- Partition key: responsible for distributing data across nodes. As a general rule, we want to spread the data evenly.
- Clustering key: responsible for sorting data within a partition.

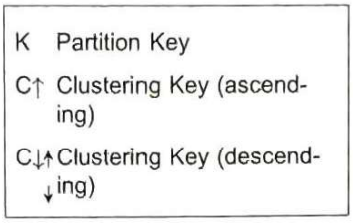
At a high level, an email service needs to support the following queries at the data layer:

- The first query is to get all folders for a user.
- The second query is to display all emails for a specific folder.
- The third query is to create/delete/get a specific email.
- The fourth query is to fetch all read or unread emails.
- Bonus point: get conversation threads.

Let's take a look at them one by one.

Query 1: get all folders for a user.

As shown in Table 8.1, `user_id` is the partition key, so folders owned by the same user are located in one partition.



folders_by_user		
user_id	UUID	K
folder_id	UUID	
folder_name	TEXT	

Table 8.1: Folders by user

Query 2: display all emails for a specific folder.

When a user loads their inbox, emails are usually sorted by timestamp, showing the most recent ones at the top. In order to store all emails for the same folder in one partition, composite partition key `<user_id, folder_id>` is used. Another column to note is `email_id`. Its data type is `TIMEUUID` [17], and it is the clustering key used to sort emails in chronological order.

emails_by_folder		
user_id	UUID	K
folder_id	UUID	K
email_id	TIMEUUID	C↓
from	TEXT	
subject	TEXT	
preview	TEXT	
is_read	BOOLEAN	

Table 8.2: Emails by folder

Query 3: create/delete/get an email

Due to space limitations, we only explain how to get detailed information about an email. The two tables in Table 8.3 are designed to support this query. The simple query looks like this:

```
SELECT * FROM emails_by_user WHERE email_id = 123;
```

An email can have multiple attachments, and these can be retrieved by the combination of `email_id` and `filename` fields.

emails_by_user		
user_id	UUID	K
email_id	TIMEUUID	C↓
from	TEXT	
to	LIST<TEXT>	
subject	TEXT	
body	TEXT	
attachments	LIST<filename|size>	

attachments		
email_id	TIMEUUID	C
filename	TEXT	K
url	TEXT	

Table 8.3: Emails by user

Query 4: fetch all read or unread emails

If our domain model was for a relational database, the query to fetch all read emails would look like this:

```
SELECT * FROM emails_by_folder
WHERE user_id = <user_id> and folder_id = <folder_id> and
  is_read = true
ORDER BY email_id;
```

The query to fetch all unread emails would look very similar. We just need to change `is_read = true` to `is_read = false` in the above query.

Our data model, however, is designed for NoSQL. A NoSQL database normally only supports queries on partition and cluster keys. Since `is_read` in the `emails_by_folder` table is neither of those, most NoSQL databases will reject this query.

One way to get around this limitation is to fetch the entire folder for a user and perform the filtering in the application. This could work for a small email service, but at our design scale, this does not work well.

This problem is commonly solved with denormalization in NoSQL. To support the read-/unread queries, we denormalize the `emails_by_folder` data into two tables as shown in Table 8.4.

- `read_emails`: it stores all emails that are in read status.
- `unread_emails`: it stores all emails that are in unread status.

To mark an `UNREAD` email as `READ`, the email is deleted from `unread_emails` and then inserted to `read_emails`.

To fetch all unread emails for a specific folder, we can run a query like this:

```
SELECT * FROM unread_emails
WHERE user_id = <user_id> and folder_id = <folder_id>
ORDER BY email_id;
```

read_emails		
user_id	UUID	K
folder_id	UUID	K
email_id	TIMEUUID	C↓
from	TEXT	
subject	TEXT	
preview	TEXT	

unread_emails		
user_id	UUID	K
folder_id	UUID	K
email_id	TIMEUUID	C↓
from	TEXT	
subject	TEXT	
preview	TEXT	

Table 8.4: Read and unread emails

Denormalization as shown above is a common practice. It makes the application code more complicated and harder to maintain, but it improves the read performance of these queries at scale.

Bonus point: conversation threads

Threads are a feature supported by many email clients. It groups email replies with their original message [8]. This allows users to retrieve all emails associated with one conversation. Traditionally, a thread is implemented using algorithms such as JWZ algorithm [18]. We will not go into detail about the algorithm, but just explain the core idea behind it. An email header generally contains the following three fields:

```
{
  "headers" {
    "Message-Id": "<7BA04B2A-430C-4D12-8B57-862103C34501@gmail.
        com>",
    "In-Reply-To": "<CAEWTXuPfN=LzECjDJtgY9Vu03kgFvJnJUSHTt6
        TW@gmail.com>",
    "References": ["<7BA04B2A-430C-4D12-8B57-862103C34501@gmail
        .com>"]
  }
}
```

Message-Id	The value of a message ID. It is generated by a client while sending a message.
In-Reply-To	The parent Message-Id to which the message replies.
References	A list of message IDs related to a thread.

Table 8.5: Email header

With these fields, an email client can reconstruct mail conversations from messages, if all messages in the reply chain are preloaded.

Consistency trade-off

Distributed databases that rely on replication for high availability must make a fundamental trade-off between consistency and availability. Correctness is very important for email systems, so by design, we want to have a single primary for any given mailbox. In the event of a failover, the mailbox isn't accessible by clients, so their sync/update operation is paused until failover ends. It trades availability in favor of consistency.

Email deliverability

It is easy to set up a mail server and start sending emails. The hard part is to get emails actually delivered to a user's inbox. If an email ends up in the spam folder, it means there is a very high chance a recipient won't read it. Email spam is a huge issue. According to research done by Statista [19], more than 50% of all emails sent are spam. If we set up a new mail server, most likely our emails will end up in the spam folder because a new email server has no reputation. There are a couple of factors to consider to improve email deliverability.

Dedicated IPs. It is recommended to have dedicated IP addresses for sending emails. Email providers are less likely to accept emails from new IP addresses that have no history.

Classify emails. Send different categories of emails from different IP addresses. For example, you may want to avoid sending marketing and other important emails from the same servers because it might make ISPs mark all emails as promotional.

Email sender reputation. Warm up new email server IP addresses slowly to build a good reputation, so big providers such as Office365, Gmail, Yahoo Mail, etc. are less likely to put our emails in the spam folder. According to Amazon Simple Email Service [20], it takes about 2 to 6 weeks to warm up a new IP address.

Ban spammers quickly. Spammers should be banned quickly before they have a significant impact on the server's reputation.

Feedback processing. It's very important to set up feedback loops with ISPs so we can keep the complaint rate low and ban spam accounts quickly. If an email fails to deliver or a user complains, one of the following outcomes occurs:

- Hard bounce. This means an email is rejected by ISP because the recipient's email address is invalid.
- Soft bounce. A soft bounce indicates an email failed to deliver due to temporary conditions, such as ISPs being too busy.
- Complaint. This means a recipient clicks the "report spam" button.

Figure 8.8 shows the process of collecting and processing bounces/complaints. We use separate queues for soft bounces, hard bounces, and complaints so they can be managed separately.

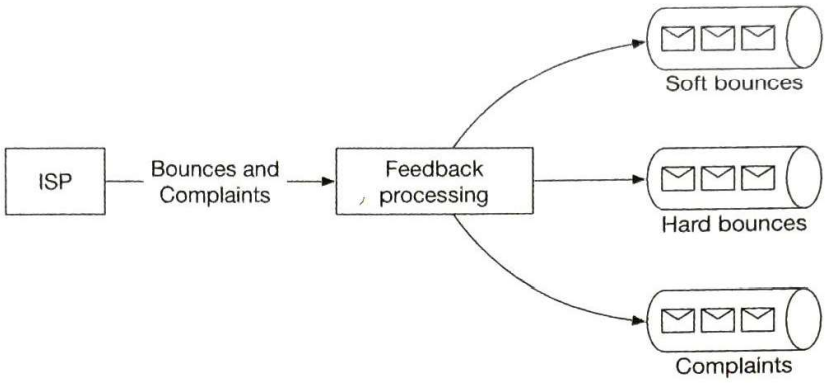


Figure 8.8: Handle feedback loop

Email authentication. According to the 2018 data breach investigation report provided by Verizon, phishing and pretexting represent 93% of breaches [21]. Some of the common techniques to combat phishing are: Sender Policy Framework (SPF) [22], DomainKeys Identified Mail (DKIM) [23], and Domain-based Message Authentication, Reporting and Conformance (DMARC) [24].

Figure 8.9 shows an example header of a Gmail message. As you can see, the sender `@info6.citi.com` is authenticated by SPF, DKIM, and DMARC.

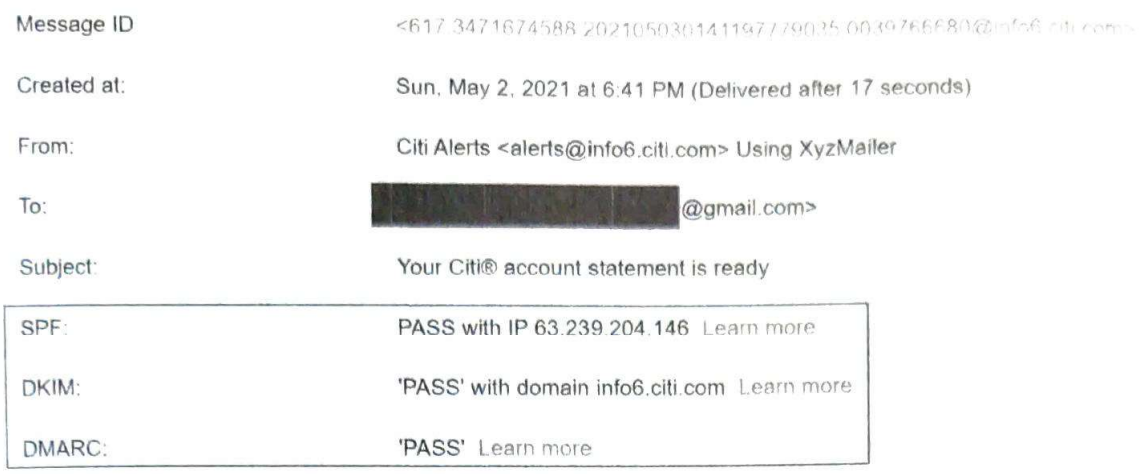


Figure 8.9: An example of a Gmail header

You don't need to remember all those terms. The important thing to keep in mind is that getting emails to work as intended is hard. It requires not only domain knowledge, but good relationships with ISPs.

