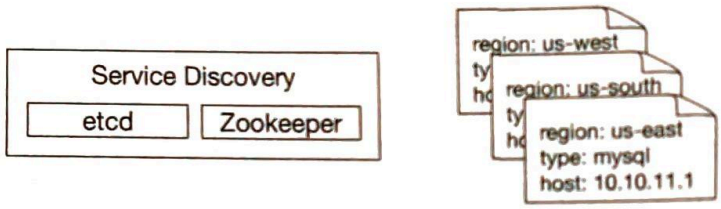


Figure 5.9: Service discovery

Figure 5.10 explains the pull model in detail.

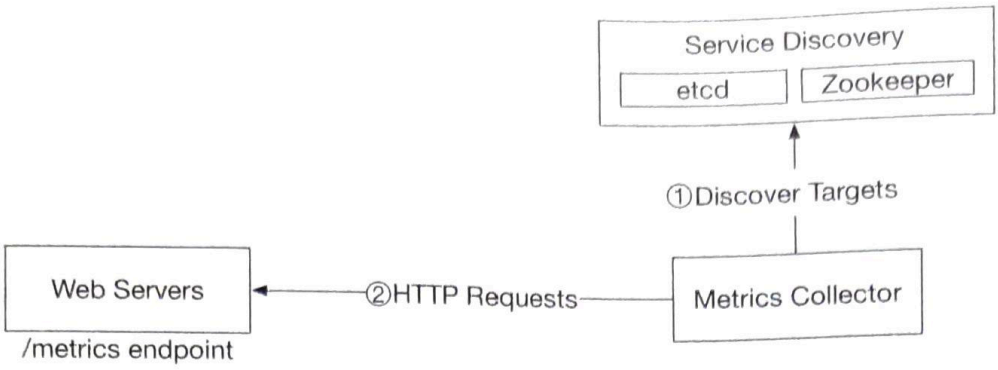


Figure 5.10: Pull model in detail

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.

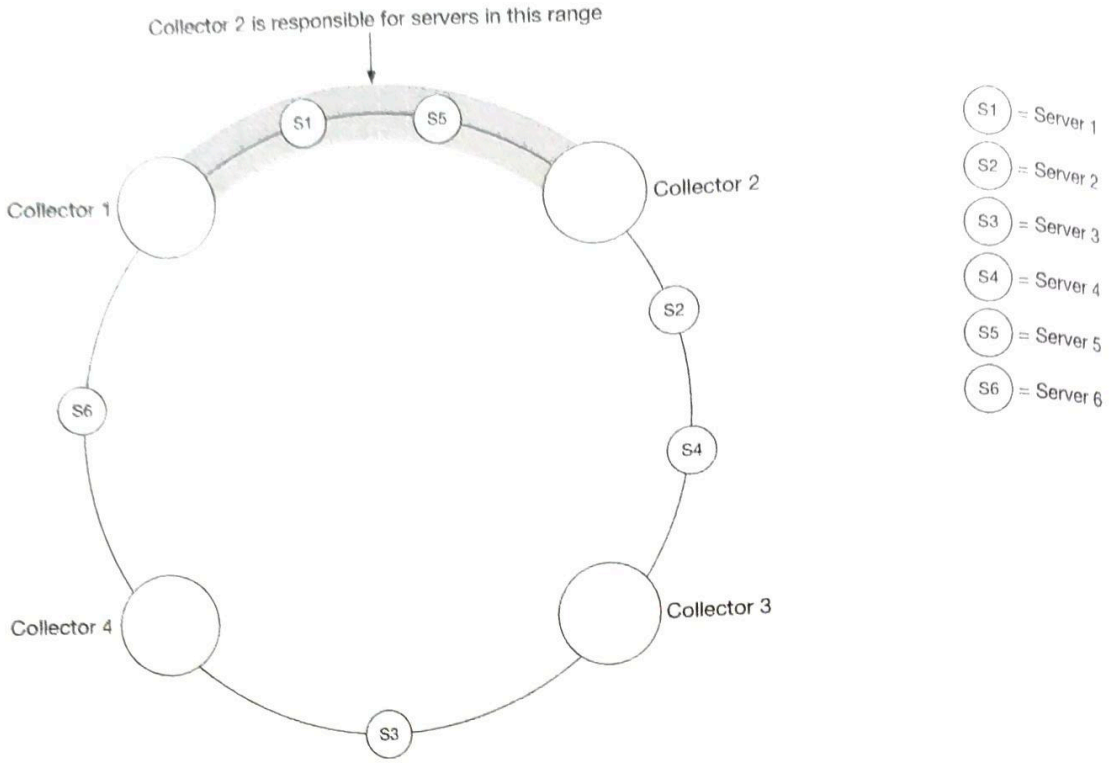


Figure 5.11: Consistent hashing

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.

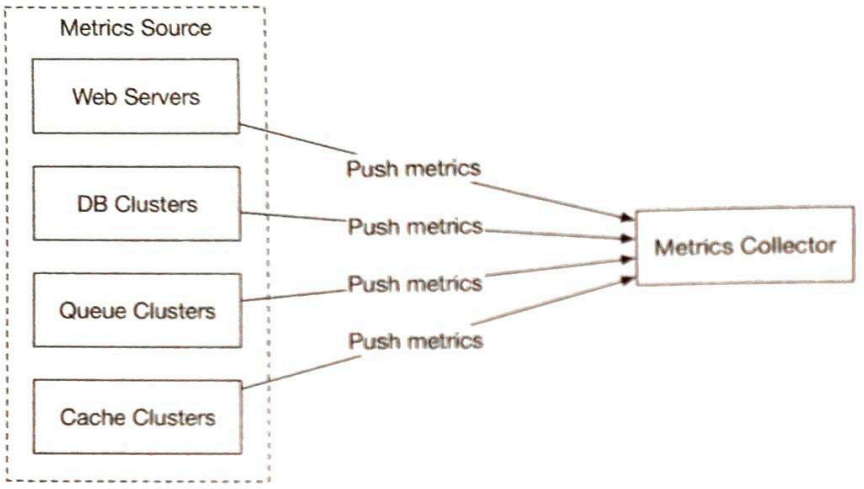


Figure 5.12: Push model

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.

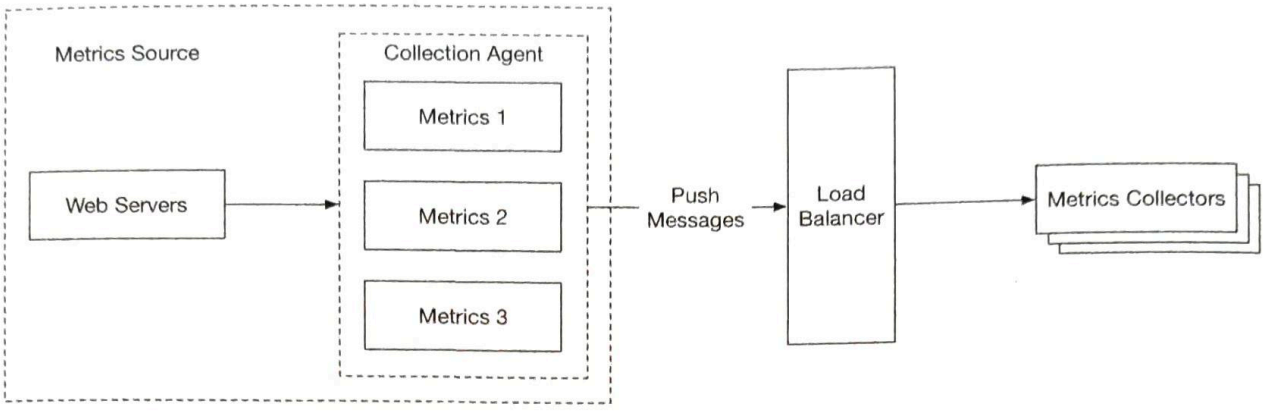


Figure 5.13: Load balancer

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

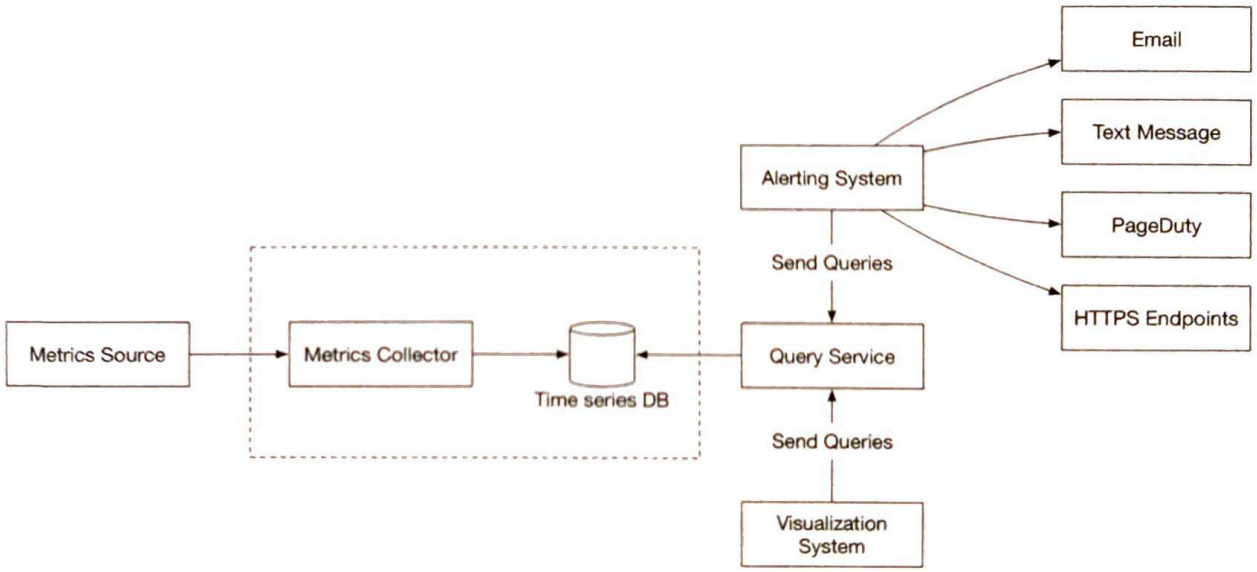


Figure 5.14: Metrics transmission pipeline

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.

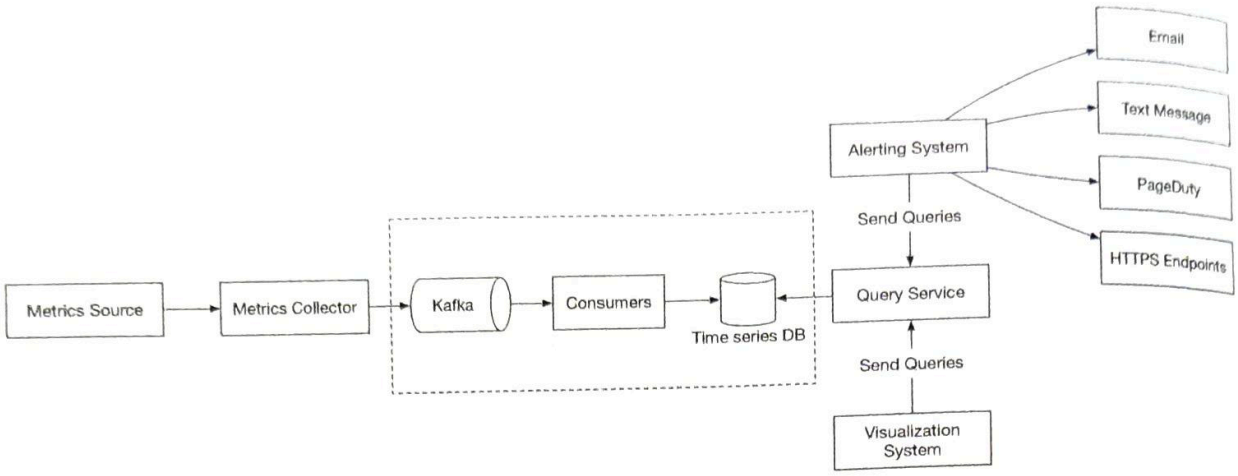


Figure 5.15: Add queues

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.

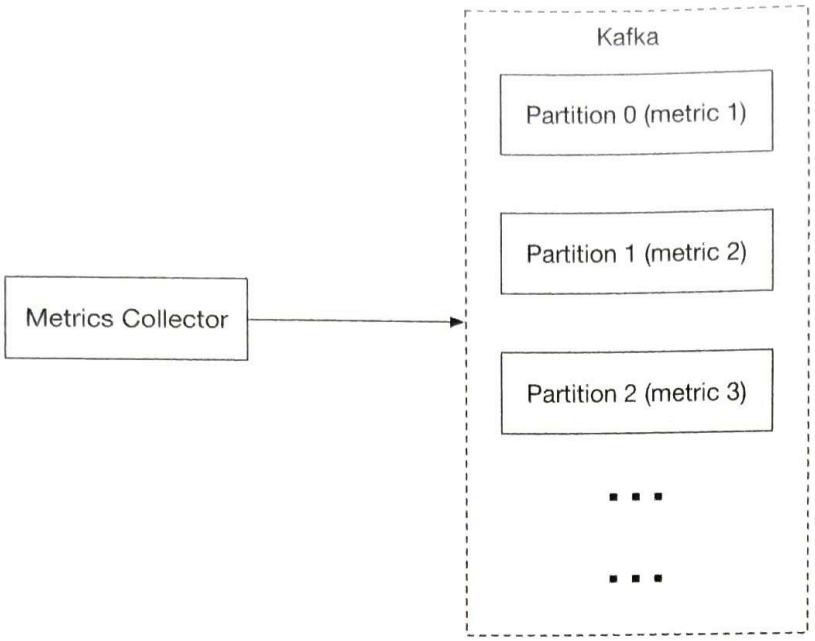


Figure 5.16: Kafka partition

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.

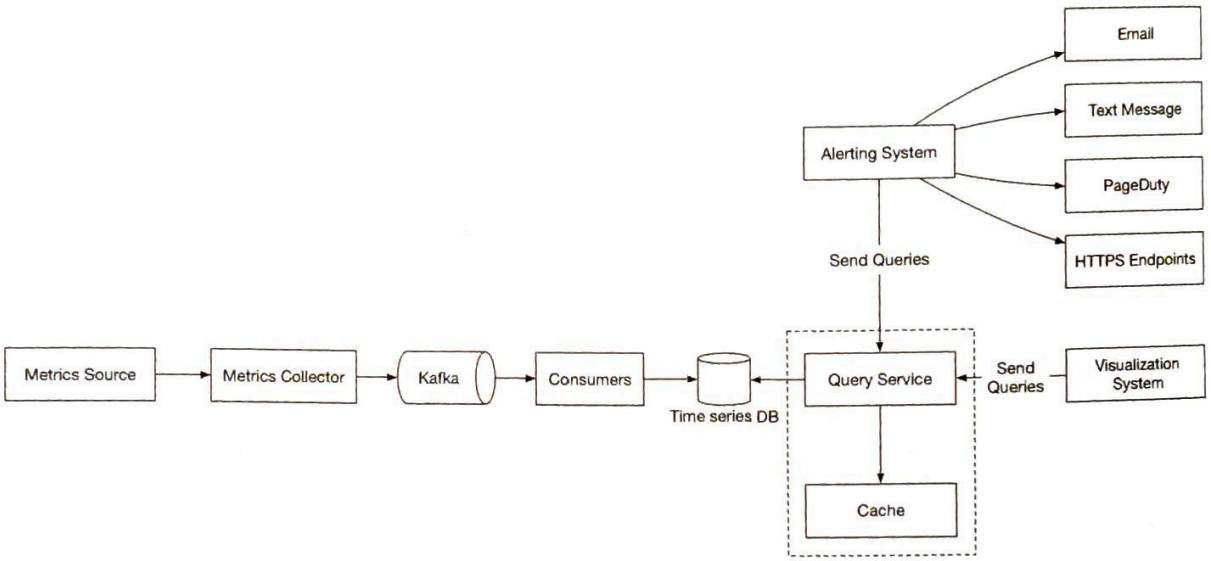


Figure 5.17: Cache layer

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.

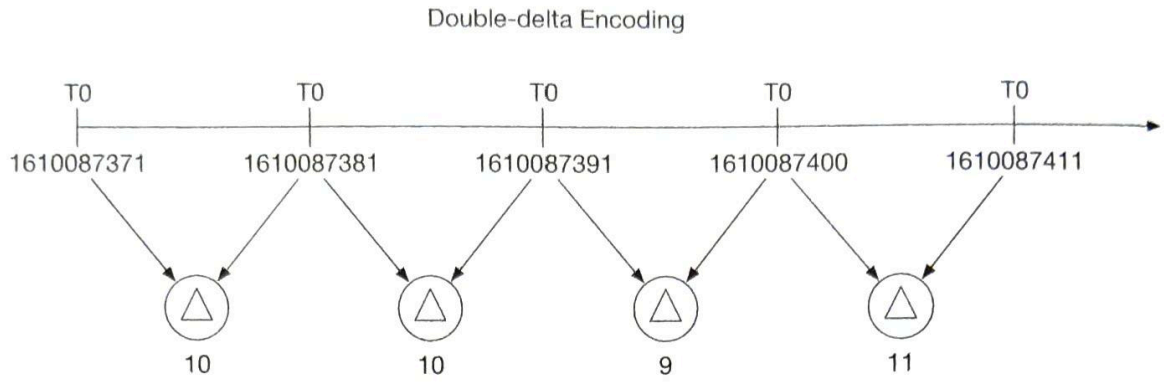


Figure 5.18: Data encoding

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.

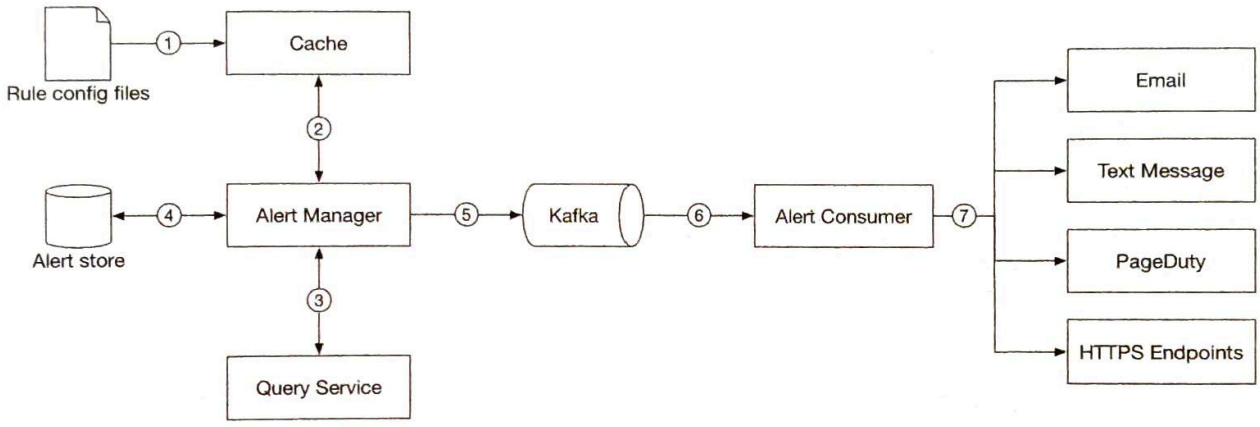


Figure 5.19: Alerting system

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).

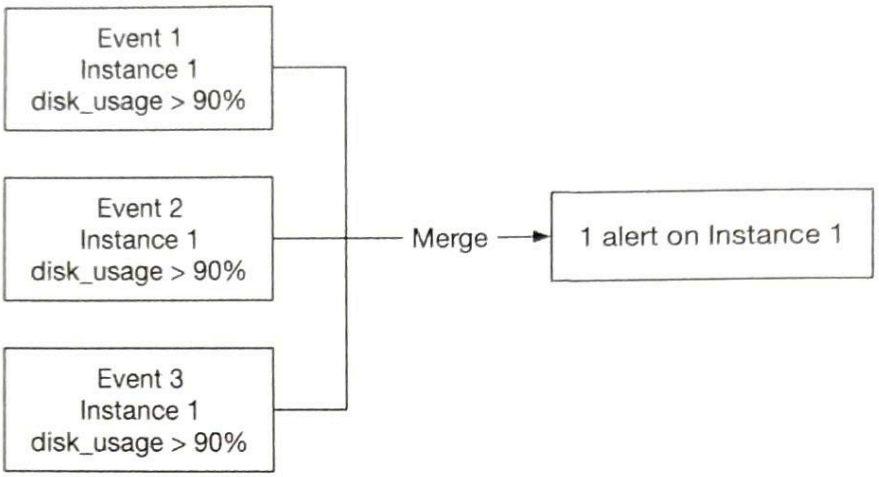


Figure 5.20: Merge alerts

   - 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].

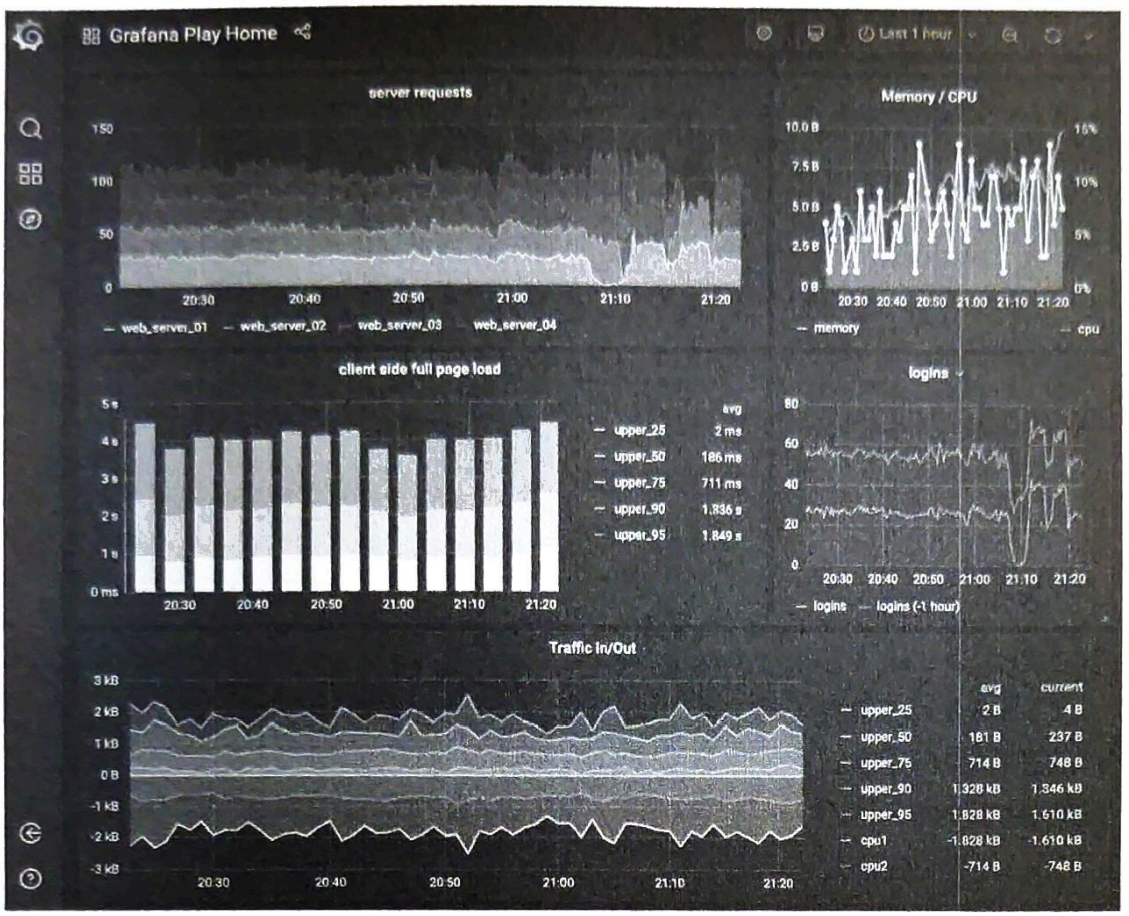


Figure 5.21: Grafana UI

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:

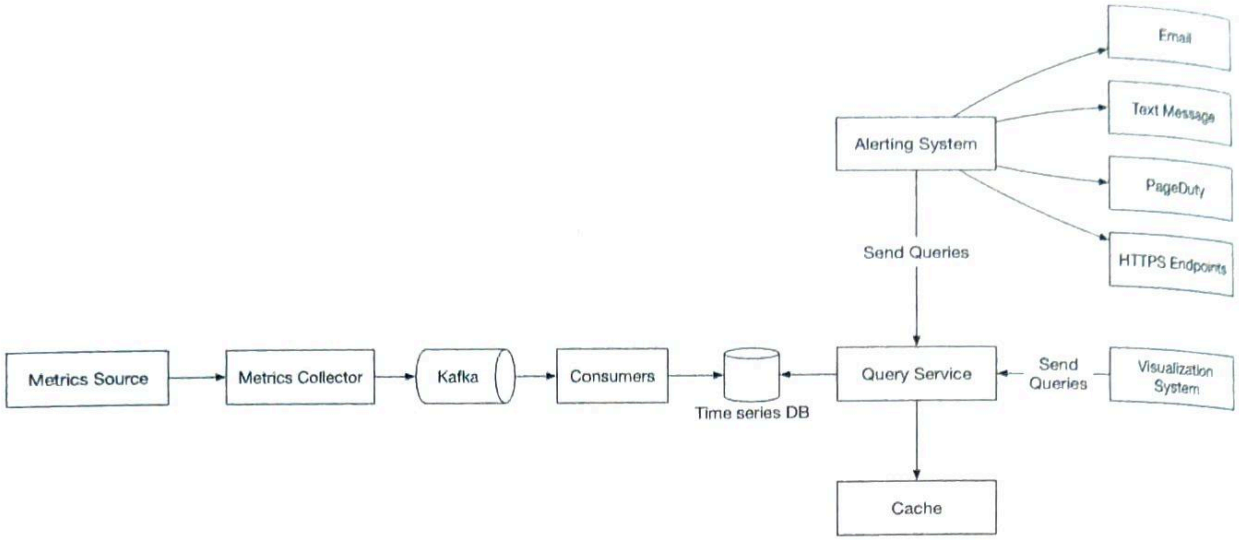
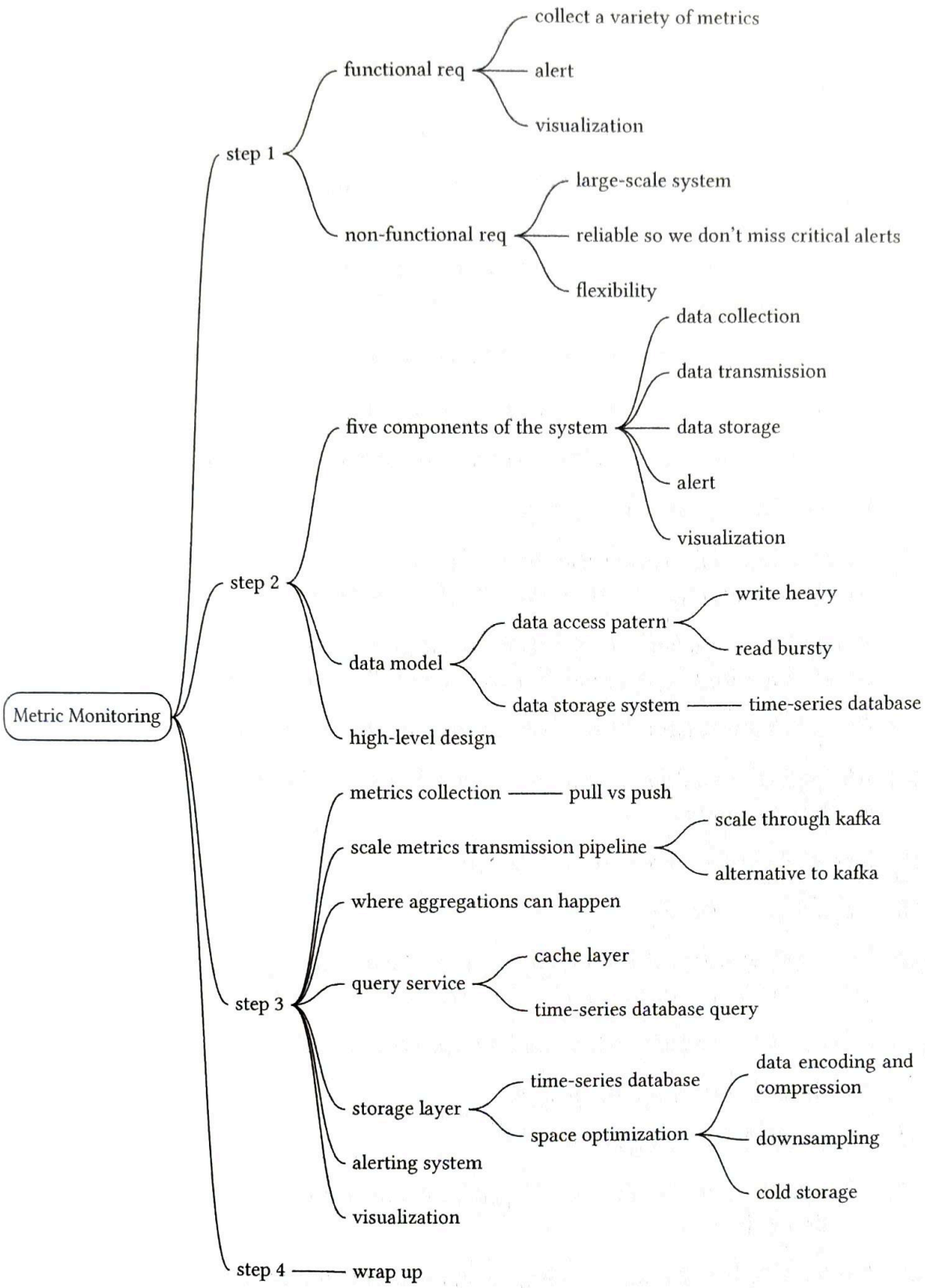


Figure 5.22: Final design

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/.

[24] Pushgateway. https://github.com/prometheus/pushgateway.

[25] Building Applications with Serverless Architectures. https://aws.amazon.com/lambda/serverless-architectures-learn-more/.

[26] Gorilla. AFast,Scalable,In-MemoryTimeSeriesDatabase:http://www.vldb.org/pvldb/vol8/p1816-teller.pdf.

[27] Why We're Building Flux, a New Data Scripting and Query Language. https://www.influxdata.com/blog/why-were-building-flux-a-new-data-scripting-and-query-language/.

[28] InfluxDB storage engine. https://docs.influxdata.com/influxdb/v2.0/reference/internals/storage-engine/.

[29] YAML. https://en.wikipedia.org/wiki/YAML.

[30] Grafana Demo. https://play.grafana.org/.

6 Ad Click Event Aggregation

With the rise of Facebook, YouTube, TikTok, and the online media economy, digital advertising is taking an ever-bigger share of the total advertising spending. As a result, tracking ad click events is very important. In this chapter, we explore how to design an ad click event aggregation system at Facebook or Google scale.

Before we dive into technical design, let's learn about the core concepts of online advertising to better understand this topic. One core benefit of online advertising is its measurability, as quantified by real-time data.

Digital advertising has a core process called Real-Time Bidding (RTB), in which digital advertising inventory is bought and sold. Figure 6.1 shows how the online advertising process works.

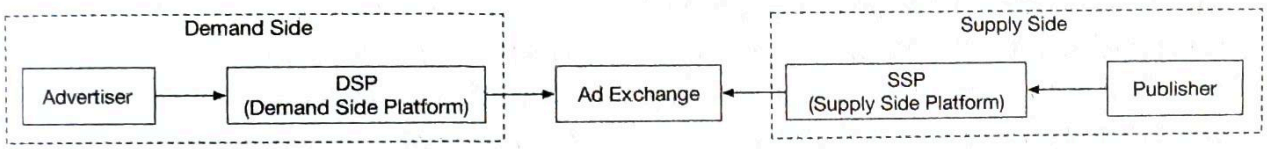


Figure 6.1: RTB process

The speed of the RTB process is important as it usually occurs in less than a second.