Search

Basic mail search refers to searching for emails that contain any of the entered keywords in the subject or body. More advanced features include filtering by "From", "Subject", "Unread", or other attributes. On one hand, whenever an email is sent, received, or deleted, we need to perform reindexing. On the other hand, a search query is only run when a user presses the search button. This means the search feature in email systems has a lot more writes than reads. By comparison with Google search, email search has quite different characteristics, as shown in Table 8.6.

	Scope	**Sorting**	**Accuracy**
Google search	The whole internet	Sort by relevance	Indexing generally takes time, so some items may not show in the search result immediately.
Email search	User's own email box	Sort by attributes such as time, has attachment, date within, is unread, etc.	Indexing should be near real-time, and the result has to be accurate.

Table 8.6: Google search vs email search

To support search functionality, we compare two approaches: Elasticsearch and native search embedded in the datastore.

Option 1: Elasticsearch

The high-level design for email search using Elasticsearch is shown in Figure 8.10. Because queries are mostly performed on the user's own email server, we can group underlying documents to the same node using `user_id` as the partition key.

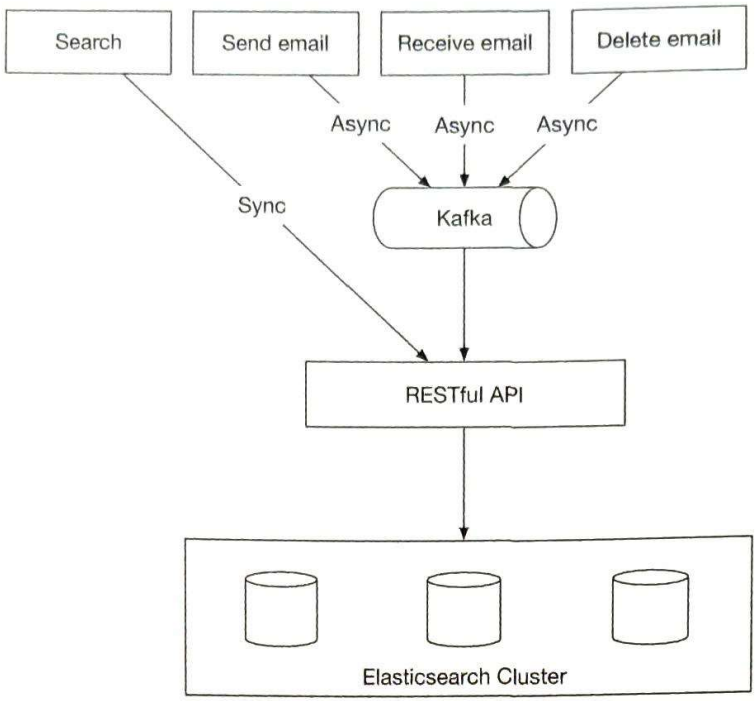


Figure 8.10: Elasticsearch

When a user clicks the search button, the user waits until the search response is received. A search request is synchronous. When events such as "send email", "receive email" or "delete email" are triggered, nothing related to search needs to be returned to the client. Reindexing is needed and it can be done with offline jobs. Kafka is used in the design to decouple services that trigger reindexing, from services that actually perform reindexing.

Elasticsearch is the most popular search-engine database as of June 2021 [25] and it supports full-text search of emails very well. One challenge of adding Elasticsearch is to keep our primary email store in sync with it.

Option 2: Custom search solution

Large-scale email providers usually develop their own custom search engines to meet their specific requirements. Designing an email search engine is a very complicated task and is out of the scope of this chapter. Here we only briefly touch on the disk I/O bottleneck, a primary challenge we will face for a custom search engine.

As shown in the back-of-the-envelope calculation, the size of the metadata and attachments added daily is at the petabyte (PB) level. Meanwhile, an email account can easily have over half a million emails. The main bottleneck of the index server is usually disk I/O.

Since the process of building the index is write-heavy, a good strategy might be to use Log-Structured Merge-Tree (LSM) [26] to structure the index data on disk (Figure 8.11). The write path is optimized by only performing sequential writes. LSM trees are the core data structure behind databases such as Bigtable, Cassandra, and RocksDB. When a new email arrives, it is first added to level 0 in-memory cache, and when data size in memory reaches the predefined threshold, data is merged to the next level. Another reason to use LSM is to separate data that change frequently from those that don't. For example, email data usually doesn't change, but folder information tends to change more often due to different filter rules. In this case, we can separate them into two different sections, so that if a request is related to a folder change, we change only the folder and leave the email data alone.

If you are interested in reading more about email search, it is highly recommended you take a look at how search works in Microsoft Exchange servers [27].

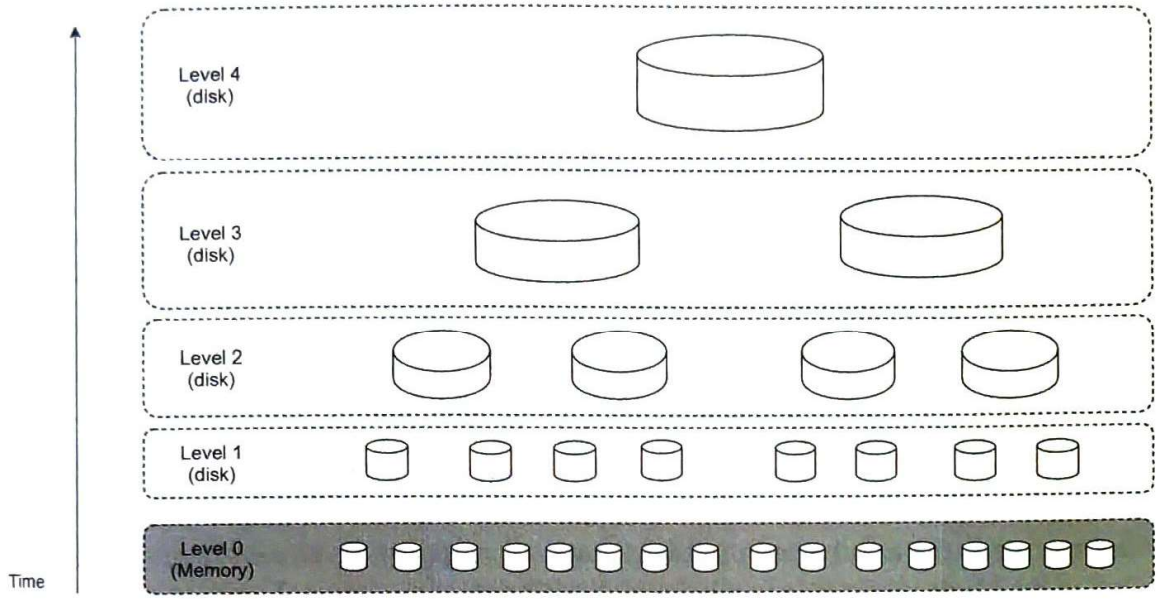


Figure 8.11: LSM tree

Each approach has pros and cons:

Feature	Elasticsearch	Custom search engine
Scalability	Scalable to some extent	Easier to scale as we can optimize the system for the email use case
System complexity	Need to maintain two different systems: datastore and Elasticsearch	One system
Data consistency	Two copies of data. One in the metadata datastore, and the other in Elasticsearch. Data consistency is hard to maintain	A single copy of data in the metadata datastore
Data loss possible	No. Can rebuild the Elasticsearch index from the primary storage, in case of failure	No
Development effort	Easy to integrate. To support large scale email search, a dedicated Elasticsearch team might be needed	Significant engineering effort is needed to develop a custom email search engine

Table 8.7: Elastic search vs custom search engine

A general rule of thumb is that for a smaller scale email system, Elasticsearch is a good option as it's easy to integrate and doesn't require significant engineering effort. For a larger scale, Elasticsearch might work, but we may need a dedicated team to develop and maintain the email search infrastructure. To support an email system at Gmail or Outlook scale, it might be a good idea to have a native search embedded in the database as opposed to the separate indexing approach.

Scalability and availability

Since data access patterns of individual users are independent of one another, we expect most components in the system are horizontally scalable.

For better availability, data is replicated across multiple data centers. Users communicate with a mail server that is physically closer to them in the network topology. During a network partition, users can access messages from other data centers (Figure 8.12).

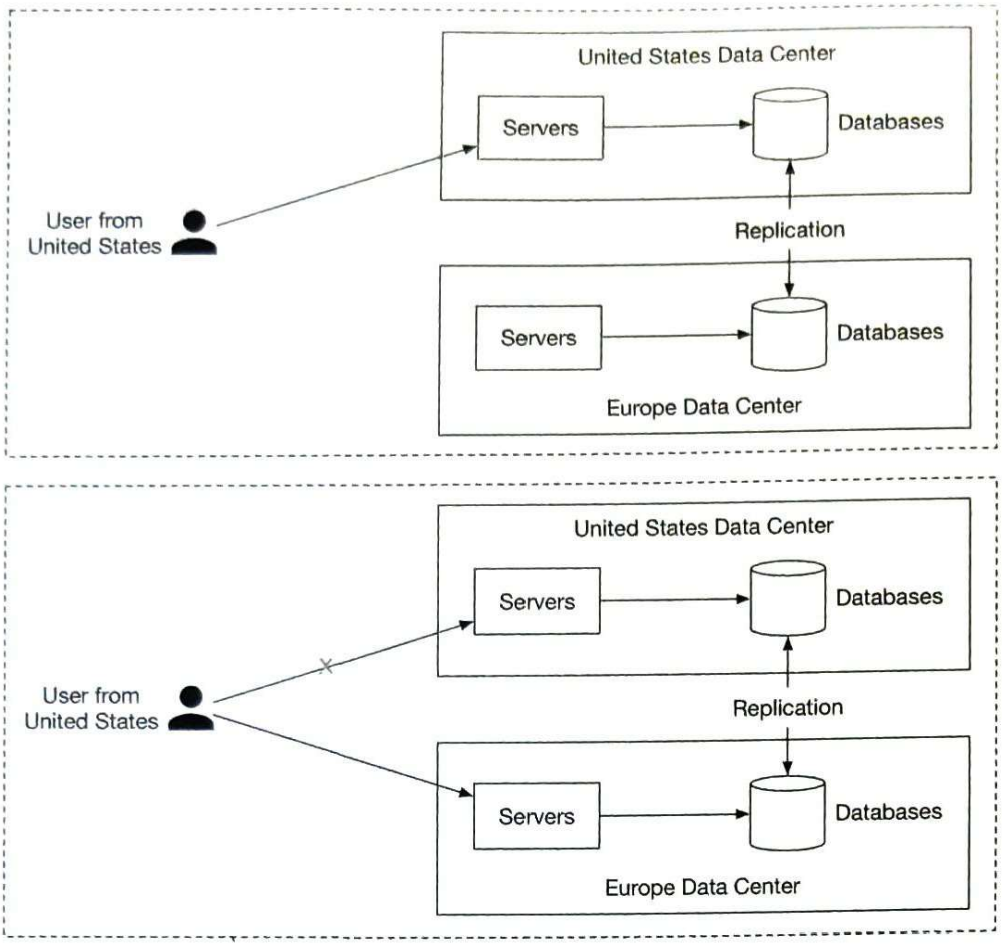


Figure 8.12: Multi-data center setup

Step 4 - Wrap Up

In this chapter, we have presented a design for building large-scale email servers. We started by gathering requirements and doing some back-of-the-envelope calculations to get a good idea of the scale. In the high-level design, we discussed how traditional email servers were designed and why they cannot satisfy modern use cases. We also discussed email APIs and high-level designs for sending and receiving flows. Finally, we dived deep into metadata database design, email deliverability, search, and scalability.