Data accuracy is also very important. Ad click event aggregation plays a critical role in measuring the effectiveness of online advertising, which essentially impacts how much money advertisers pay. Based on the click aggregation results, campaign managers can control the budget or adjust bidding strategies, such as changing targeted audience groups, keywords, etc. The key metrics used in online advertising, including click-through rate (CTR) [1] and conversion rate (CVR) [2], depend on aggregated ad click data.

Step 1 - Understand the Problem and Establish Design Scope

The following set of questions helps to clarify requirements and narrow down the scope.

Candidate: What is the format of the input data?
Interviewer: It's a log file located in different servers and the latest click events are appended to the end of the log file. The event has the following attributes: `ad_id`, `click_timestamp`, `user_id`, `ip`, and `country`.

Candidate: What's the data volume?
Interviewer: 1 billion ad clicks per day and 2 million ads in total. The number of ad click events grows 30% year-over-year.

Candidate: What are some of the most important queries to support?
Interviewer: The system needs to support the following 3 queries:

- Return the number of click events for a particular ad in the last M minutes.
- Return the top 100 most clicked ads in the past 1 minute. Both parameters should be configurable. Aggregation occurs every minute.
- Support data filtering by `ip`, `user_id`, or `country` for the above two queries.

Candidate: Do we need to worry about edge cases? I can think of the following:

- There might be events that arrive later than expected.
- There might be duplicated events.
- Different parts of the system might be down at any time, so we need to consider system recovery.

Interviewer: That's a good list. Yes, take these into consideration.

Candidate: What is the latency requirement?
Interviewer: A few minutes of end-to-end latency. Note that latency requirements for RTB and ad click aggregation are very different. While latency for RTB is usually less than one second due to the responsiveness requirement, a few minutes of latency is acceptable for ad click event aggregation because it is primarily used for ad billing and reporting.

With the information gathered above, we have both functional and non-functional requirements.

Functional requirements

- Aggregate the number of clicks of `ad_id` in the last M minutes.
- Return the top 100 most clicked `ad_id` every minute.
- Support aggregation filtering by different attributes.
- Dataset volume is at Facebook or Google scale (see the back-of-envelope estimation section below for detailed system scale requirements).

Non-functional requirements

- Correctness of the aggregation result is important as the data is used for RTB and ads billing.

- Properly handle delayed or duplicate events.
- Robustness. The system should be resilient to partial failures.
- Latency requirement. End-to-end latency should be a few minutes, at most.

Back-of-the-envelope estimation

Let's do an estimation to understand the scale of the system and the potential challenges we will need to address.

- 1 billion DAU (Daily Active Users).
- Assume on average each user clicks 1 ad per day. That's 1 billion ad click events per day.
- Ad click QPS $= \dfrac{10^9 \text{ events}}{10^5 \text{ seconds in a day}} = 10{,}000$
- Assume peak ad click QPS is 5 times the average number. Peak QPS = 50,000 QPS.
- Assume a single ad click event occupies 0.1KB storage. Daily storage requirement is: 0.1KB $\times$ 1 billion = 100GB. The monthly storage requirement is about 3TB.

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

In this section, we discuss query API design, data model, and high-level design.

Query API design

The purpose of the API design is to have an agreement between the client and the server. In a consumer app, a client is usually the end-user who uses the product. In our case, however, a client is the dashboard user (data scientist, product manager, advertiser, etc.) who runs queries against the aggregation service.

Let's review the functional requirements so we can better design the APIs:

- Aggregate the number of clicks of `ad_id` in the last M minutes.
- Return the top N most clicked `ad_ids` in the last M minute.
- Support aggregation filtering by different attributes.

We only need two APIs to support those three use cases because filtering (the last requirement) can be supported by adding query parameters to the requests.

API 1: Aggregate the number of clicks of `ad_id` in the last M minutes.

API	Detail
`GET /v1/ads/{:ad_id}/aggregated_count`	Return aggregated event count for a given `ad_id`

Table 6.1: API for aggregating the number of clicks

Request parameters are:

Field	Description	Type
from	Start minute (default is now minus 1 minute)	long
to	End minute (default is now)	long
filter	An identifier for different filtering strategies. For example, filter = 001 filters out non-US clicks	long

Table 6.2: Request parameters for /v1/ads/{:ad_id}/aggregated_count

Response:

Field	Description	Type
ad_id	The identifier of the ad	string
count	The aggregated count between the start and end minutes	long

Table 6.3: Response for /v1/ads/{:ad_id}/aggregated_count

API 2: Return top N most clicked ad_ids in the last M minutes

API	Detail
GET /v1/ads/popular_ads	Return top N most clicked ads in the last M minutes

Table 6.4: API for /v1/ads/popular_ads

Request parameters are:

Field	Description	Type
count	Top N most clicked ads	integer
window	The aggregation window size (M) in minutes	integer
filter	An identifier for different filtering strategies	long

Table 6.5: Request parameters for /v1/ads/popular_ads

Response:

Field	Description	Type
ad_ids	A list of the most clicked ads	array

Table 6.6: Response for /v1/ads/popular_ads

Data model

There are two types of data in the system: raw data and aggregated data.

Raw data

Below shows what the raw data looks like in log files:

```
[AdClickEvent] ad001, 2021-01-01 00:00:01, user 1, 207.148.22.22, USA
```

Table 6.7 lists what the data fields look like in a structured way. Data is scattered on different application servers.

ad_id	click_timestamp	user_id	ip	country
ad001	2021-01-01 00:00:01	user1	207.148.22.22	USA
ad001	2021-01-01 00:00:02	user1	207.148.22.22	USA
ad002	2021-01-01 00:00:02	user2	209.153.56.11	USA

Table 6.7: Raw data

Aggregated data

Assume that ad click events are aggregated every minute. Table 6.8 shows the aggregated result.

ad_id	click_minute	count
ad001	202101010000	5
ad001	202101010001	7

Table 6.8: Aggregated data

To support ad filtering, we add an additional field called `filter_id` to the table. Records with the same `ad_id` and `click_minute` are grouped by `filter_id` as shown in Table 6.9, and filters are defined in Table 6.10.

ad_id	click_minute	filter_id	count
ad001	202101010000	0012	2
ad001	202101010000	0023	3
ad001	202101010001	0012	1
ad001	202101010001	0023	6

Table 6.9: Aggregated data with filters

filter_id	region	ip	user_id
0012	US	0012	*
0013	*	0023	123.1.2.3

Table 6.10: Filter table

To support the query to return the top N most clicked ads in the last M minutes, the following structure is used.

Most_clicked_ads		
window_size	integer	The aggregation window size (M) in minutes
update_time_minute	timestamp	Last updated timestamp in 1-minute granularity
most_clicked_ads	array	List of ad IDs in JSON format

Table 6.11: Support top N most clicked ads in the last M minutes

Comparison

The comparison between storing raw data and aggregated data is shown below:

	Raw data only	Aggregated data only
Pros	• Full data set • Support data filter and recalculation	• Smaller data set • Fast query
Cons	• Huge data storage • Slow query	• Data loss. This is derived data. For example, 10 entries might be aggregated to 1 entry

Table 6.12: Raw data vs aggregated data

Should we store raw data or aggregated data? Our recommendation is to store both. Let's take a look at why.

- It's a good idea to keep the raw data. If something goes wrong, we could use the raw data for debugging. If the aggregated data is corrupted due to a bad bug, we can recalculate the aggregated data from the raw data, after the bug is fixed.
- Aggregated data should be stored as well. The data size of the raw data is huge. The large size makes querying raw data directly very inefficient. To mitigate this problem, we run read queries on aggregated data.
- Raw data serves as backup data. We usually don't need to query raw data unless recalculation is needed. Old raw data could be moved to cold storage to reduce costs.
- Aggregated data serves as active data. It is tuned for query performance.

Choose the right database

When it comes to choosing the right database, we need to evaluate the following:

- What does the data look like? Is the data relational? Is it a document or a blob?
- Is the workflow read-heavy, write-heavy, or both?
- Is transaction support needed?
- Do the queries rely on many online analytical processing (OLAP) functions [3] like SUM, COUNT?

Let's examine the raw data first. Even though we don't need to query the raw data during normal operations, it is useful for data scientists or machine learning engineers to study user response prediction, behavioral targeting, relevance feedback, etc. [4].

As shown in the back of the envelope estimation, the average write QPS is 10,000, and the peak QPS can be 50,000, so the system is write-heavy. On the read side, raw data is used as backup and a source for recalculation, so in theory, the read volume is low.

Relational databases can do the job, but scaling the write can be challenging. NoSQL databases like Cassandra and InfluxDB are more suitable because they are optimized for write and time-range queries.

Another option is to store the data in Amazon S3 using one of the columnar data formats like ORC [5], Parquet [6], or AVRO [7]. We could put a cap on the size of each file (say, 10GB) and the stream processor responsible for writing the raw data could handle the file rotation when the size cap is reached. Since this setup may be unfamiliar for many, in this design we use Cassandra as an example.

For aggregated data, it is time-series in nature and the workflow is both read and write heavy. This is because, for each ad, we need to query the database every minute to display the latest aggregation count for customers. This feature is useful for auto-refreshing the dashboard or triggering alerts in a timely manner. Since there are two million ads in total, the workflow is read-heavy. Data is aggregated and written every minute by the aggregation service, so it's write-heavy as well. We could use the same type of database to store both raw data and aggregated data.

Now we have discussed query API design and data model, let's put together the high-level design.

High-level design

In real-time big data [8] processing, data usually flows into and out of the processing system as unbounded data streams. The aggregation service works in the same way; the input is the raw data (unbounded data streams), and the output is the aggregated results (see Figure 6.2).

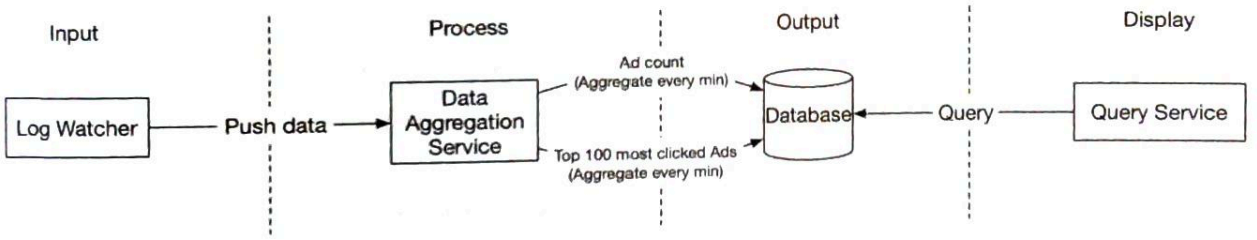


Figure 6.2: Aggregation workflow

Asynchronous processing

The design we currently have is synchronous. This is not good because the capacity of producers and consumers is not always equal. Consider the following case; if there is a sudden increase in traffic and the number of events produced is far beyond what consumers can handle, consumers might get out-of-memory errors or experience an unex-

pected shutdown. If one component in the synchronous link is down, the whole system stops working.

A common solution is to adopt a message queue (Kafka) to decouple producers and consumers. This makes the whole process asynchronous and producers/consumers can be scaled independently.

Putting everything we have discussed together, we come up with the high-level design as shown in Figure 6.3. Log watcher, aggregation service, and database are decoupled by two message queues. The database writer polls data from the message queue, transforms the data into the database format, and writes it to the database.

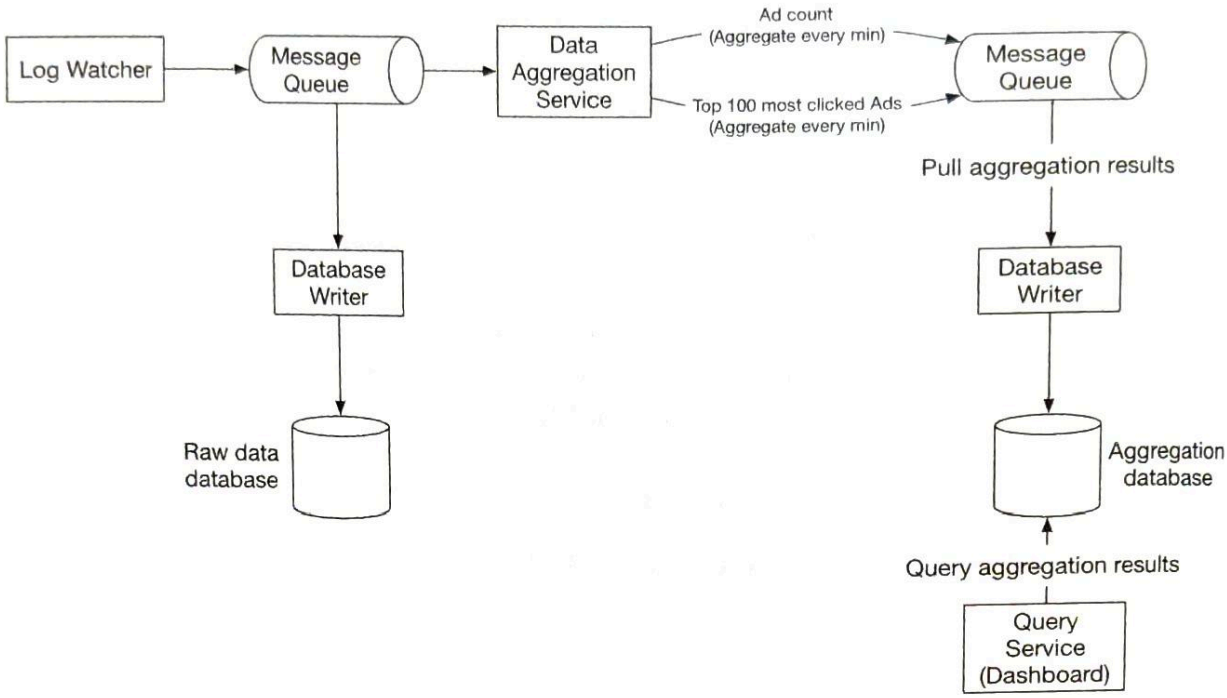


Figure 6.3: High-level design

What is stored in the first message queue? It contains ad click event data as shown in Table 6.13.

ad_id	click_timestamp	user_id	ip	country

Table 6.13: Data in the first message queue

What is stored in the second message queue? The second message queue contains two types of data:

1. Ad click counts aggregated at per-minute granularity.

ad_id	click_minute	count

Table 6.14: Data in the second message queue

2. Top N most clicked ads aggregated at per-minute granularity.

update_time_minute	most_clicked_ads

Table 6.15: Data in the second message queue

You might be wondering why we don't write the aggregated results to the database directly. The short answer is that we need the second message queue like Kafka to achieve end-to-end exactly once semantics (atomic commit) [9].

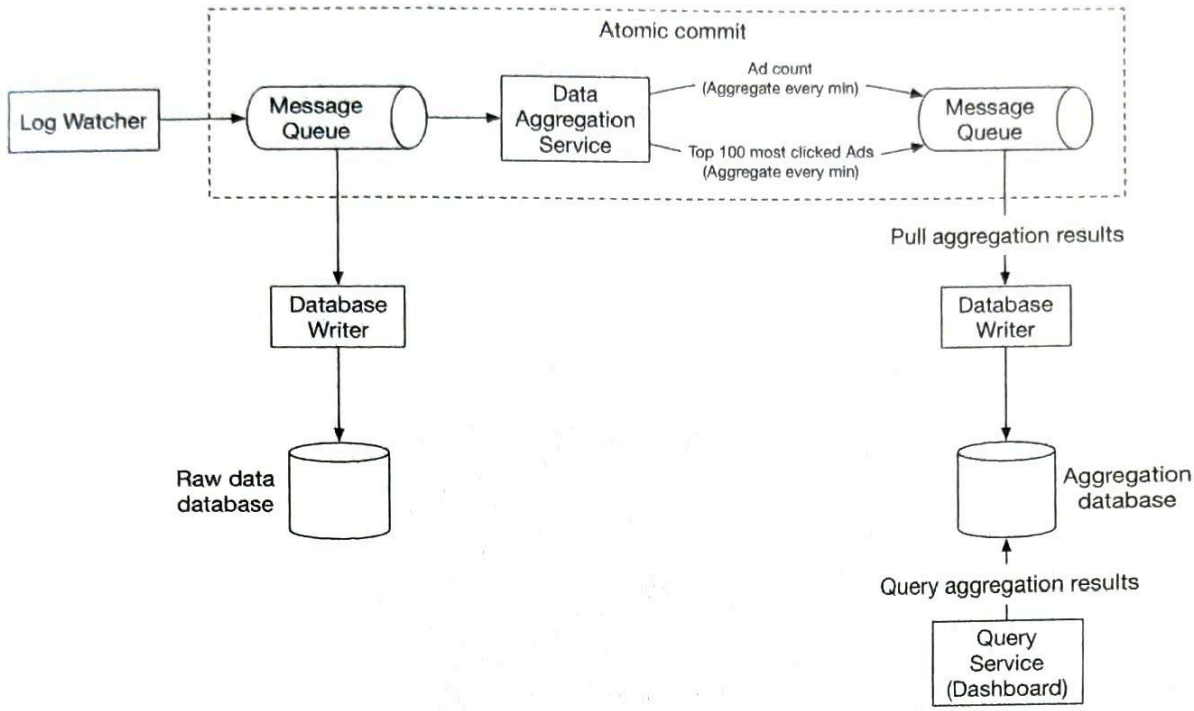


Figure 6.4: End-to-end exactly once

Next, let's dig into the details of the aggregation service.