If there is extra time at the end of the interview, here are a few additional talking points:

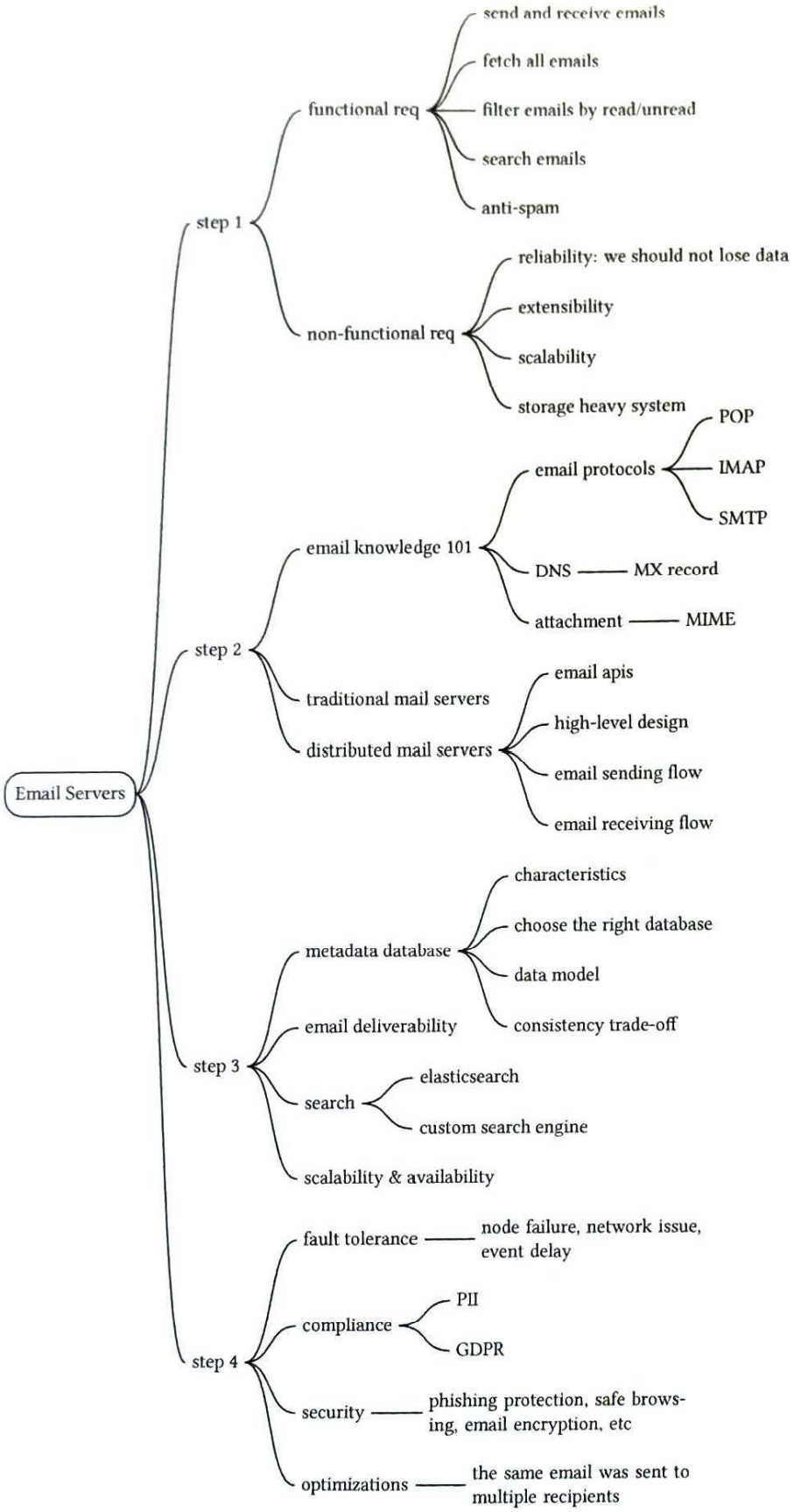
- Fault tolerance. Many parts of the system can fail, and you can talk about how to handle node failures, network issues, event delays, etc.
- Compliance. Email service works all around the world and there are legal regulations to comply with. For instance, we need to handle and store personally identifiable information (PII) from Europe in a way that complies with General Data Protection Regulation (GDPR) [28]. Legal intercept is another typical feature in this area [29].
- Security. Email security is important because emails contain sensitive information.

Gmail provides safety features such as phishing protections, safe browsing, proactive alerts, account safety, confidential mode, and email encryption [30].

- Optimizations. Sometimes, the same email is sent to multiple recipients, and the same email attachment is stored several times in the object store (S3) in the group emails. One optimization we could do is to check the existence of the attachment in storage, before performing the expensive save operation.

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

Chapter Summary



Reference Material

[1] Number of Active Gmail Users. https://financesonline.com/number-of-active-gmail-users/.

[2] Outlook. https://en.wikipedia.org/wiki/Outlook.com.

[3] How Many Emails Are Sent Per Day in 2021? https://review42.com/resources/how-many-emails-are-sent-per-day/.

[4] RFC 1939 - Post Office Protocol - Version 3. http://www.faqs.org/rfcs/rfc1939.html.

[5] ActiveSync. https://en.wikipedia.org/wiki/ActiveSync.

[6] Email attachment. https://en.wikipedia.org/wiki/Email_attachment.

[7] MIME. https://en.wikipedia.org/wiki/MIME.

[8] Threading. https://en.wikipedia.org/wiki/Conversation_threading.

[9] IMAP LIST Extension for Special-Use Mailboxes. https://datatracker.ietf.org/doc/html/rfc6154.

[10] Apache James. https://james.apache.org/.

[11] A JSON Meta Application Protocol (JMAP) Subprotocol for WebSocket. https://tools.ietf.org/id/draft-ietf-jmap-websocket-07.html#RFC7692.

[12] Cassandra Limitations. https://cwiki.apache.org/confluence/display/CASSANDRA2/CassandraLimitations.

[13] Inverted index. https://en.wikipedia.org/wiki/Inverted_index.

[14] Exponential backoff. https://en.wikipedia.org/wiki/Exponential_backoff.

[15] QQ Email System Optimization (in Chinese). https://www.slideshare.net/areyouok/06-qq-5431919.

[16] IOPS. https://en.wikipedia.org/wiki/IOPS.

[17] UUID and timeuuid types. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/uuid_type_r.html.

[18] Message threading. https://www.jwz.org/doc/threading.html.

[19] Global spam volume. https://www.statista.com/statistics/420391/spam-email-traffic-share/.

[20] Warming up dedicated IP addresses. https://docs.aws.amazon.com/ses/latest/dg/dedicated-ip-warming.html.

[21] 2018 Data Breach Investigations Report. https://enterprise.verizon.com/resources/reports/DBIR_2018_Report.pdf.

[22] Sender Policy Framework. https://en.wikipedia.org/wiki/Sender_Policy_Framework.

[23] DomainKeys Identified Mail. https://en.wikipedia.org/wiki/DomainKeys_Identified_Mail.

[24] Domain-based Message Authentication, Reporting & Conformance. https://dmarc.org/.

[25] DB-Engines Ranking of Search Engines. https://db-engines.com/en/ranking/search+engine.

[26] Log-structured merge-tree. https://en.wikipedia.org/wiki/Log-structured_merge-tree.

[27] Microsoft Exchange Conference 2014 Search in Exchange. https://www.youtube.com/watch?v=5EXGCSzzQak&t=2173s.

[28] General Data Protection Regulation. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation.

[29] Lawful interception. https://en.wikipedia.org/wiki/Lawful_interception.

[30] Email safety. https://safety.google/intl/en_us/gmail/.

9 S3-like Object Storage

In this chapter, we design an object storage service similar to Amazon Simple Storage Service (S3). S3 is a service offered by Amazon Web Services (AWS) that provides object storage through a RESTful API-based interface. Here are some facts about AWS S3:

- Launched in June 2006.
- S3 added versioning, bucket policy, and multipart upload support in 2010.
- S3 added server-side encryption, multi-object delete, and object expiration in 2011.
- Amazon reported 2 trillion objects stored in S3 by 2013.
- Life cycle policy, event notification, and cross-region replication support were introduced in 2014 and 2015.
- Amazon reported over 100 trillion objects stored in S3 by 2021.

Before we dig into object storage, let's first review storage systems in general and define some terminologies.

Storage System 101

At a high-level, storage systems fall into three broad categories:

- Block storage
- File storage
- Object storage

Block storage

Block storage came first, in the 1960s. Common storage devices like hard disk drives (HDD) and solid-state drives (SSD) that are physically attached to servers are all considered as block storage.

Block storage presents the raw blocks to the server as a volume. This is the most flexible and versatile form of storage. The server can format the raw blocks and use them as a file system, or it can hand control of those blocks to an application. Some applications like

a database or a virtual machine engine manage these blocks directly in order to squeeze every drop of performance out of them.

Block storage is not limited to physically attached storage. Block storage could be connected to a server over a high-speed network or over industry-standard connectivity protocols like Fibre Channel (FC) [1] and iSCSI [2]. Conceptually, the network-attached block storage still presents raw blocks. To the servers, it works the same as physically attached block storage.