Aggregation service

The MapReduce framework is a good option to aggregate ad click events. The directed acyclic graph (DAG) is a good model for it [10]. The key to the DAG model is to break down the system into small computing units, like the Map/Aggregate/Reduce nodes, as shown in Figure 6.5.

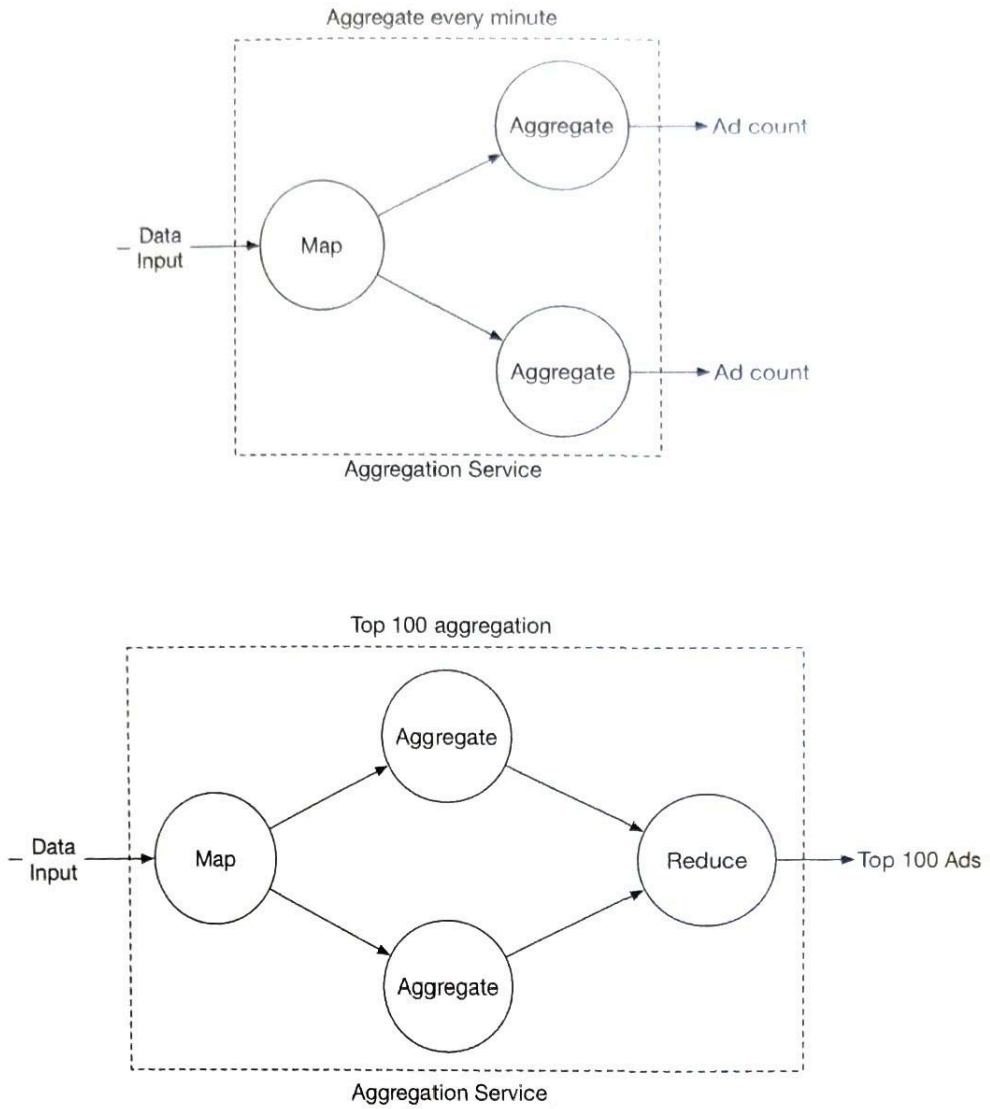


Figure 6.5: Aggregation service

Each node is responsible for one single task and it sends the processing result to its downstream nodes.

Map node

A Map node reads data from a data source, and then filters and transforms the data. For example, a Map node sends ads with `ad_id` %2 = 0 to node 1, and the other ads go to node 2, as shown in Figure 6.6.

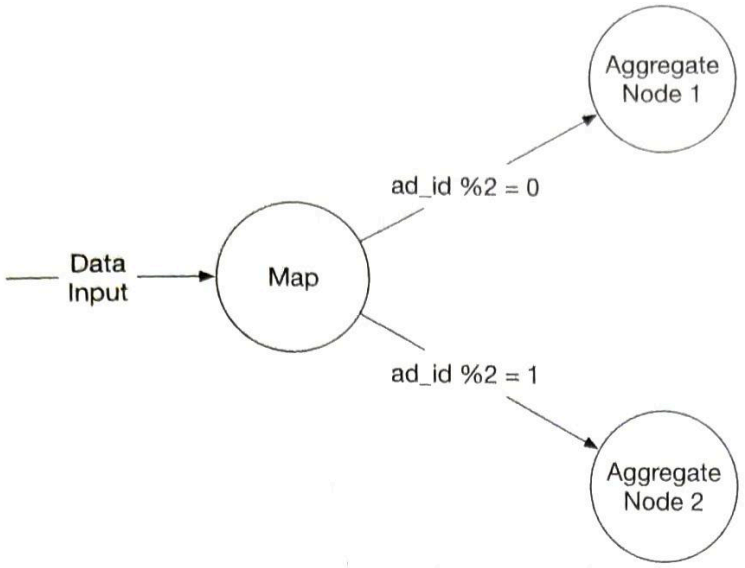


Figure 6.6: Map operation

You might be wondering why we need the Map node. An alternative option is to set up Kafka partitions or tags and let the aggregate nodes subscribe to Kafka directly. This works, but the input data may need to be cleaned or normalized, and these operations can be done by the Map node. Another reason is that we may not have control over how data is produced and therefore events with the same `ad_id` might land in different Kafka partitions.

Aggregate node

An Aggregate node counts ad click events by `ad_id` in memory every minute. In the MapReduce paradigm, the Aggregate node is part of the Reduce. So the map-aggregate-reduce process really means map-reduce-reduce.

Reduce node

A Reduce node reduces aggregated results from all "Aggregate" nodes to the final result. For example, as shown in Figure 6.7, there are three aggregation nodes and each contains the top 3 most clicked ads within the node. The Reduce node reduces the total number of most clicked ads to 3.

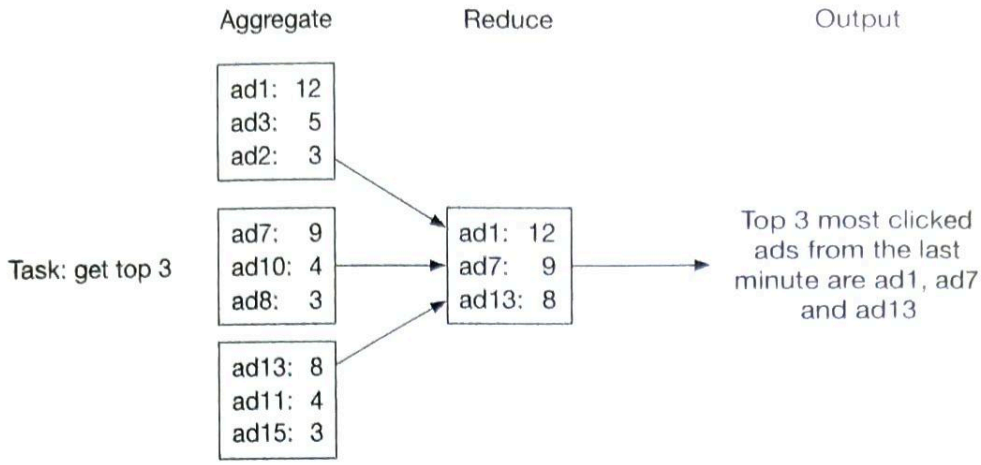


Figure 6.7: Reduce node

The DAG model represents the well-known MapReduce paradigm. It is designed to take big data and use parallel distributed computing to turn big data into little- or regular-sized data.

In the DAG model, intermediate data can be stored in memory and different nodes communicate with each other through either TCP (nodes running in different processes) or shared memory (nodes running in different threads).

Main use cases

Now that we understand how MapReduce works at the high level, let's take a look at how it can be utilized to support the main use cases:

- Aggregate the number of clicks of `ad_id` in the last M mins.
- Return top N most clicked `ad_ids` in the last M minutes.
- Data filtering.

Use case 1: aggregate the number of clicks

As shown in Figure 6.8, input events are partitioned by `ad_id` (`ad_id` %3) in Map nodes and are then aggregated by Aggregation nodes.

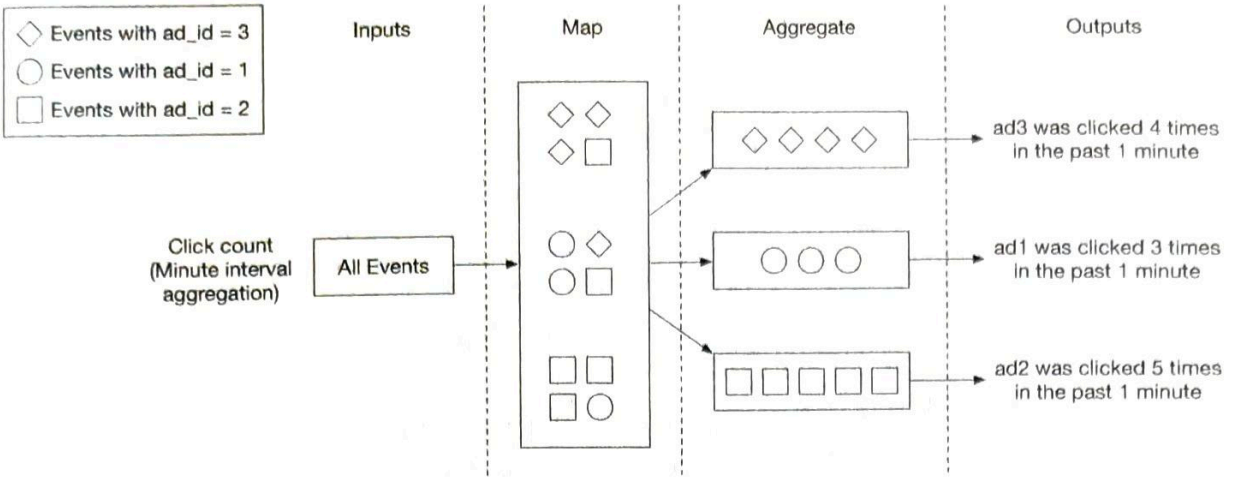


Figure 6.8: Aggregate the number of clicks

Use case 2: return top N most clicked ads

Figure 6.9 shows a simplified design of getting the top 3 most clicked ads, which can be extended to top N. Input events are mapped using ad_id and each Aggregate node maintains a heap data structure to get the top 3 ads within the node efficiently. In the last step, the Reduce node reduces 9 ads (top 3 from each aggregate node) to the top 3 most clicked ads every minute.

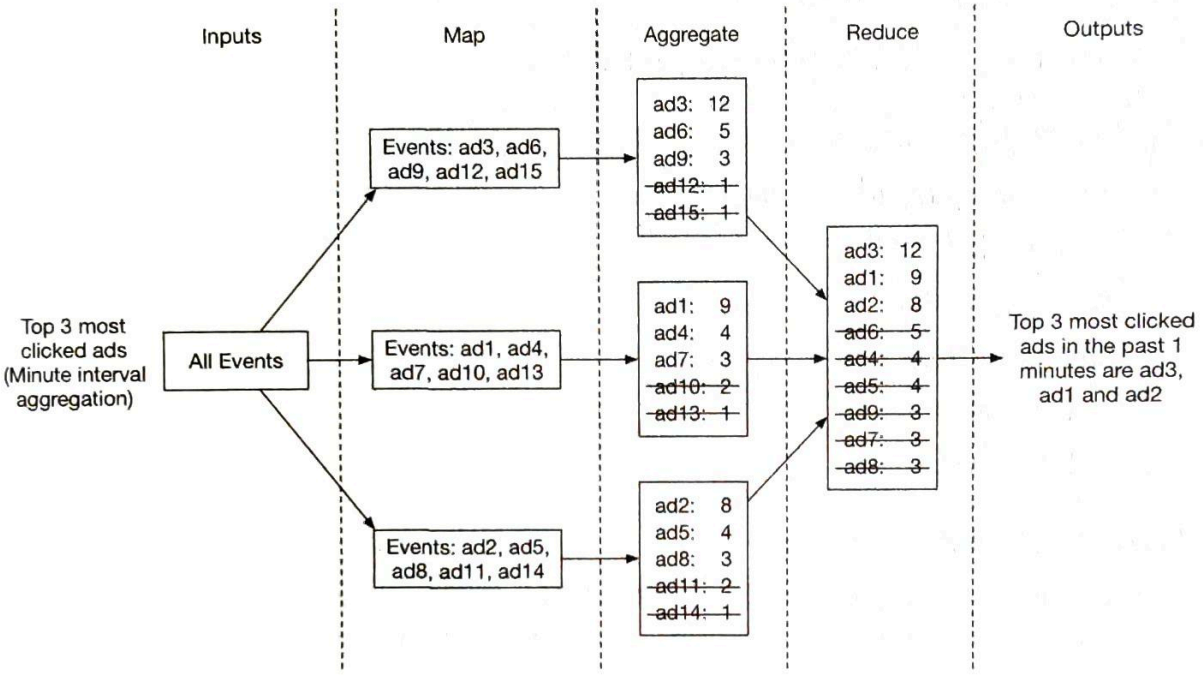


Figure 6.9: Return top N most clicked ads

Use case 3: data filtering

To support data filtering like "show me the aggregated click count for ad001 within the USA only", we can pre-define filtering criteria and aggregate based on them. For example, the aggregation results look like this for `ad001` and `ad002`:

ad_id	click_minute	country	count
ad001	202101010001	USA	100
ad001	202101010001	GPB	200
ad001	202101010001	others	3000
ad002	202101010001	USA	10
ad002	202101010001	GPB	25
ad002	202101010001	others	12

Table 6.16: Aggregation results (filter by country)

This technique is called the star schema [11], which is widely used in data warehouses. The filtering fields are called dimensions. This approach has the following benefits:

- It is simple to understand and build.
- The current aggregation service can be reused to create more dimensions in the star schema. No additional component is needed.
- Accessing data based on filtering criteria is fast because the result is pre-calculated.

A limitation with this approach is that it creates many more buckets and records, especially when we have a lot of filtering criteria.

Step 3 - Design Deep Dive

In this section, we will dive deep into the following:

- Streaming vs batching
- Time and aggregation window
- Delivery guarantees
- Scale the system
- Data monitoring and correctness
- Final design diagram
- Fault tolerance

Streaming vs batching

The high-level architecture we proposed in Figure 6.3 is a type of stream processing system. Table 6.17 shows the comparison of three types of systems [12]:

	Services (Online system)	Batch system (offline system)	Streaming system (near real-time system)
Responsiveness	Respond to the client quickly	No response to the client needed	No response to the client needed
Input	User requests	Bounded input with finite size. A large amount of data	Input has no boundary (infinite streams)
Output	Responses to clients	Materialized views, aggregated metrics, etc.	Materialized views, aggregated metrics, etc.
Performance measurement	Availability, latency	Throughput	Throughput, latency
Example	Online shopping	MapReduce	Flink [13]

Table 6.17: Comparison of three types of systems

In our design, both stream processing and batch processing are used. We utilized stream processing to process data as it arrives and generates aggregated results in a near real-time fashion. We utilized batch processing for historical data backup.

For a system that contains two processing paths (batch and streaming) simultaneously, this architecture is called lambda [14]. A disadvantage of lambda architecture is that you have two processing paths, meaning there are two codebases to maintain. Kappa architecture [15], which combines the batch and streaming in one processing path, solves the problem. The key idea is to handle both real-time data processing and continuous data reprocessing using a single stream processing engine. Figure 6.10 shows a comparison of lambda and kappa architecture.

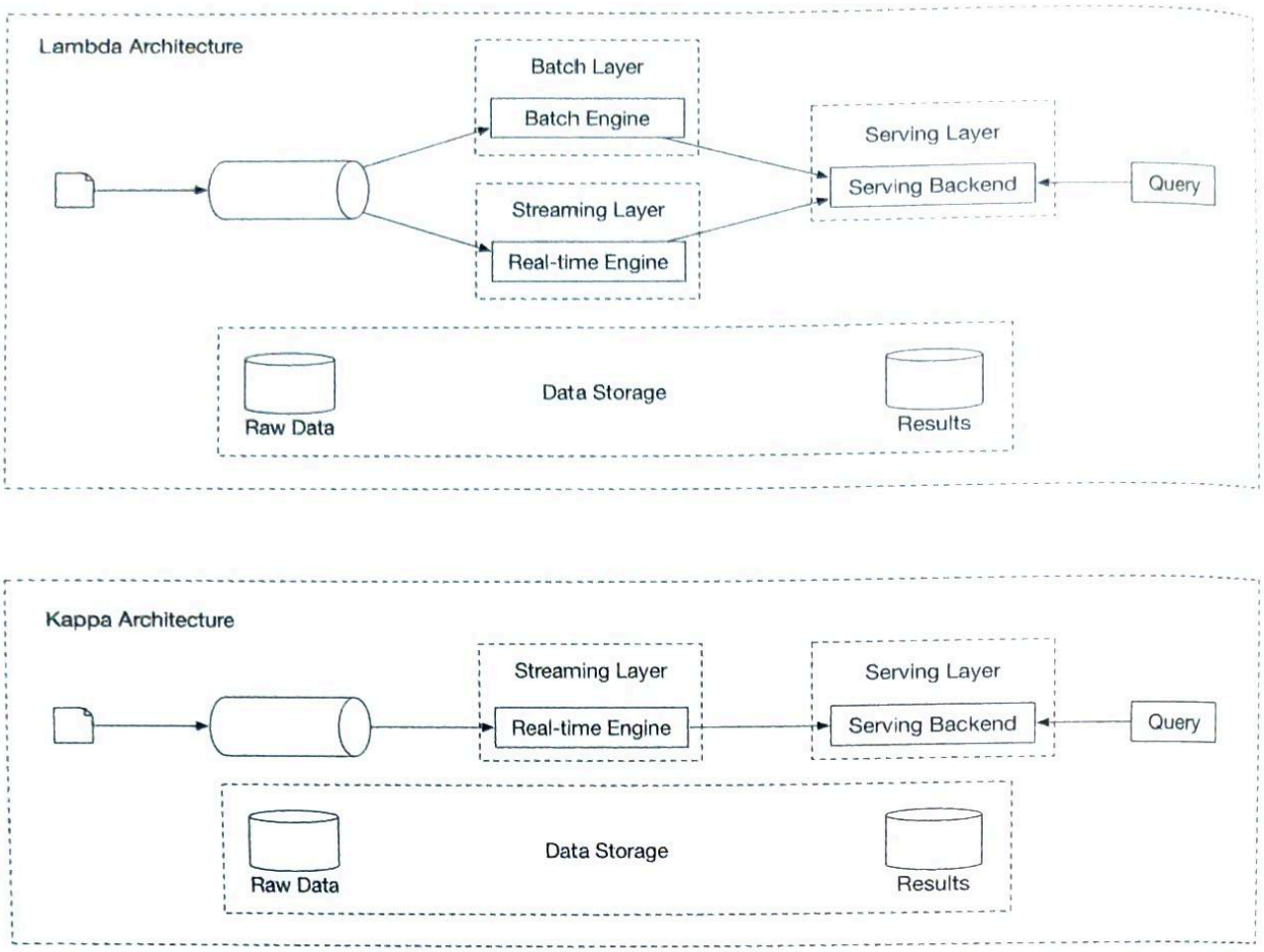


Figure 6.10: Lambda and Kappa architectures

Our high-level design uses Kappa architecture, where the reprocessing of historical data also goes through the real-time aggregation service. See the "Data recalculation" section below for details.

Data recalculation

Sometimes we have to recalculate the aggregated data, also called historical data replay. For example, if we discover a major bug in the aggregation service, we would need to recalculate the aggregated data from raw data starting at the point where the bug was introduced. Figure 6.11 shows the data recalculation flow:

1. The recalculation service retrieves data from raw data storage. This is a batched job.
2. Retrieved data is sent to a dedicated aggregation service so that the real-time processing is not impacted by historical data replay.
3. Aggregated results are sent to the second message queue, then updated in the aggregation database.

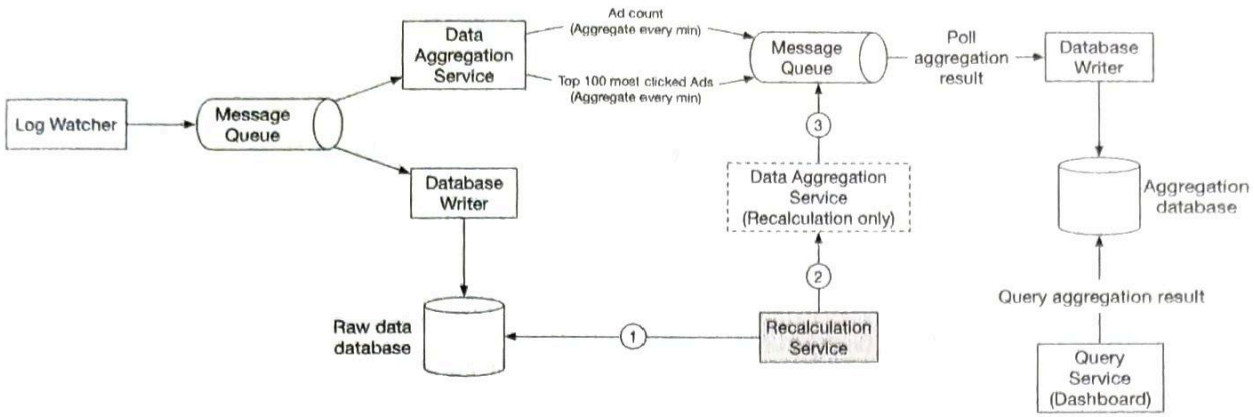


Figure 6.11: Recalculation service

The recalculation process reuses the data aggregation service but uses a different data source (the raw data).

Time

We need a timestamp to perform aggregation. The timestamp can be generated in two different places:

- Event time: when an ad click happens.
- Processing time: refers to the system time of the aggregation server that processes the click event.

Due to network delays and asynchronous environments (data go through a message queue), the gap between event time and processing time can be large. As shown in Figure 6.12, event 1 arrives at the aggregation service very late (5 hours later).

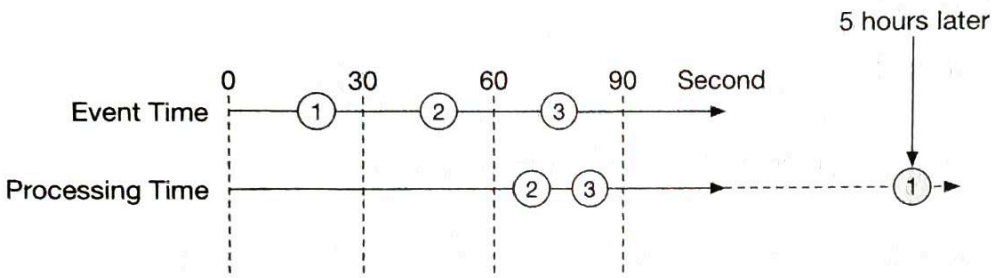


Figure 6.12: Late events

If event time is used for aggregation, we have to deal with delayed events. If processing time is used for aggregation, the aggregation result may not be accurate. There is no perfect solution, so we need to consider the trade-offs.

	Pros	Cons
Event time	Aggregation results are more accurate because the client knows exactly when an ad is clicked	It depends on the timestamp generated on the client-side. Clients might have the wrong time, or the timestamp might be generated by malicious users
Processing time	Server timestamp is more reliable	The timestamp is not accurate if an event reaches the system at a much later time

Table 6.18: Event time vs processing time

Since data accuracy is very important, we recommend using event time for aggregation. How do we properly process delayed events in this case? A technique called "watermark" is commonly utilized to handle slightly delayed events.

In Figure 6.13, ad click events are aggregated in the one-minute tumbling window (see the "Aggregation window" section on page 177 for more details). If event time is used to decide whether the event is in the window, window 1 misses event 2, and window 3 misses event 5 because they arrive slightly later than the end of their aggregation windows.

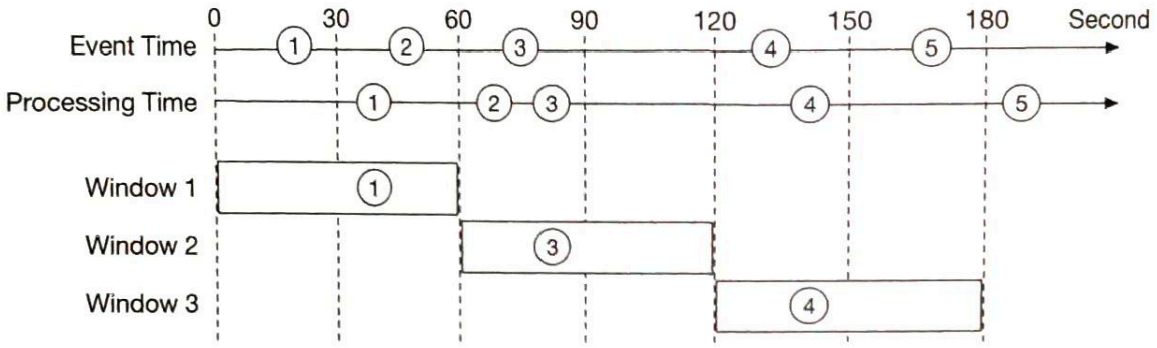


Figure 6.13: Miss events in an aggregation window

One way to mitigate this problem is to use "watermark" (the extended rectangles in Figure 6.14), which is regarded as an extension of an aggregation window. This improves the accuracy of the aggregation result. By extending an extra 15 second (adjustable) aggregation window, window 1 is able to include event 2, and window 3 is able to include event 5.

The value set for the watermark depends on the business requirement. A long watermark could catch events that arrive very late, but it adds more latency to the system. A short watermark means data is less accurate, but it adds less latency to the system.

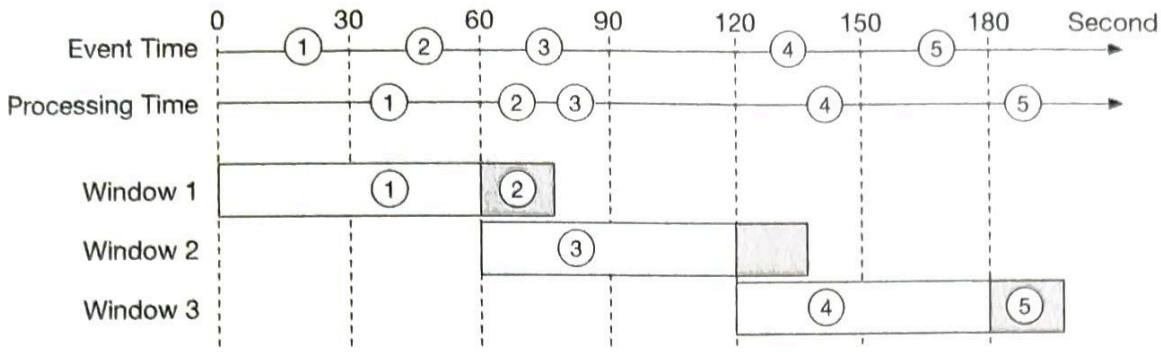


Figure 6.14: Watermark

Notice that the watermark technique does not handle events that have long delays. We can argue that it is not worth the return on investment (ROI) to have a complicated design for low probability events. We can always correct the tiny bit of inaccuracy with end-of-day reconciliation (see "Reconciliation" section on page 189). One trade-off to consider is that using watermark improves data accuracy but increases overall latency, due to extended wait time.

Aggregation window

According to the "Designing data-intensive applications" book by Martin Kleppmann [16], there are four types of window functions: tumbling window (also called fixed window), hopping window, sliding window, and session window. We will discuss the tumbling window and sliding window as they are most relevant to our system.

In the tumbling window (highlighted in Figure 6.15), time is partitioned into same-length, non-overlapping chunks. The tumbling window is a good fit for aggregating ad click events every minute (use case 1).

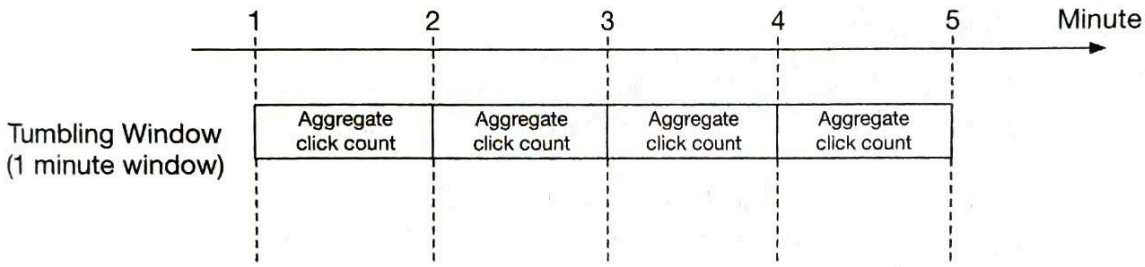


Figure 6.15: Tumbling window

In the sliding window (highlighted in Figure 6.16), events are grouped within a window that slides across the data stream, according to a specified interval. A sliding window can be an overlapping one. This is a good strategy to satisfy our second use case; to get the top N most clicked ads during the last M minutes.

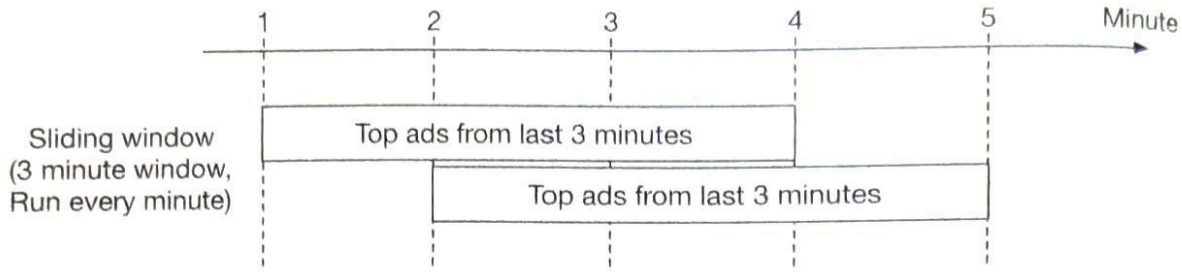


Figure 6.16: Sliding window

Delivery guarantees

Since the aggregation result is utilized for billing, data accuracy and completeness are very important. The system needs to be able to answer questions such as:

- How to avoid processing duplicate events?
- How to ensure all events are processed?

Message queues such as Kafka usually provide three delivery semantics: at-most once, at-least once, and exactly once.