File storage

File storage is built on top of block storage. It provides a higher-level abstraction to make it easier to handle files and directories. Data is stored as files under a hierarchical directory structure. File storage is the most common general-purpose storage solution. File storage could be made accessible by a large number of servers using common file-level network protocols like SMB/CIFS [3] and NFS [4]. The servers accessing file storage do not need to deal with the complexity of managing the blocks, formatting volume, etc. The simplicity of file storage makes it a great solution for sharing a large number of files and folders within an organization.

Object storage

Object storage is new. It makes a very deliberate tradeoff to sacrifice performance for high durability, vast scale, and low cost. It targets relatively "cold" data and is mainly used for archival and backup. Object storage stores all data as objects in a flat structure. There is no hierarchical directory structure. Data access is normally provided via a RESTful API. It is relatively slow compared to other storage types. Most public cloud service providers have an object storage offering, such as AWS S3, Google object storage, and Azure blob storage.

Comparison

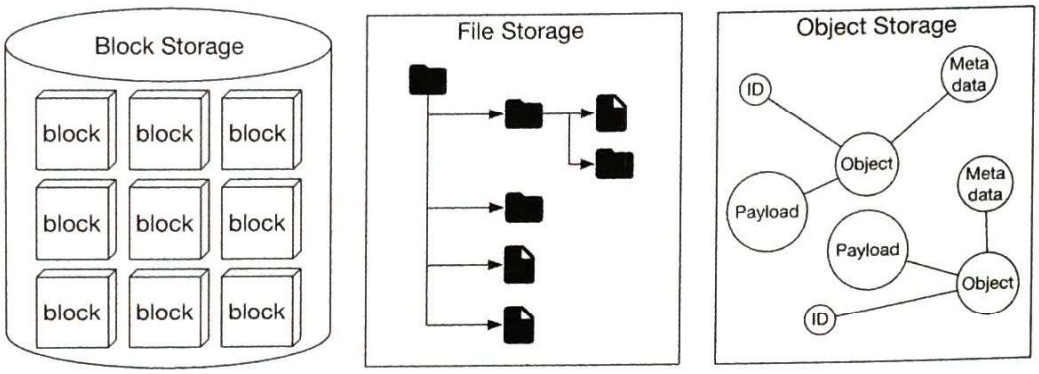


Figure 9.1: Three different storage options

Table 9.1 compares block storage, file storage, and object storage.

	Block storage	File storage	Object storage
Mutable Content	Y	Y	N (object versioning is supported, in-place update is not)
Cost	High	Medium to high	Low
Performance	Medium to high, very high	Medium to high	Low to medium
Consistency	Strong consistency	Strong consistency	Strong consistency [5]
Data access	SAS [6]/iSCSI/FC	Standard file access, CIFS/SMB, and NFS	RESTful API
Scalability	Medium scalability	High scalability	Vast scalability
Good for	Virtual machines (VM), high-performance applications like database	General-purpose file system access	Binary data, unstructured data

Table 9.1: Storage options

Terminology

To design S3-like object storage, we need to understand some core object storage concepts first. This section provides an overview of the terms that apply to object storage.

Bucket. A logical container for objects. The bucket name is globally unique. To upload data to S3, we must first create a bucket.

Object. An object is an individual piece of data we store in a bucket. It contains object data (also called payload) and metadata. Object data can be any sequence of bytes we want to store. The metadata is a set of name-value pairs that describe the object.

Versioning. A feature that keeps multiple variants of an object in the same bucket. It is enabled at bucket-level. This feature enables users to recover objects that are deleted or overwritten by accident.

Uniform Resource Identifier (URI). The object storage provides RESTful APIs to access its resources, namely, buckets and objects. Each resource is uniquely identified by its URI.

Service-level agreement (SLA). A service-level agreement is a contract between a service provider and a client. For example, the Amazon S3 Standard-Infrequent Access storage class provides the following SLA [7]:

- Designed for durability of 99.999999999% of objects across multiple Availability Zones.
- Data is resilient in the event of one entire Availability Zone destruction.

- Designed for 99.9% availability.

Step 1 - Understand the Problem and Establish Design Scope

The following questions help to clarify the requirements and narrow down the scope.

Candidate: Which features should be included in the design?
Interviewer: We would like you to design an S3-like object storage system with the following functionalities:

- Bucket creation.
- Object uploading and downloading.
- Object versioning.
- Listing objects in a bucket. It's similar to the `aws S3 ls` command [8].

Candidate: What is the typical data size?
Interviewer: We need to store both massive objects (a few GBs or more) and a large number of small objects (tens of KBs,) efficiently.

Candidate: How much data do we need to store in one year?
Interviewer: 100 petabytes (PB).

Candidate: Can we assume data durability is 6 nines (99.9999%) and service availability is 4 nines (99.99%)?
Interviewer: Yes, that sounds reasonable.

Non-functional requirements

- 100PB of data
- Data durability is 6 nines
- Service availability is 4 nines
- Storage efficiency. Reduce storage costs while maintaining a high degree of reliability and performance.

Back-of-the-envelope estimation

Object storage is likely to have bottlenecks in either disk capacity or disk IO per second (IOPS). Let's take a look.

- Disk capacity. Let's assume objects follow the distribution listed below:
  - 20% of all objects are small objects (less than 1MB).
  - 60% of objects are medium-sized objects (1 MB $\sim$ 64MB).
  - 20% are large objects (larger than 64MB).
- IOPS. Let's assume one hard disk (SATA interface, 7200 rpm) is capable of doing

$100 \sim 150$ random seeks per second ($100 \sim 150$ IOPS).

With those assumptions, we can estimate the total number of objects the system can persist. To simplify the calculation, let's use the median size for each object type (0.5MB for small objects, 32MB for medium objects, and 200MB for large objects). A 40% storage usage ratio gives us:

- $100\,\text{PB} = 100 \times 1000 \times 1000 \times 1000\,\text{MB} = 10^{11}\,\text{MB}$
- $\frac{10^{11} \times 0.4}{(0.2 \times 0.5\,\text{MB} + 0.6 \times 32\,\text{MB} + 0.2 \times 200\,\text{MB})} = 0.68$ billion objects.
- If we assume the metadata of an object is about 1KB in size, we need 0.68TB space to store all metadata information.

Even though we may not use those numbers, it's good to have a general idea about the scale and constraint of the system.

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

Before diving into the design, let's explore a few interesting properties of object storage, as they may influence it.

Object immutability. One of the main differences between object storage and the other two types of storage systems is that the objects stored inside of object storage are immutable. We may delete them or replace them entirely with a new version, but we cannot make incremental changes.

Key-value store. We could use object URI to retrieve object data (Listing 9.1). The object URI is the key and object data is the value.

```
Request:
GET /bucket1/object1.txt HTTP/1.1

Response:
HTTP/1.1 200 OK
Content-Length: 4567

[4567 bytes of object data]
```

Listing 9.1: Use object URI to retrieve object data

Write once, read many times. The data access pattern for object data is written once and read many times. According to the research done by LinkedIn, 95% of requests are read operations [9].

Support both small and large objects. Object size may vary and we need to support both.

The design philosophy of object storage is very similar to that of the UNIX file system. In UNIX, when we save a file in the local file system, it does not save the filename and file data together. Instead, the filename is stored in a data structure called "inode" [10],

and the file data is stored in different disk locations. The inode contains a list of file block pointers that point to the disk locations of the file data. When we access a local file, we first fetch the metadata in the inode. We then read the file data by following the file block pointers to the actual disk locations.

The object storage works similarly. The inode becomes the metadata store that stores all the object metadata. The hard disk becomes the data store that stores the object data. In the UNIX file system, the inode uses the file block pointer to record the location of data on the hard disk. In object storage, the metadata store uses the ID of the object to find the corresponding object data in the data store, via a network request. Figure 9.2 shows the UNIX file system and the object storage.

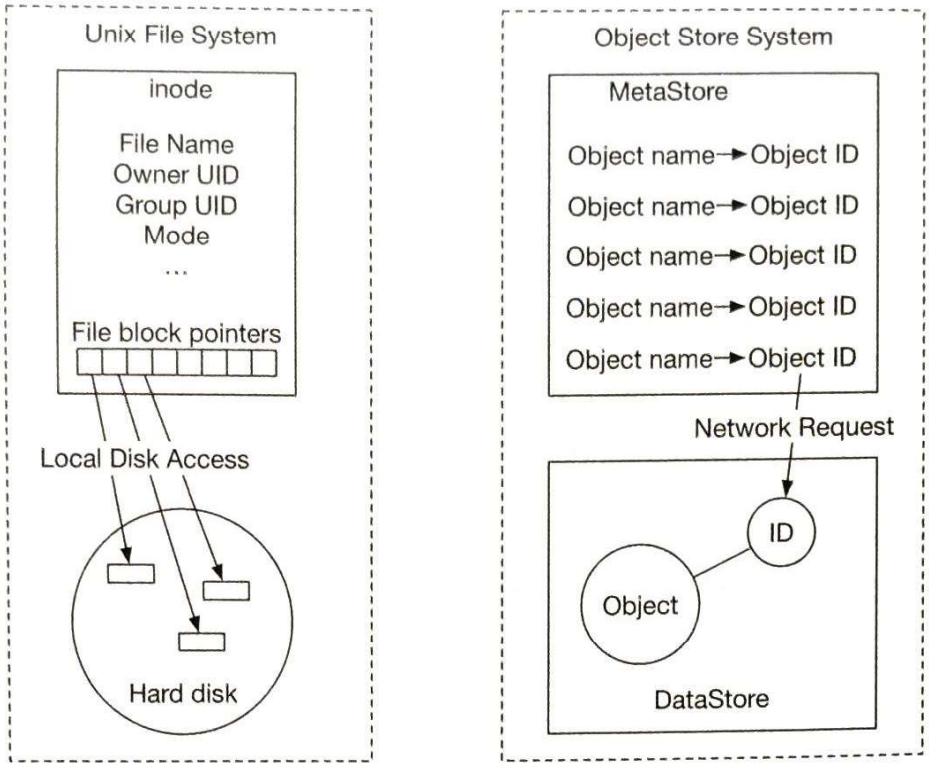


Figure 9.2: UNIX file system and object store

Separating metadata and object data simplifies the design. The data store contains immutable data while the metadata store contains mutable data. This separation enables us to implement and optimize these two components independently. Figure 9.3 shows what the bucket and object look like.

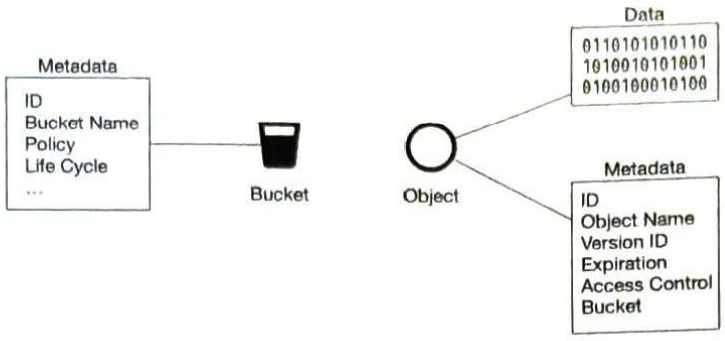


Figure 9.3: Bucket & object

High-level design

Figure 9.4 shows the high-level design.

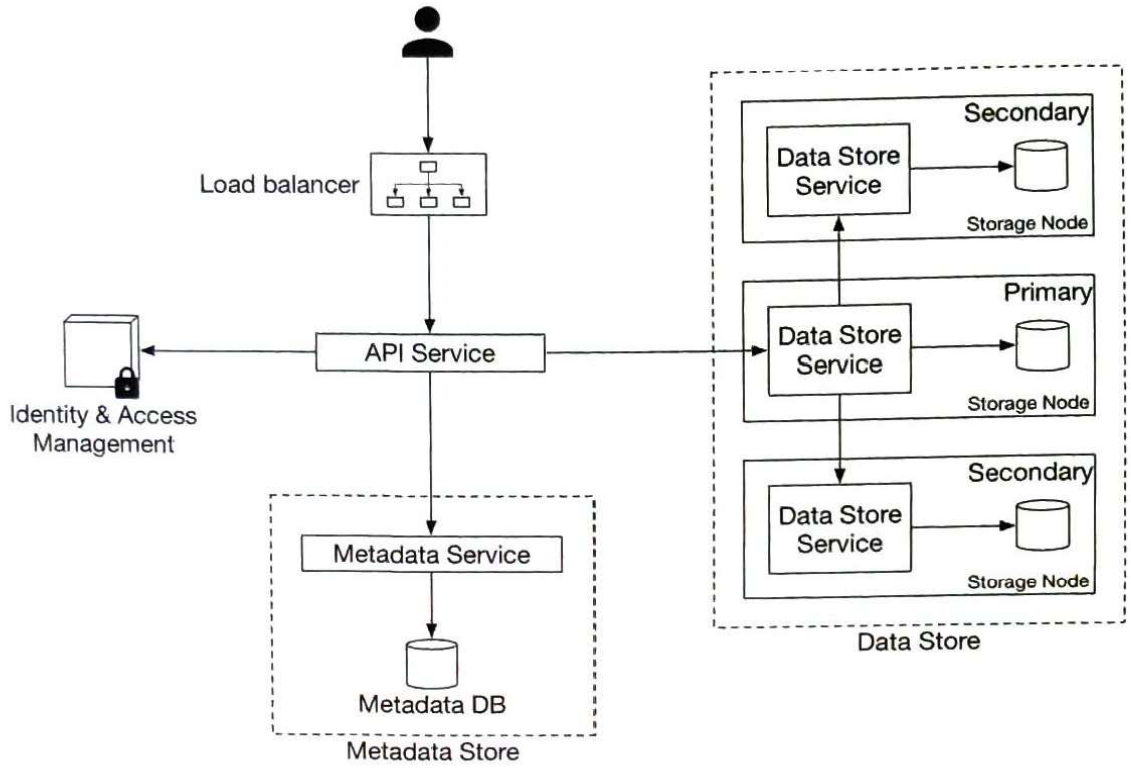


Figure 9.4: High-level design

Let's go over the components one by one.

Load balancer. Distributes RESTful API requests across a number of API servers.

API service. Orchestrates remote procedure calls to the identity and access management service, metadata service, and storage stores. This service is stateless so it can be horizontally scaled.

Identity and access management (IAM). The central place to handle authentication, authorization, and access control. Authentication verifies who you are, and authorization validates what operations you could perform based on who you are.

Data store. Stores and retrieves the actual data. All data-related operations are based on object ID (UUID).

Metadata store. Stores the metadata of the objects.

Note that the metadata and data stores are just logical components, and there are different ways to implement them. For example, in Ceph's Rados Gateway [11], there is no standalone metadata store. Everything, including the object bucket, is persisted as one or multiple Rados objects.

Now we have a basic understanding of the high-level design, let's explore some of the most important workflows in object storage.

- Uploading an object.
- Downloading an object.
- Object versioning and listing objects in a bucket. They will be explained in the "design deep dive" section on page 263.

Uploading an object

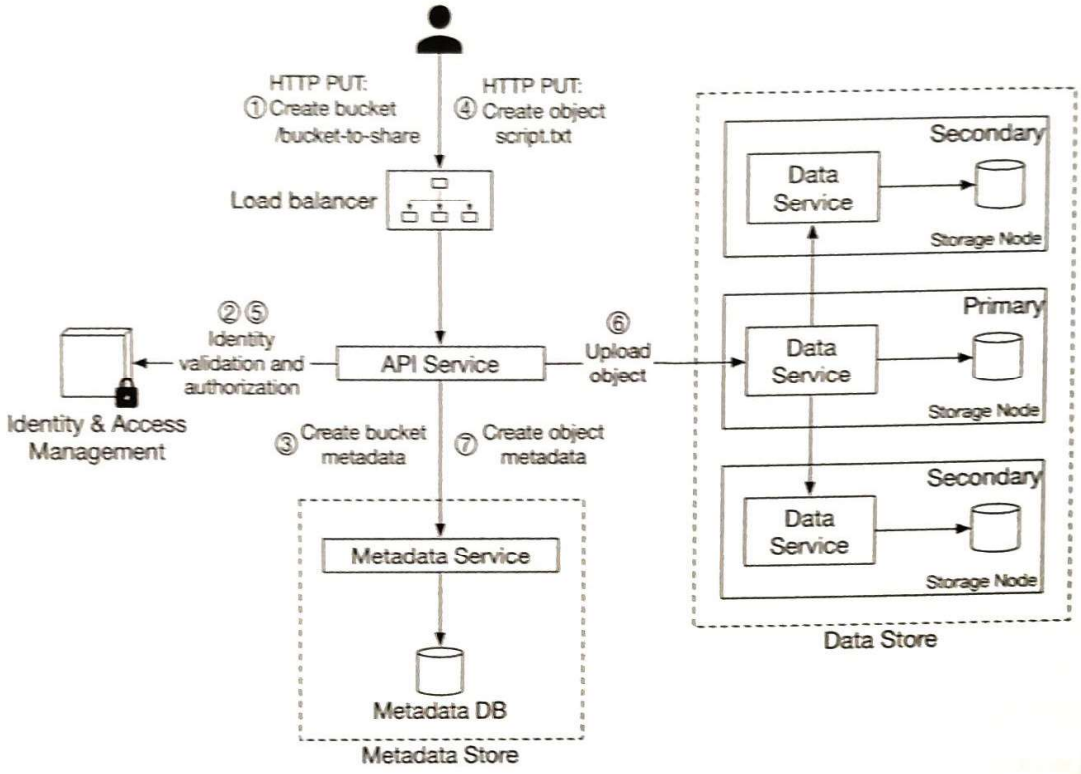


Figure 9.5: Uploading an object

An object has to reside in a bucket. In this example, we first create a bucket named `bucket-to-share` and then upload a file named `script.txt` to the bucket. Figure 9.5 explains how this flow works in 7 steps.

1. The client sends an `HTTP PUT` request to create a bucket named `bucket-to-share`. The

request is forwarded to the API service.

2. The API service calls the IAM to ensure the user is authorized and has `WRITE` permission.
3. The API service calls the metadata store to create an entry with the bucket info in the metadata database. Once the entry is created, a success message is returned to the client.
4. After the bucket is created, the client sends an `HTTP PUT` request to create an object named `script.txt`.
5. The API service verifies the user's identity and ensures the user has `WRITE` permission on the bucket.
6. Once validation succeeds, the API service sends the object data in the `HTTP PUT` payload to the data store. The data store persists the payload as an object and returns the UUID of the object.
7. The API service calls the metadata store to create a new entry in the metadata database. It contains important metadata such as the `object_id` (UUID), `bucket_id` (which bucket the object belongs to), `object_name`, etc. A sample entry is shown in Table 9.2.

object_name	object_id	bucket_id
script.txt	239D5866-0052-00F6-014E-C914E61ED42B	82AA1B2E-F599-4590-B5E4-1F51AAE5F7E4

Table 9.2: Sample entry

The API to upload an object could look like this:

```
PUT /bucket-to-share/script.txt HTTP/1.1
Host: foo.s3example.org
Date: Sun, 12 Sept 2021 17:51:00 GMT
Authorization: authorization string
Content-Type: text/plain
Content-Length: 4567
x-amz-meta-author: Alex

[4567 bytes of object data]
```

Listing 9.2: Uploading an object

Downloading an object

A bucket has no directory hierarchy. However, we can create a logical hierarchy by concatenating the bucket name and the object name to simulate a folder structure. For example, we name the object `bucket-to-share/script.txt` instead of `script.txt`. To get an object, we specify the object name in the `GET` request. The API to download an object looks like this:

```
GET /bucket-to-share/script.txt HTTP/1.1
Host: foo.s3example.org
Date: Sun, 12 Sept 2021 18:30:01 GMT
Authorization: authorization string
```