Which delivery method should we choose?

In most circumstances, at-least once processing is good enough if a small percentage of duplicates are acceptable.

However, this is not the case for our system. Differences of a few percent in data points could result in discrepancies of millions of dollars. Therefore, we recommend exactly-once delivery for the system. If you are interested in learning more about a real-life ad aggregation system, take a look at how Yelp implements it [17].

Data deduplication

One of the most common data quality issues is duplicated data. Duplicated data can come from a wide range of sources and in this section, we discuss two common sources.

- Client-side. For example, a client might resend the same event multiple times. Duplicated events sent with malicious intent are best handled by ad fraud/risk control components. If this is of interest, please refer to the reference material [18].
- Server outage. If an aggregation service node goes down in the middle of aggregation and the upstream service hasn't yet received an acknowledgment, the same events might be sent and aggregated again. Let's take a closer look.

Figure 6.17 shows how the aggregation service node (Aggregator) outage introduces duplicate data. The Aggregator manages the status of data consumption by storing the offset in upstream Kafka.

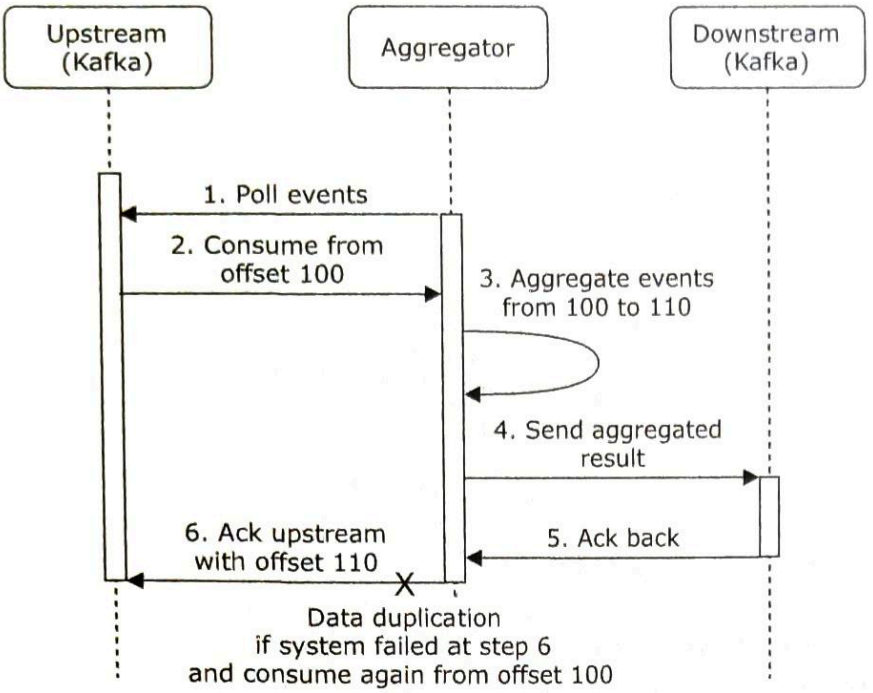


Figure 6.17: Duplicate data

If step 6 fails, perhaps due to Aggregator outage, events from 100 to 110 are already sent to the downstream, but the new offset 110 is not persisted in upstream Kafka. In this case, a new Aggregator would consume again from offset 100, even if those events are already processed, causing duplicate data.

The most straightforward solution (Figure 6.18) is to use external file storage, such as HDFS or S3, to record the offset. However, this solution has issues as well.

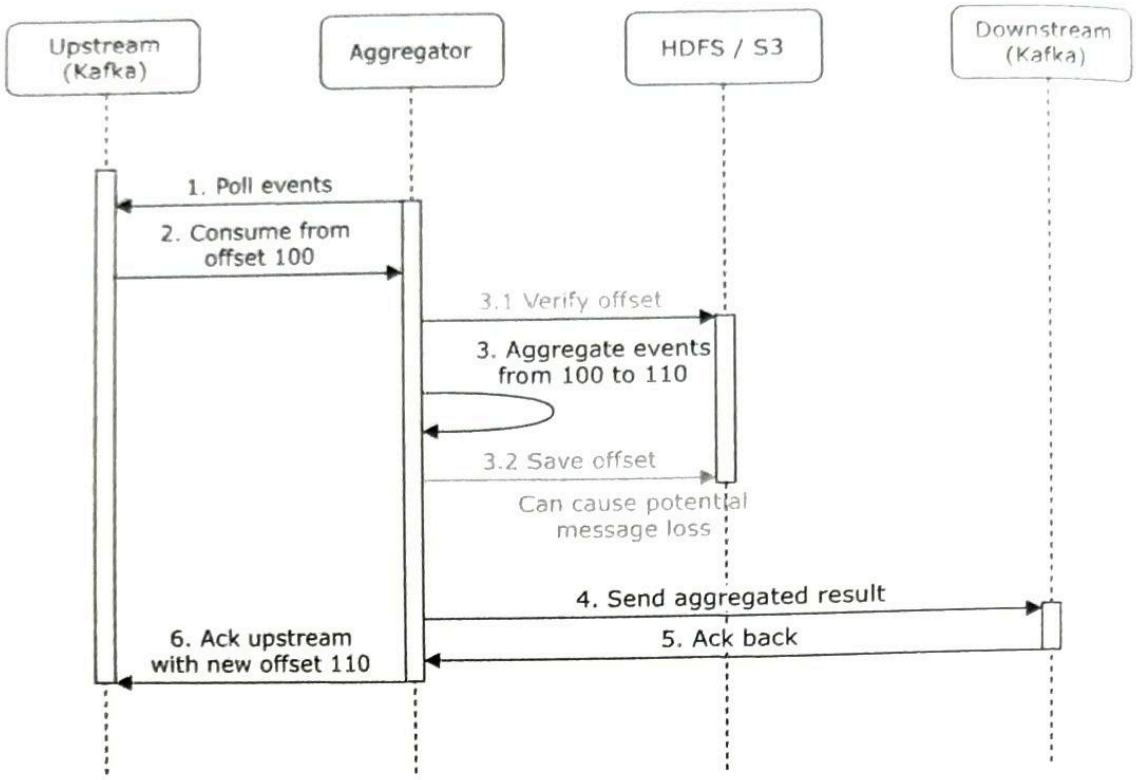


Figure 6.18: Record the offset

In step 3, the aggregator will process events from offset 100 to 110, only if the last offset stored in external storage is 100. If the offset stored in the storage is 110, the aggregator ignores events before offset 110.

But this design has a major problem: the offset is saved to HDFS or S3 (step 3.2) before the aggregation result is sent downstream. If step 4 fails due to Aggregator outage, events from 100 to 110 will never be processed by a newly brought up aggregator node, since the offset stored in external storage is 110.

To avoid data loss, we need to save the offset once we get an acknowledgment back from downstream. The updated design is shown in Figure 6.19.

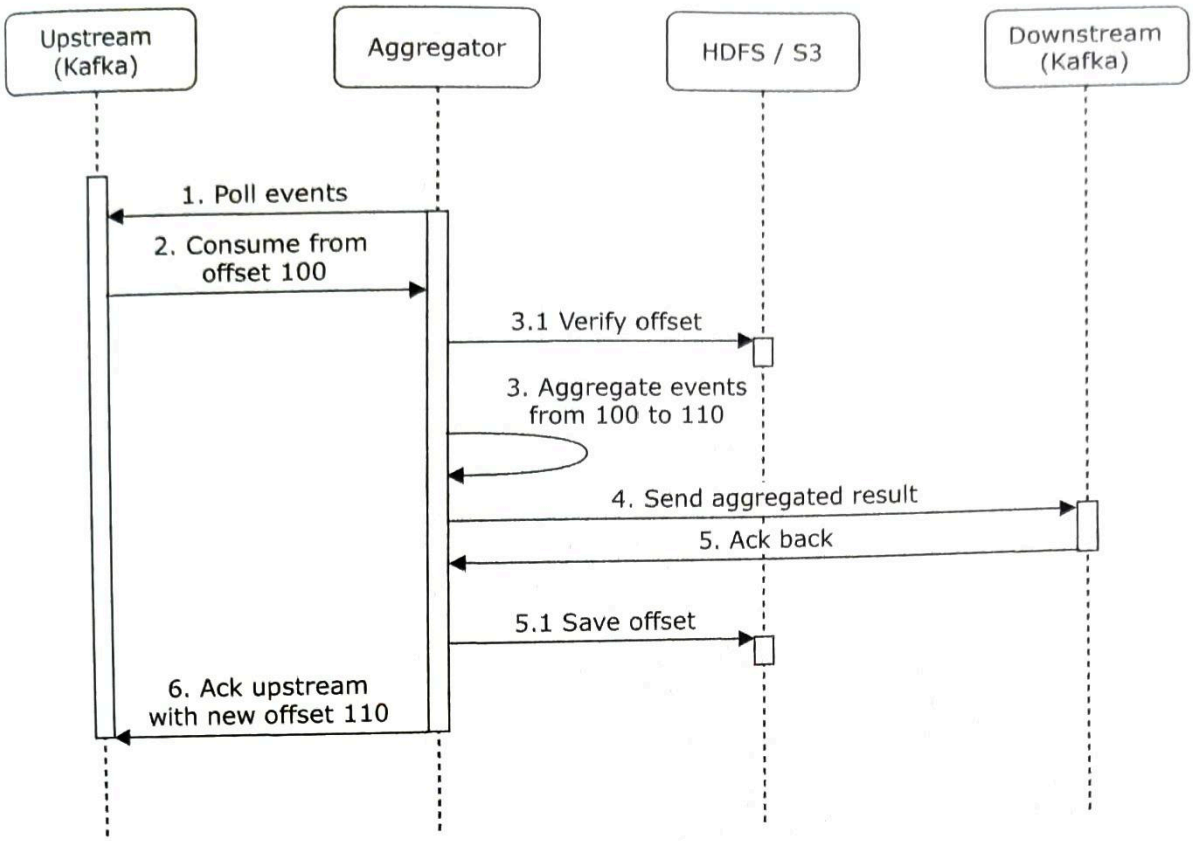


Figure 6.19: Save offset after receiving ack

In this design, if the Aggregator is down before step 5.1 is executed, events from 100 to 110 will be sent downstream again. To achieve exactly once processing, we need to put operations between step 4 to step 6 in one distributed transaction. A distributed transaction is a transaction that works across several nodes. If any of the operations fails, the whole transaction is rolled back.

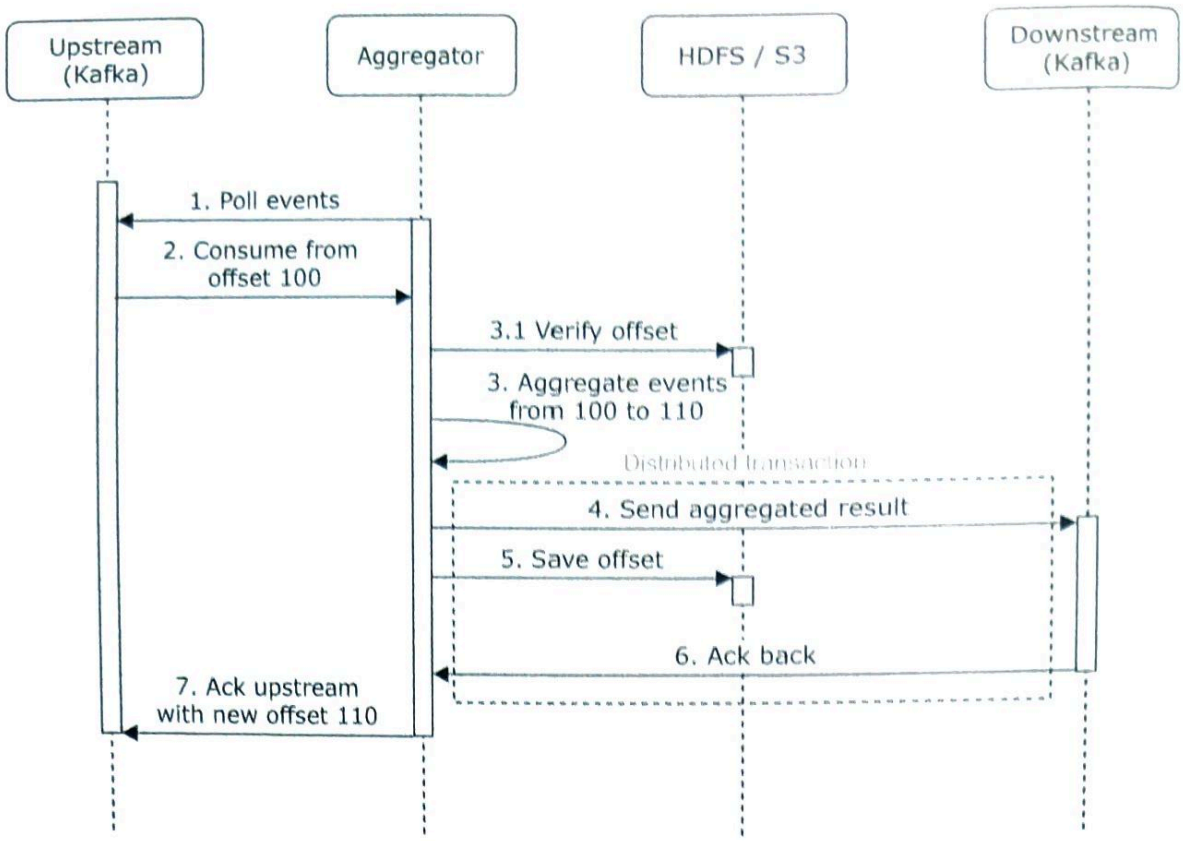


Figure 6.20: Distributed transaction

As you can see, it's not easy to dedupe data in large-scale systems. How to achieve exactly-once processing is an advanced topic. If you are interested in the details, please refer to reference material [9].

Scale the system

From the back-of-the-envelope estimation, we know the business grows 30% per year, which doubles traffic every 3 years. How do we handle this growth? Let's take a look.

Our system consists of three independent components: message queue, aggregation service, and database. Since these components are decoupled, we can scale each one independently.

Scale the message queue

We have already discussed how to scale the message queue extensively in the "Distributed Message Queue" chapter, so we'll only briefly touch on a few points.

Producers. We don't limit the number of producer instances, so the scalability of producers can be easily achieved.

Consumers. Inside a consumer group, the rebalancing mechanism helps to scale the consumers by adding or removing nodes. As shown in Figure 6.21, by adding two more consumers, each consumer only processes events from one partition.

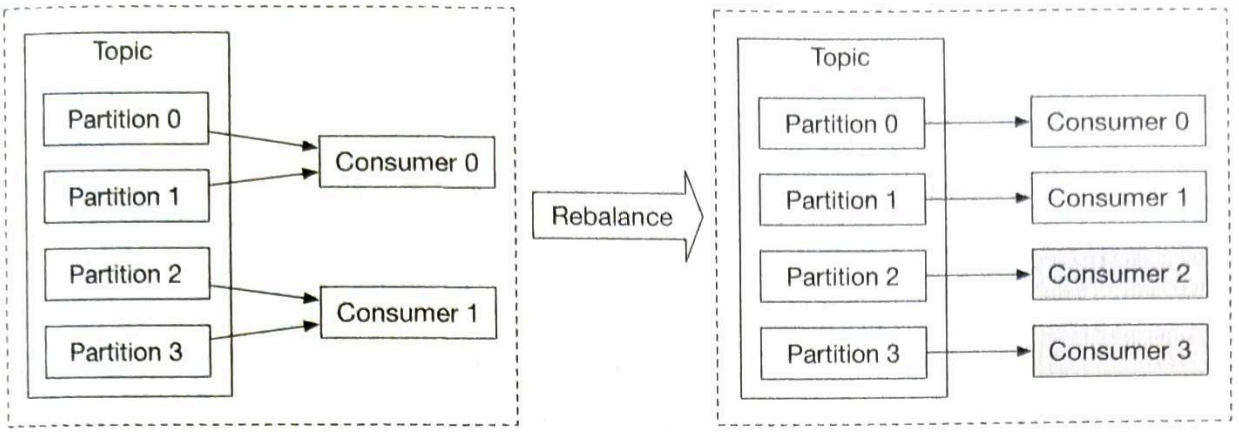


Figure 6.21: Add consumers

When there are hundreds of Kafka consumers in the system, consumer rebalance can be quite slow and could take a few minutes or even more. Therefore, if more consumers need to be added, try to do it during off-peak hours to minimize the impact.