Listing 9.3: Downloading an object

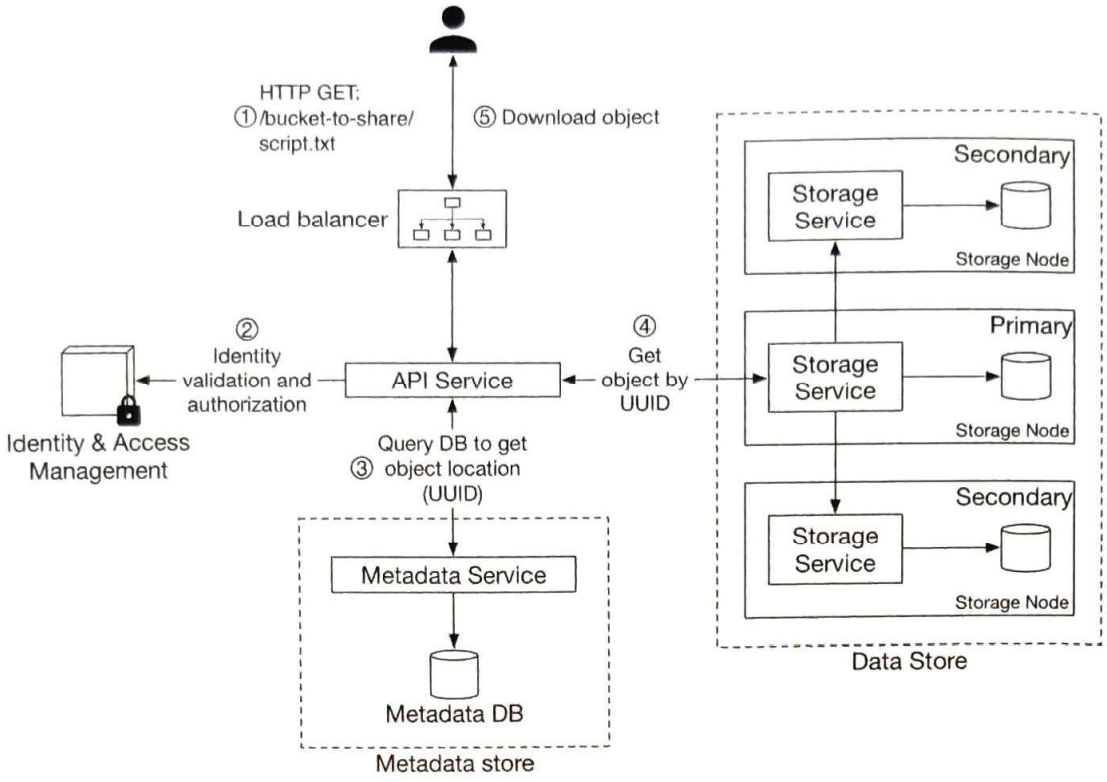


Figure 9.6: Downloading an object

As mentioned earlier, the data store does not store the name of the object and it only supports object operations via `object_id` (UUID). In order to download the object, we first map the object name to the UUID. The workflow of downloading an object is shown below:

1. The client sends an `HTTP GET` request to the load balancer: `GET /bucket-to-share/script.txt`
2. The API service queries the IAM to verify that the user has `READ` access to the bucket.
3. Once validated, the API service fetches the corresponding object's UUID from the metadata store.
4. Next, the API service fetches the object data from the data store by its UUID.
5. The API service returns the object data to the client in HTTP GET response.

Step 3 - Design Deep Dive

In this section, we dive deep into a few areas:

- Data store
- Metadata data model
- Listing objects in a bucket
- Object versioning
- Optimizing uploads of large files
- Garbage collection

Data store

Let's take a closer look at the design of the data store. As discussed previously, the API service handles external requests from users and calls different internal services to fulfill those requests. To persist or retrieve an object, the API service calls the data store. Figure 9.7 shows the interactions between the API service and the data store for uploading and downloading an object.

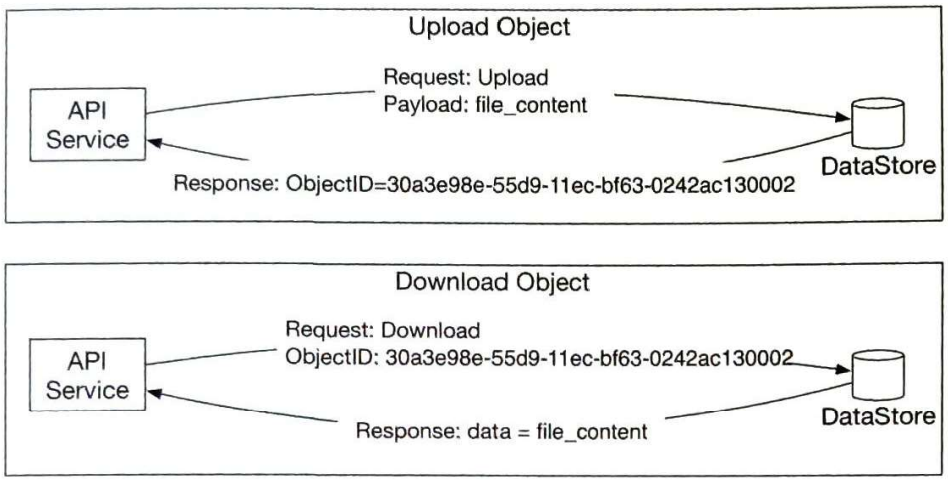


Figure 9.7: Upload and download an object

High-level design for the data store

The data store has three main components as shown in Figure 9.8.

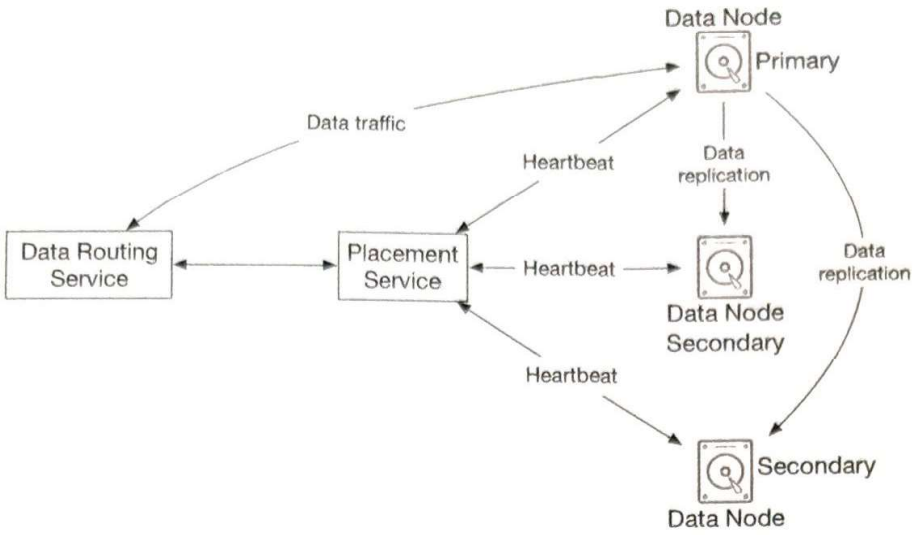


Figure 9.8: Data store components

Data routing service

The data routing service provides RESTful or gRPC [12] APIs to access the data node cluster. It is a stateless service that can scale by adding more servers. This service has the following responsibilities:

- Query the placement service to get the best data node to store data.
- Read data from data nodes and return it to the API service.
- Write data to data nodes.

Placement service

The placement service determines which data nodes (primary and replicas) should be chosen to store an object. It maintains a virtual cluster map, which provides the physical topology of the cluster. The virtual cluster map contains location information for each data node which the placement service uses to make sure the replicas are physically separated. This separation is key to high durability. See the "Durability" section on page 270 for details. An example of the virtual cluster map is shown in Figure 9.9.

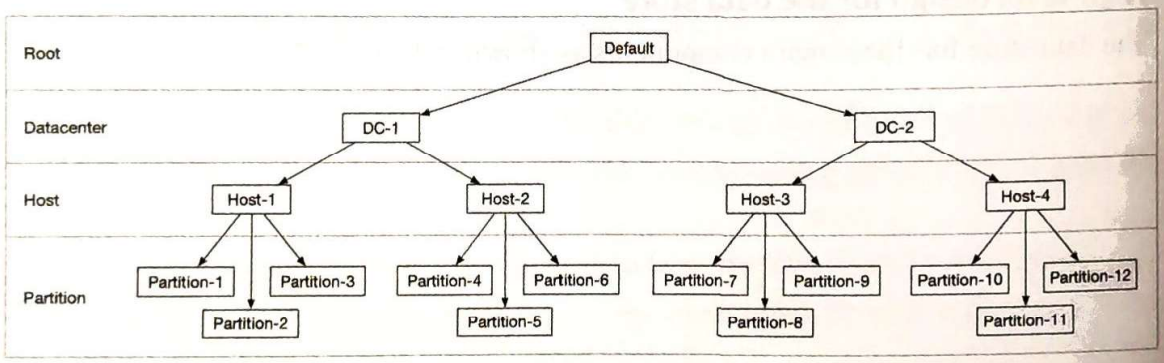


Figure 9.9: Virtual cluster map

The placement service continuously monitors all data nodes through heartbeats. If a data node doesn't send a heartbeat within a configurable 15-second grace period, the placement service marks the node as "down" in the virtual cluster map.

This is a critical service, so we suggest building a cluster of 5 or 7 placement service nodes with Paxos [13] or Raft [14] consensus protocol. The consensus protocol ensures that as long as more than half of the nodes are healthy, the service as a whole continues to work. For example, if the placement service cluster has 7 nodes, it can tolerate a 3 node failure. To learn more about consensus protocols, refer to the reference materials [13] [14].

Data node

The data node stores the actual object data. It ensures reliability and durability by replicating data to multiple data nodes, also called a replication group.

Each data node has a data service daemon running on it. The data service daemon continuously sends heartbeats to the placement service. The heartbeat message includes the following essential information:

- How many disk drives (HDD or SSD) does the data node manage?
- How much data is stored on each drive?

When the placement service receives the heartbeat for the first time, it assigns an ID for this data node, adds it to the virtual cluster map, and returns the following information:

- a unique ID of the data node
- the virtual cluster map
- where to replicate data

Data persistence flow

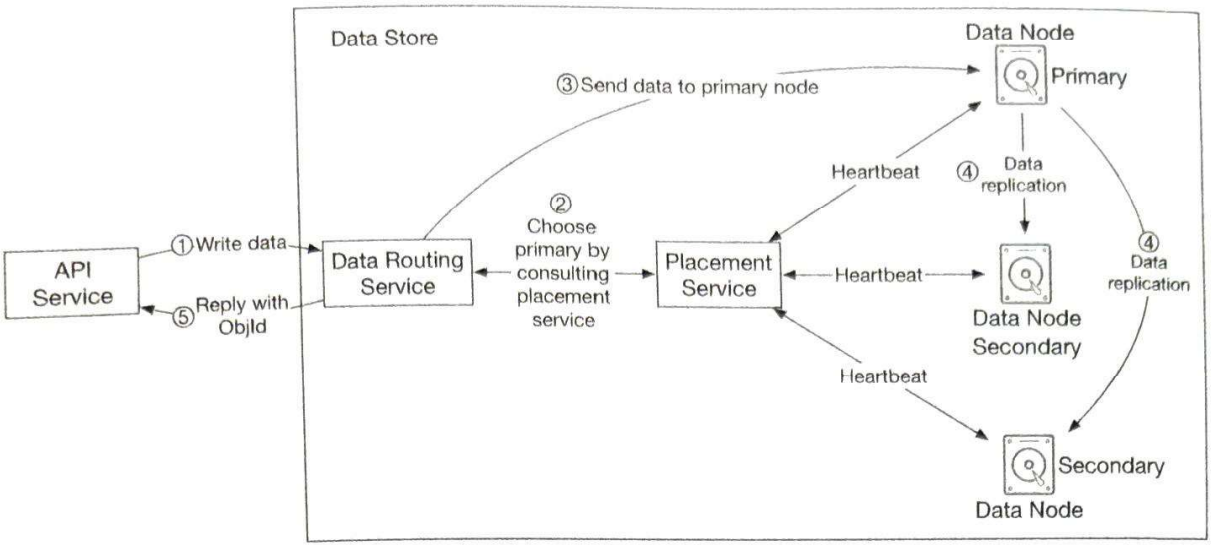


Figure 9.10: Data persistence flow

Now let's take a look at how data is persisted in the data node.

1. The API service forwards the object data to the data store.
2. The data routing service generates a UUID for this object and queries the placement service for the data node to store this object. The placement service checks the virtual cluster map and returns the primary data node.
3. The data routing service sends data directly to the primary data node, together with its UUID.
4. The primary data node saves the data locally and replicates it to two secondary data nodes. The primary node responds to the data routing service when data is successfully replicated to all secondary nodes.
5. The UUID of the object (ObjId) is returned to the API service.

In step 2, given a UUID for the object as an input, the placement service returns the replication group for the object. How does the placement service do this? Keep in mind that this lookup needs to be deterministic, and it must survive the addition or removal of replication groups. Consistent hashing is a common implementation of such a lookup function. Refer to [15] for more information.

In step 4, the primary data node replicates data to all secondary nodes before it returns a response. This makes data strongly consistent among all data nodes. This consistency comes with latency costs because we have to wait until the slowest replica finishes. Figure 9.11 shows the trade-offs between consistency and latency.

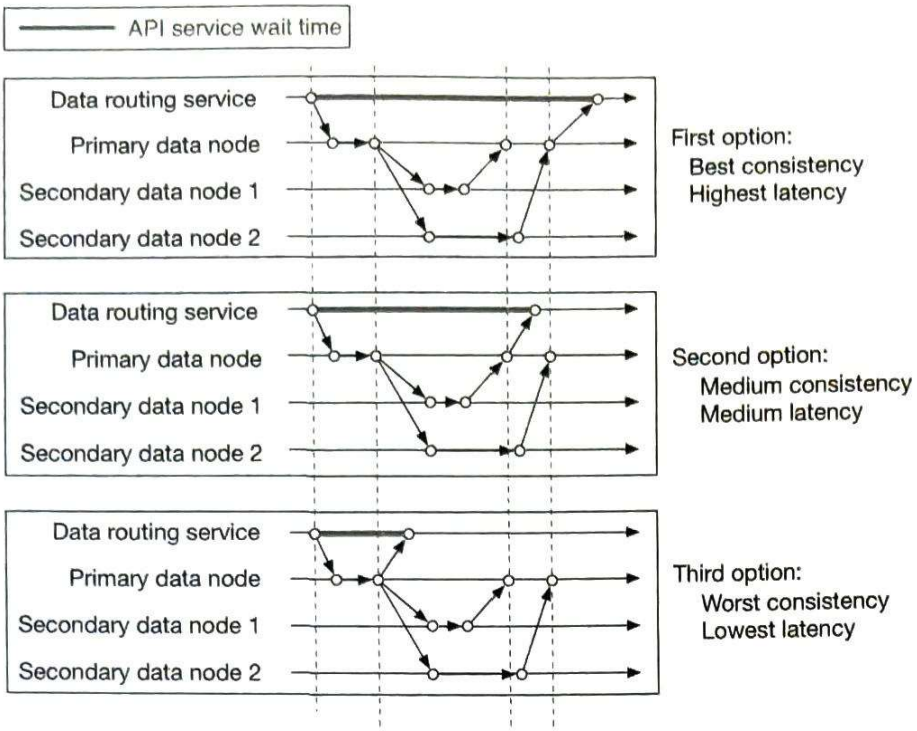


Figure 9.11: Trade-off between consistency and latency

1. Data is considered as successfully saved after all three nodes store the data. This approach has the best consistency but the highest latency.
2. Data is considered as successfully saved after the primary and one of the secondaries store the data. This approach has a medium consistency and medium latency.
3. Data is considered as successfully saved after the primary persists the data. This approach has the worst consistency but the lowest latency.

Both 2 and 3 are forms of eventual consistency.

How data is organized

Now let's take a look at how each data node manages the data. A simple solution is to store each object in a stand-alone file. This works, but the performance suffers when there are many small files. Two issues arise when having too many small files on a file system. First, it wastes many data blocks. A file system stores files in discrete disk blocks. Disk blocks have the same size, and the size is fixed when the volume is initialized. The typical block size is around 4KB. For a file smaller than 4KB, it would still consume the entire disk block. If the file system holds a lot of small files, it wastes a lot of disk blocks, with each one only lightly filled with a small file.

Second, it could exceed the system's inode capacity. The file system stores the location and other information about a file in a special type of block called inode. For most file systems, the number of inodes is fixed when the disk is initialized. With millions of small files, it runs the risk of consuming all inodes. Also, the operating system does not handle a large number of inodes very well, even with aggressive caching of file system

metadata. For these reasons, storing small objects as individual files does not work well in practice.

To address these issues, we can merge many small objects into a larger file. It works conceptually like a write-ahead log (WAL). When we save an object, it is appended to an existing read-write file. When the read-write file reaches its capacity threshold (usually set to a few GBs), the read-write file is marked as read-only and a new read-write file is created to receive new objects. Once a file is marked as read-only, it can only serve read requests. Figure 9.12 explains how this process works.

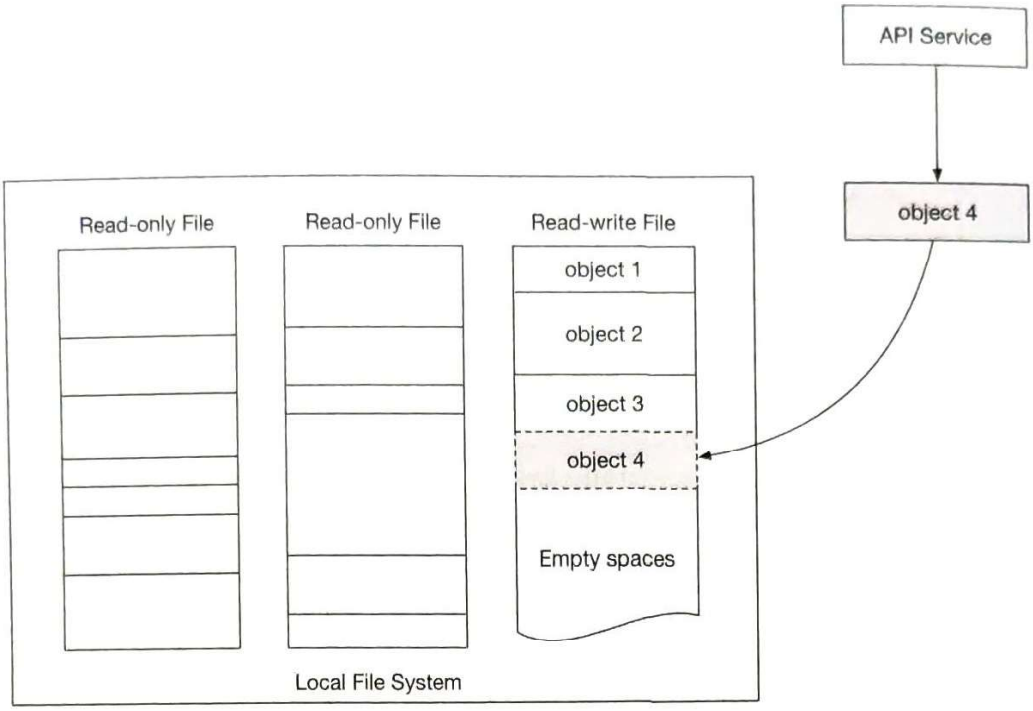


Figure 9.12: Store multiple small objects in one big file

Note that write access to the read-write file must be serialized. As shown in Figure 9.12, objects are stored in order, one after the other, in the read-write file. To maintain this on-disk layout, multiple cores processing incoming write requests in parallel must take their turns to write to the read-write file. For a modern server with a large number of cores processing many incoming requests in parallel, this seriously restricts write throughput. To fix this, we could provide dedicated read-write files, one for each core processing incoming requests.

Object lookup

With each data file holding many small objects, how does the data node locate an object by UUID? The data node needs the following information:

- The data file that contains the object
- The starting offset of the object in the data file
- The size of the object

The database schema to support this lookup is shown in Table 9.3.

object_mapping
object_id
file_name
start_offset
object_size

Table 9.3: `Object_mapping` table

Field	Description
`object_id`	UUID of the object
`file_name`	The name of the file that contains the object
`start_offset`	Beginning address of the object in the file
`object_size`	The number of bytes in the object

Table 9.4: `Object_mapping` fields

We considered two options for storing this mapping: a file-based key-value store such as RocksDB [16] or a relational database. RocksDB is based on SSTable [17], and it is fast for writes but slower for reads. A relational database usually uses a B+ tree [18] based storage engine, and it is fast for reads but slower for writes. As mentioned earlier, the data access pattern is write once and read multiple times. Since a relational database provides better read performance, it is a better choice than RocksDB.

How should we deploy this relational database? At our scale, the data volume for the mapping table is massive. Deploying a single large cluster to support all data nodes could work, but is difficult to manage. Note that this mapping data is isolated within each data node. There is no need to share this across data nodes. To take advantage of this property, we could simply deploy a simple relational database on each data node. SQLite [19] is a good choice here. It is a file-based relational database with a solid reputation.

Updated data persistence flow

Since we have made quite a few changes to the data node, let's revisit how to save a new object in the data node (Figure 9.13).