Brokers

- **Hashing key**
  Using `ad_id` as hashing key for Kafka partition to store events from the same `ad_id` in the same Kafka partition. In this case, an aggregation service can subscribe to all events of the same `ad_id` from one single partition.
- **The number of partitions**
  If the number of partitions changes, events of the same `ad_id` might be mapped to a different partition. Therefore, it's recommended to pre-allocate enough partitions in advance, to avoid dynamically increasing the number of partitions in production.
- **Topic physical sharding**
  One single topic is usually not enough. We can split the data by geography (`topic_north_america`, `topic_europe`, `topic_asia`, etc.) or by business type (`topic_web_ads`, `topic_mobile_ads`, etc).
  - Pros: Slicing data to different topics can help increase the system throughput. With fewer consumers for a single topic, the time to rebalance consumer groups is reduced.
  - Cons: It introduces extra complexity and increases maintenance costs.

Scale the aggregation service

In the high-level design, we talked about the aggregation service being a map/reduce operation. Figure 6.22 shows how things are wired together.

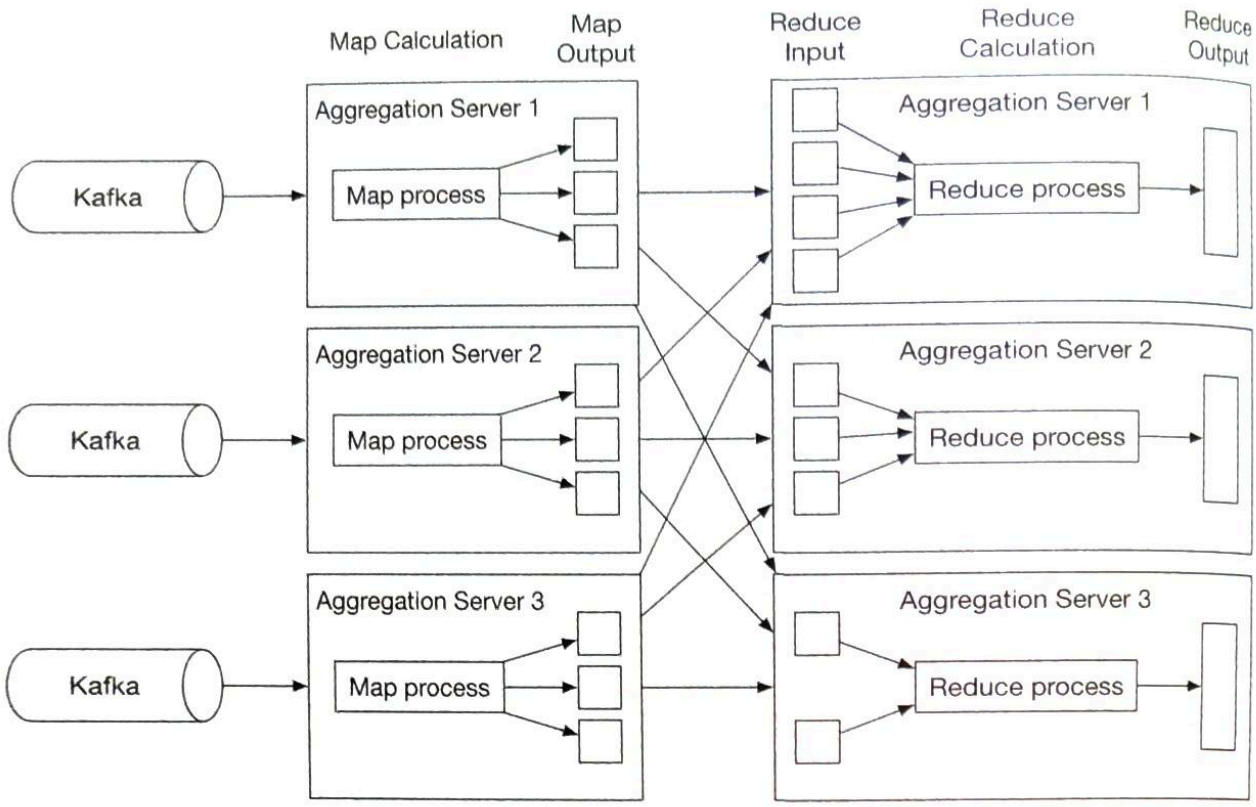


Figure 6.22: Aggregation service

If you are interested in the details, please refer to reference material [19]. Aggregation service is horizontally scalable by adding or removing nodes. Here is an interesting question; how do we increase the throughput of the aggregation service? There are two options.

Option 1: Allocate events with different `ad_ids` to different threads, as shown in Figure 6.23.

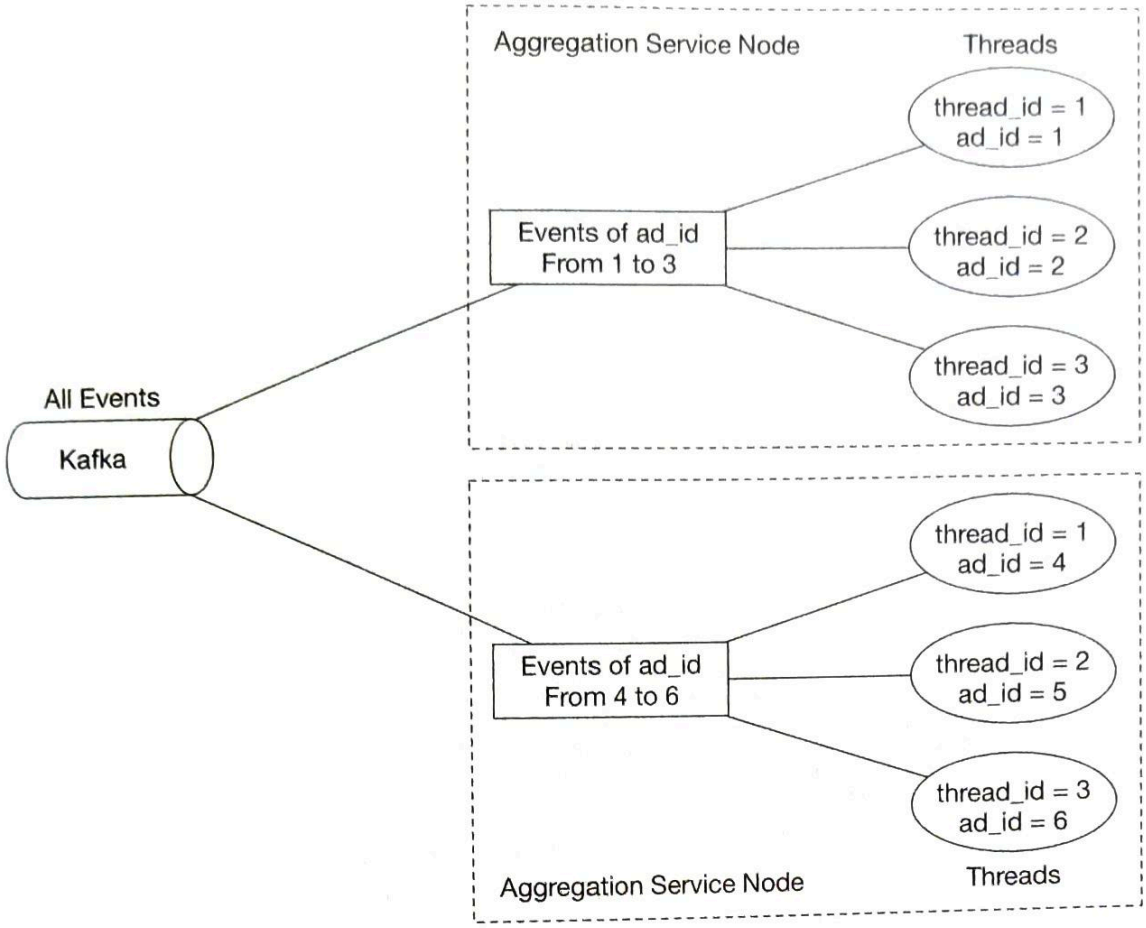


Figure 6.23: Multi-threading

Option 2: Deploy aggregation service nodes on resource providers like Apache Hadoop YARN [20]. You can think of this approach as utilizing multi-processing.

Option 1 is easier to implement and doesn't depend on resource providers. In reality, however, option 2 is more widely used because we can scale the system by adding more computing resources.

Scale the database

Cassandra natively supports horizontal scaling, in a way similar to consistent hashing.

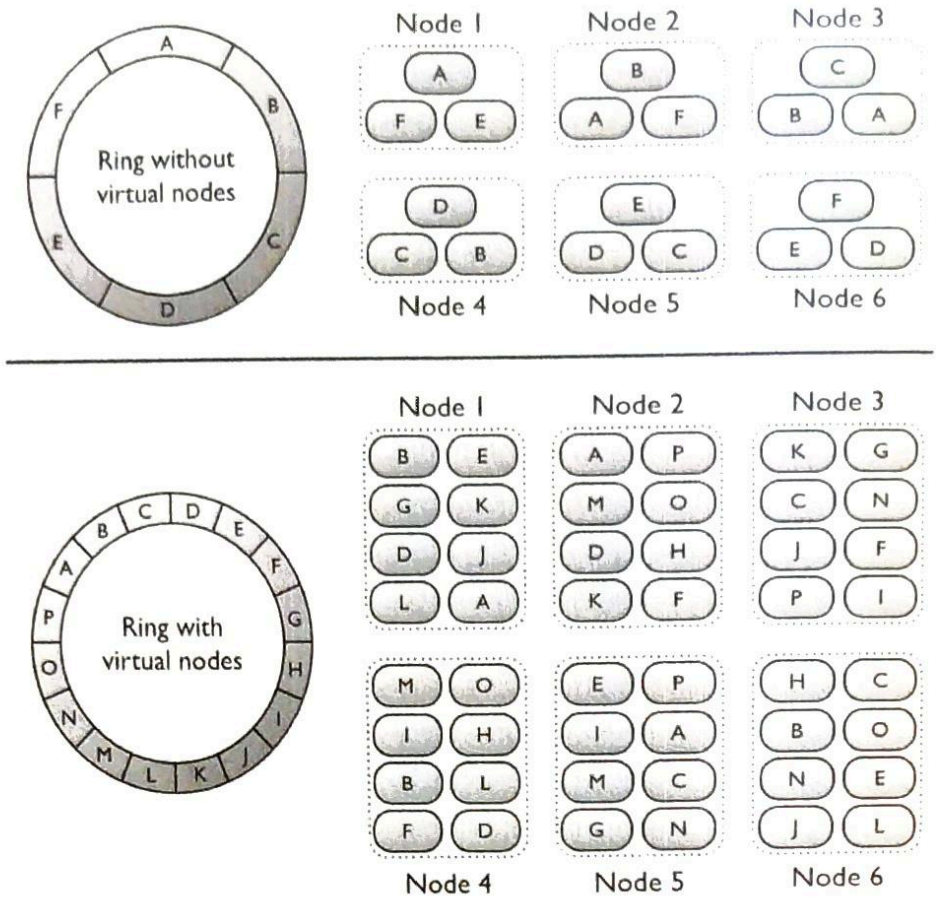


Figure 6.24: Virtual nodes [21]

Data is evenly distributed to every node with a proper replication factor. Each node saves its own part of the ring based on hashed value and also saves copies from other virtual nodes.

If we add a new node to the cluster, it automatically rebalances the virtual nodes among all nodes. No manual resharding is required. See Cassandra's official documentation for more details [21].

Hotspot issue

A shard or service that receives much more data than the others is called a hotspot. This occurs because major companies have advertising budgets in the millions of dollars and their ads are clicked more often. Since events are partitioned by `ad_id`, some aggregation service nodes might receive many more ad click events than others, potentially causing server overload.

This problem can be mitigated by allocating more aggregation nodes to process popular ads. Let's take a look at an example as shown in Figure 6.25. Assume each aggregation node can handle only 100 events.

1. Since there are 300 events in the aggregation node (beyond the capacity of a node can handle), it applies for extra resources through the resource manager.
2. The resource manager allocates more resources (for example, add two more aggregation nodes) so the original aggregation node isn't overloaded.

3. The original aggregation node split events into 3 groups and each aggregation node handles 100 events.
4. The result is written back to the original aggregate node.

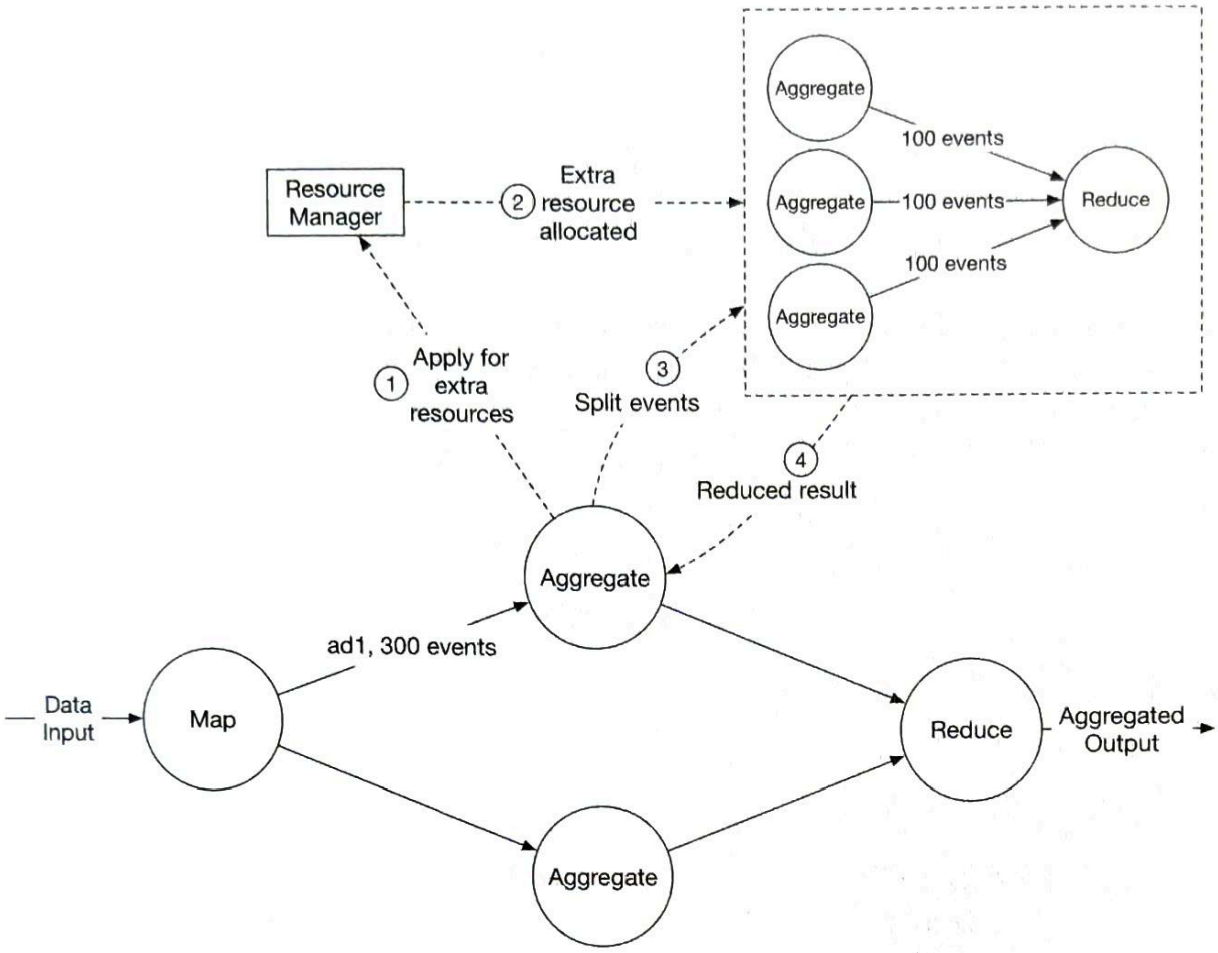


Figure 6.25: Allocate more aggregation nodes

There are more sophisticated ways to handle this problem, such as Global-Local Aggregation or Split Distinct Aggregation. For more information, please refer to [22].

Fault tolerance

Let's discuss the fault tolerance of the aggregation service. Since aggregation happens in memory, when an aggregation node goes down, the aggregated result is lost as well. We can rebuild the count by replaying events from upstream Kafka brokers.

Replaying data from the beginning of Kafka is slow. A good practice is to save the "system status" like upstream offset to a snapshot and recover from the last saved status. In our design, the "system status" is more than just the upstream offset because we need to store data like top N most clicked ads in the past M minutes.

Figure 6.26 shows a simple example of what the data looks like in a snapshot.

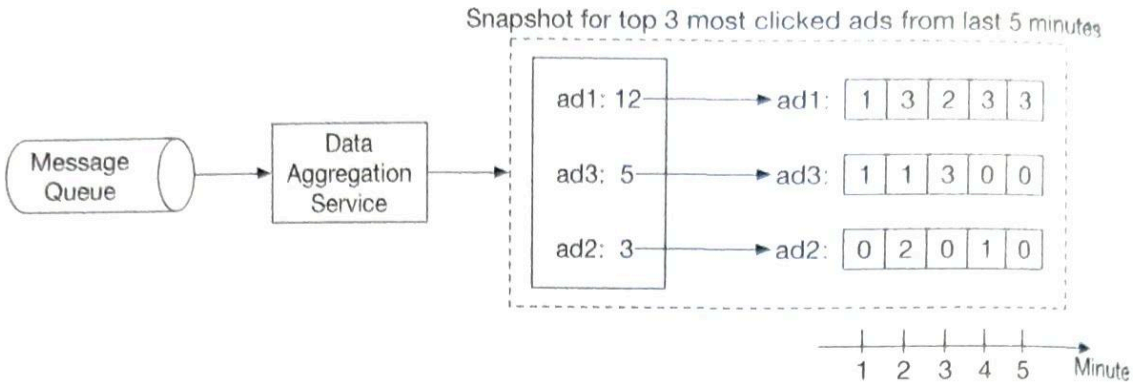


Figure 6.26: Data in a snapshot

With a snapshot, the failover process of the aggregation service is quite simple. If one aggregation service node fails, we bring up a new node and recover data from the latest snapshot (Figure 6.27). If there are new events that arrive after the last snapshot was taken, the new aggregation node will pull those data from the Kafka broker for replay.

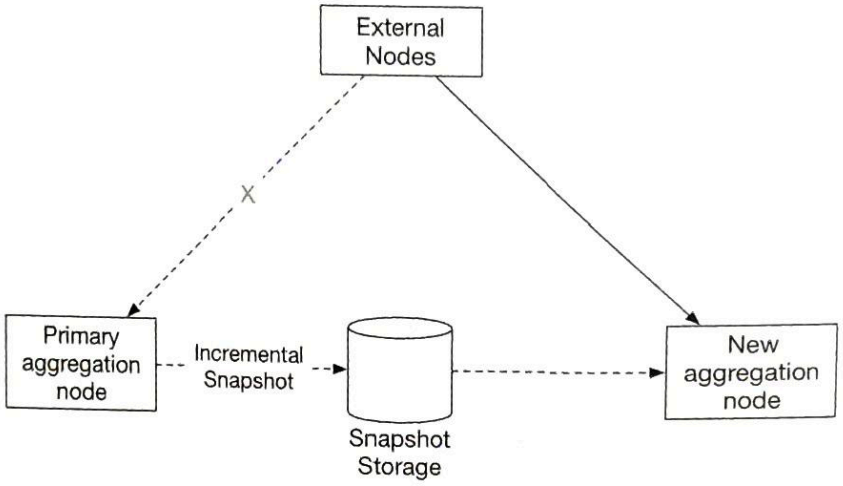


Figure 6.27: Aggregation node failover

Data monitoring and correctness

As mentioned earlier, aggregation results can be used for RTB and billing purposes. It's critical to monitor the system's health and to ensure correctness.

Continuous monitoring

Here are some metrics we might want to monitor:

- Latency. Since latency can be introduced at each stage, it's invaluable to track timestamps as events flow through different parts of the system. The differences between those timestamps can be exposed as latency metrics.
- Message queue size. If there is a sudden increase in queue size, we may need to add more aggregation nodes. Notice that Kafka is a message queue implemented as a distributed commit log, so we need to monitor the records-lag metrics instead.

- System resources on aggregation nodes: CPU, disk, JVM, etc.

Reconciliation

Reconciliation means comparing different sets of data in order to ensure data integrity. Unlike reconciliation in the banking industry, where you can compare your records with the bank's records, the result of ad click aggregation has no third-party result to reconcile with.

What we can do is to sort the ad click events by event time in every partition at the end of the day, by using a batch job and reconciling with the real-time aggregation result. If we have higher accuracy requirements, we can use a smaller aggregation window; for example, one hour. Please note, no matter which aggregation window is used, the result from the batch job might not match exactly with the real-time aggregation result, since some events might arrive late (see "Time" section on page 175).

Figure 6.28 shows the final design diagram with reconciliation support.

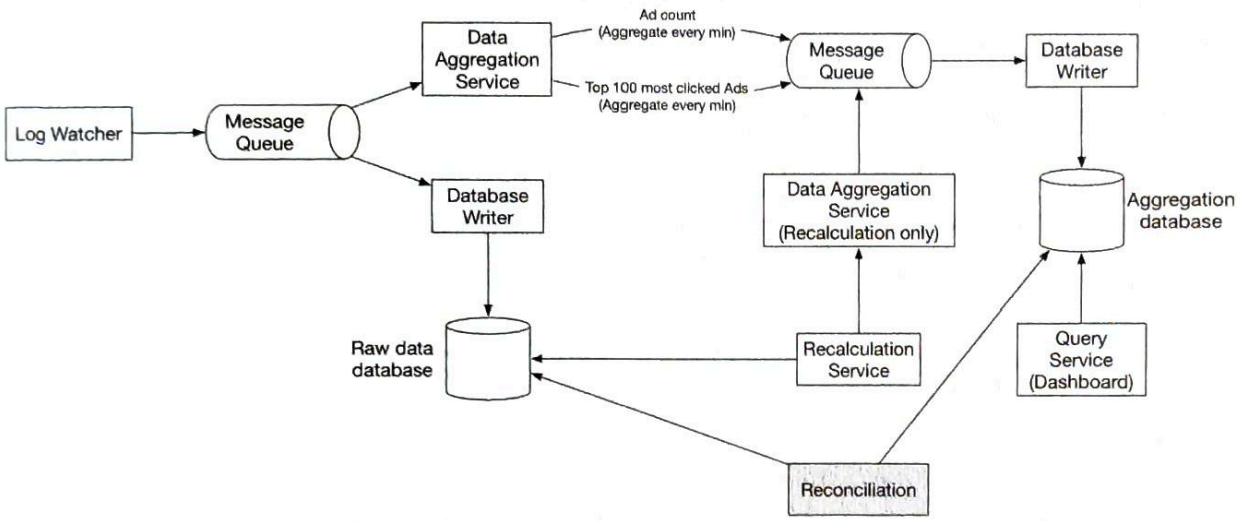


Figure 6.28: Final design

Alternative design

In a generalist system design interview, you are not expected to know the internals of different pieces of specialized software used in a big data pipeline. Explaining your thought process and discussing trade-offs is very important, which is why we propose a generic solution. Another option is to store ad click data in Hive, with an ElasticSearch layer built for faster queries. Aggregation is usually done in OLAP databases such as ClickHouse [23] or Druid [24]. Figure 6.29 shows the architecture.

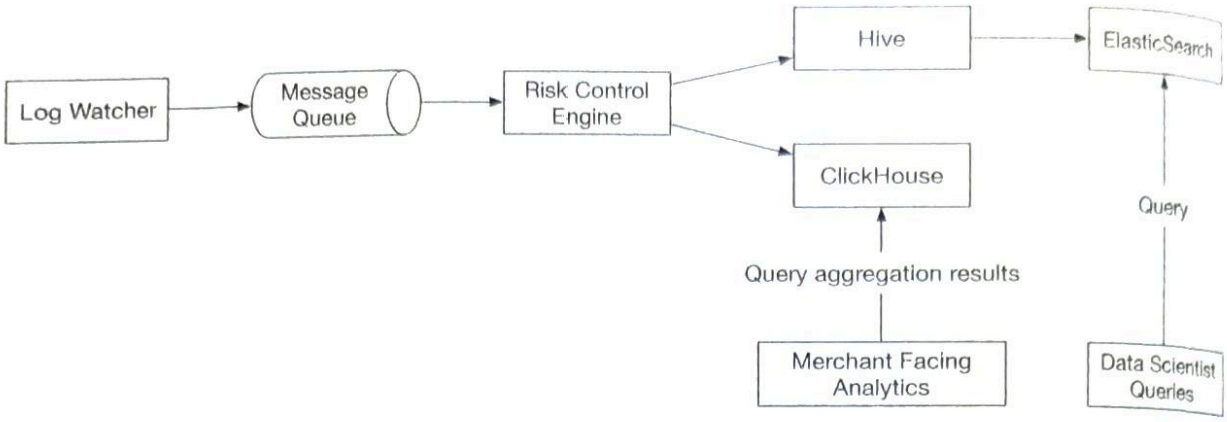


Figure 6.29: Alternative design

For more detail on this, please refer to reference material [25].

Step 4 - Wrap Up

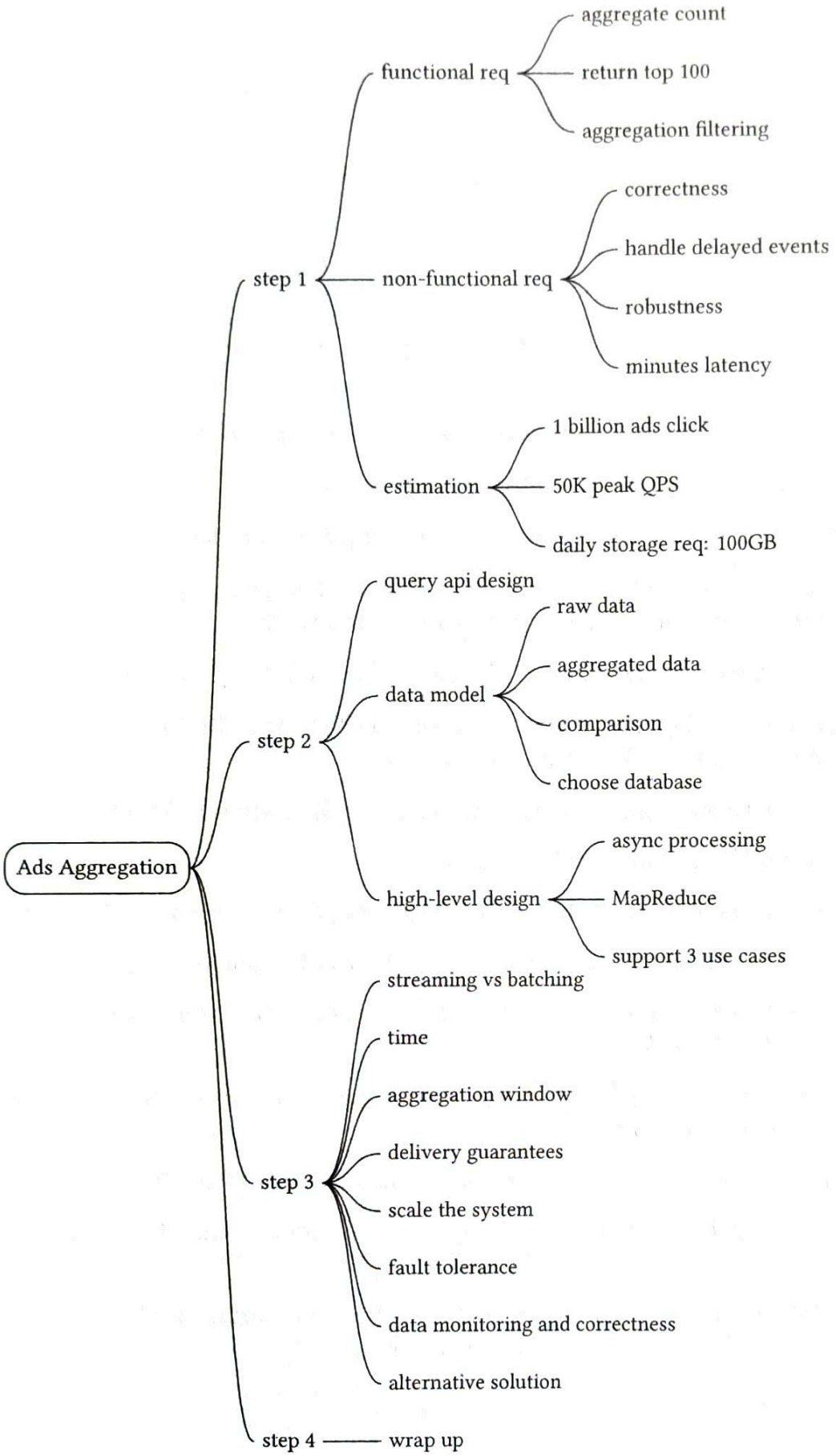
In this chapter, we went through the process of designing an ad click event aggregation system at the scale of Facebook or Google. We covered:

- Data model and API design.
- Use MapReduce paradigm to aggregate ad click events.
- Scale the message queue, aggregation service, and database.
- Mitigate hotspot issue.
- Monitor the system continuously.
- Use reconciliation to ensure correctness.
- Fault tolerance.

The ad click event aggregation system is a typical big data processing system. It will be easier to understand and design if you have prior knowledge or experience with industry-standard solutions such as Apache Kafka, Apache Flink, or Apache Spark.

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

Chapter Summary



Reference Material

[1] Clickthrough rate (CTR): Definition. https://support.google.com/google-ads/answer/2615875?hl=en.

[2] Conversion rate: Definition. https://support.google.com/google-ads/answer/2684489?hl=en.

[3] OLAP functions. https://docs.oracle.com/database/121/OLAXS/olap_functions.htm#OLAXS169.

[4] Display Advertising with Real-Time Bidding (RTB) and Behavioural Targeting. https://arxiv.org/pdf/1610.03013.pdf.

[5] LanguageManual ORC. https://cwiki.apache.org/confluence/display/hive/languagemanual+orc.

[6] Parquet. https://databricks.com/glossary/what-is-parquet.

[7] What is avro. https://www.ibm.com/topics/avro.

[8] Big Data. https://www.datakwery.com/techniques/big-data/.

[9] An Overview of End-to-End Exactly-Once Processing in Apache Flink. https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html.

[10] DAG model. https://en.wikipedia.org/wiki/Directed_acyclic_graph.

[11] Understand star schema and the importance for Power BI. https://docs.microsoft.com/en-us/power-bi/guidance/star-schema.

[12] Martin Kleppmann. *Designing Data-Intensive Applications.* O'Reilly Media, 2017.

[13] Apache Flink. https://flink.apache.org/.

[14] Lambda architecture. https://databricks.com/glossary/lambda-architecture.

[15] Kappa architecture. https://hazelcast.com/glossary/kappa-architecture/.

[16] Martin Kleppmann. Stream Processing. In *Designing Data-Intensive Applications.* O'Reilly Media, 2017.

[17] End-to-end Exactly-once Aggregation Over Ad Streams. https://www.youtube.com/watch?v=hzxytnPcAUM.

[18] Ad traffic quality. https://www.google.com/ads/adtrafficquality/.

[19] Understanding MapReduce in Hadoop. https://www.section.io/engineering-education/understanding-map-reduce-in-hadoop/.

[20] Flink on Apache Yarn. https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/deployment/resource-providers/yarn/.

[21] How data is distributed across a cluster (using virtual nodes). https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeDistribute.html.

[22] Flink performance tuning. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/table/tuning/.

[23] ClickHouse. https://clickhouse.com/.

[24] Druid. https://druid.apache.org/.

[25] Real-Time Exactly-Once Ad Event Processing with Apache Flink, Kafka, and Pinot. https://eng.uber.com/real-time-exactly-once-ad-event-processing/.