1. The API service sends a request to save a new object named `object 4`.
2. The data node service appends the object named `object 4` at the end of the read-write file named `/data/c`.
3. A new record of `object 4` is inserted into the `object_mapping` table.
4. The data node service returns the UUID to the API service.

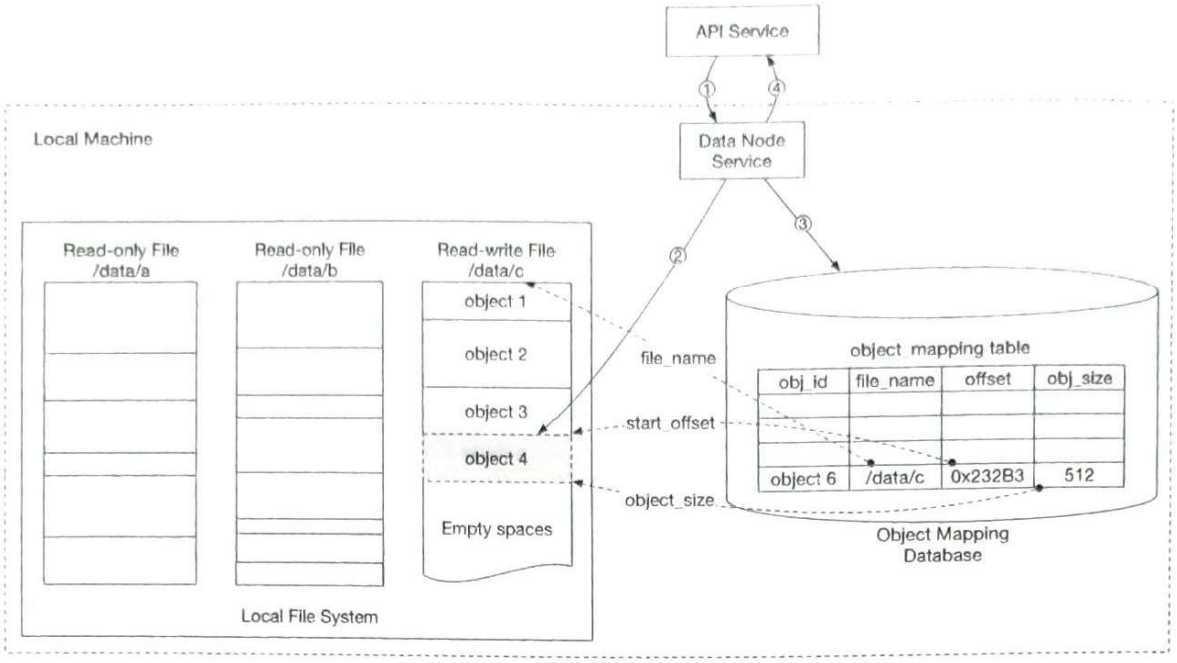


Figure 9.13: Updated data persistence flow

Durability

Data reliability is extremely important for data storage systems. How can we create a storage system that offers six nines of durability? Each failure case has to be carefully considered and data needs to be properly replicated.

Hardware failure and failure domain

Hard drive failures are inevitable no matter which media we use. Some storage media may have better durability than others, but we cannot rely on a single hard drive to achieve our durability objective. A proven way to increase durability is to replicate data to multiple hard drives, so a single disk failure does not impact the data availability, as a whole. In our design, we replicate data three times.

Let's assume the spinning hard drive has an annual failure rate of 0.81% [20]. This number highly depends on the model and make. Making 3 copies of data gives us $1 - 0.0081^3 =\sim 0.999999$ reliability. This is a very rough estimate. For more sophisticated calculations, please read [20].

For a complete durability evaluation, we also need to consider the impacts of different failure domains. A failure domain is a physical or logical section of the environment that is negatively affected when a critical service experiences problems. In a modern data center, a server is usually put into a rack [21], and the racks are grouped into rows/floors/-rooms. Since each rack shares network switches and power, all the servers in a rack are in a rack-level failure domain. A modern server shares components like the motherboard, processors, power supply, HDD drives, etc. The components in a server are in a node-level failure domain.

Here is a good example of a large-scale failure domain isolation. Typically, data cen-

ters divide infrastructure that shares nothing into different Availability Zones (AZs). We replicate our data to different AZs to minimize the failure impact (Figure 9.14). Note that the choice of failure domain level doesn't directly increase the durability of data, but it will result in better reliability in extreme cases, such as large-scale power outages, cooling system failures, natural disasters, etc.

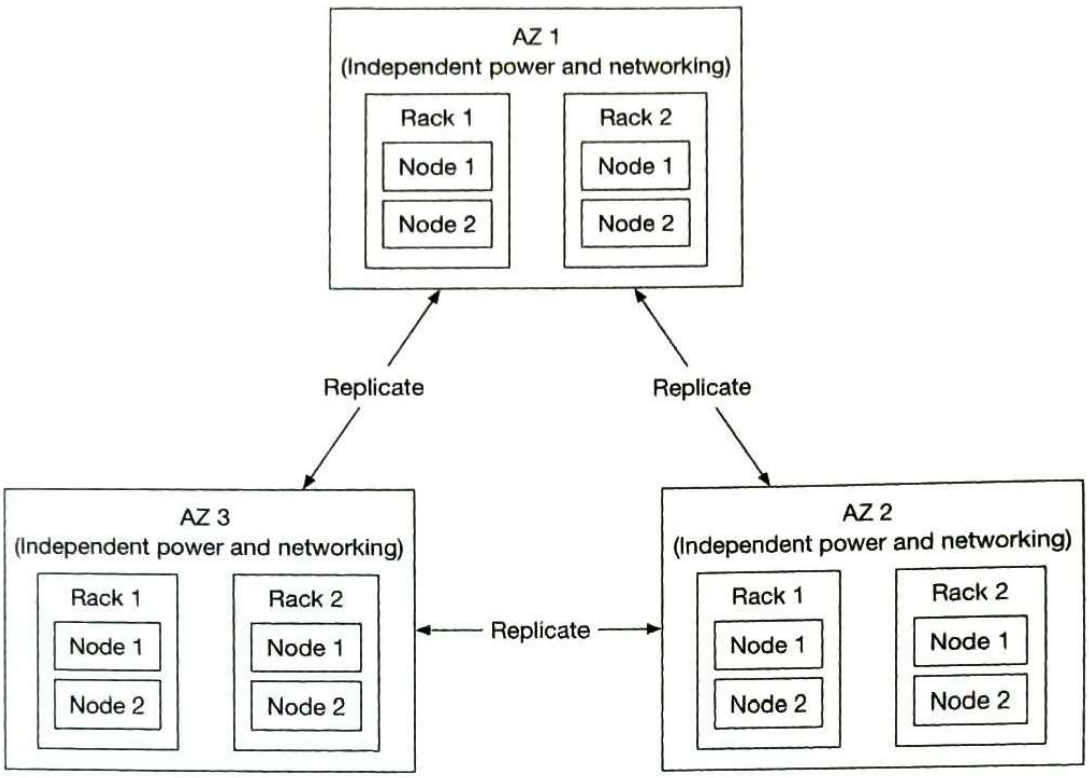


Figure 9.14: Multi-Datacenter replication

Erasure coding

Making three full copies of data gives us roughly 6 nines of data durability. Are there other options to further increase durability? Yes, erasure coding is one option. Erasure coding [22] deals with data durability differently. It chunks data into smaller pieces (placed on different servers) and creates parities for redundancy. In the event of failures, we can use chunk data and parities to reconstruct the data. Let's take a look at a concrete example ($4 + 2$ erasure coding) as shown in Figure 9.15.

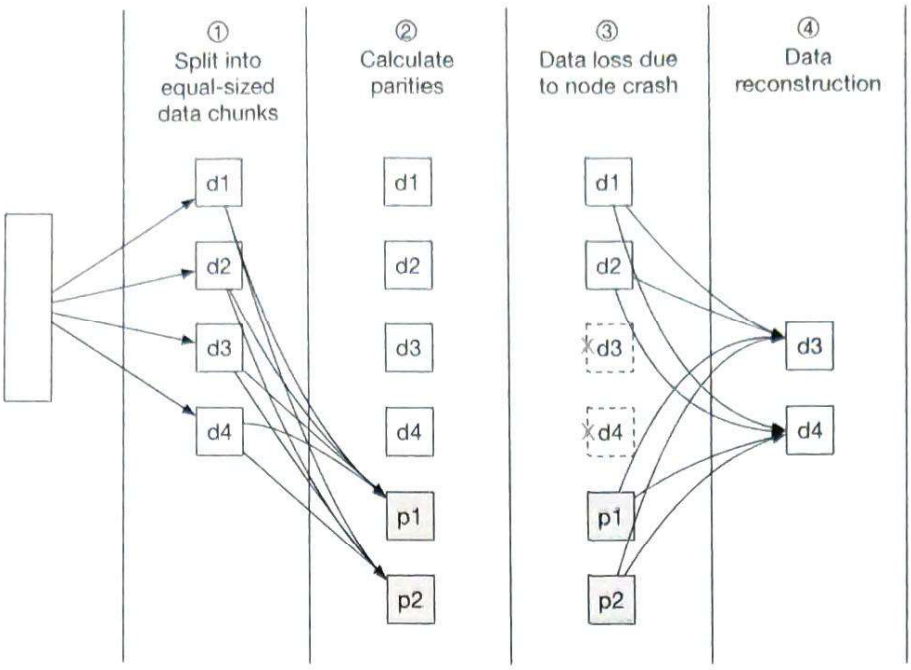


Figure 9.15: Erasure coding

1. Data is broken up into four even-sized data chunks d1, d2, d3, and d4.
2. The mathematical formula [23] is used to calculate the parities p1 and p2. To give a much simplified example, $p1 = d1 + 2 \times d2 - d3 + 4 \times d4$ and $p2 = -d1 + 5 \times d2 + d3 - 3 \times d4$ [24].
3. Data d3 and d4 are lost due to node crashes.
4. The mathematical formula is used to reconstruct lost data d3 and d4, using the known values of d1, d2, p1, and p2.

Let's take a look at another example as shown in Figure 9.16 to better understand how erasure coding works with failure domains. An (8+4) erasure coding setup breaks up the original data evenly into 8 chunks and calculates 4 parities. All 12 pieces of data have the same size. All 12 chunks of data are distributed across 12 different failure domains. The mathematics behind erasure coding ensures that the original data can be reconstructed when at most 4 nodes are down.

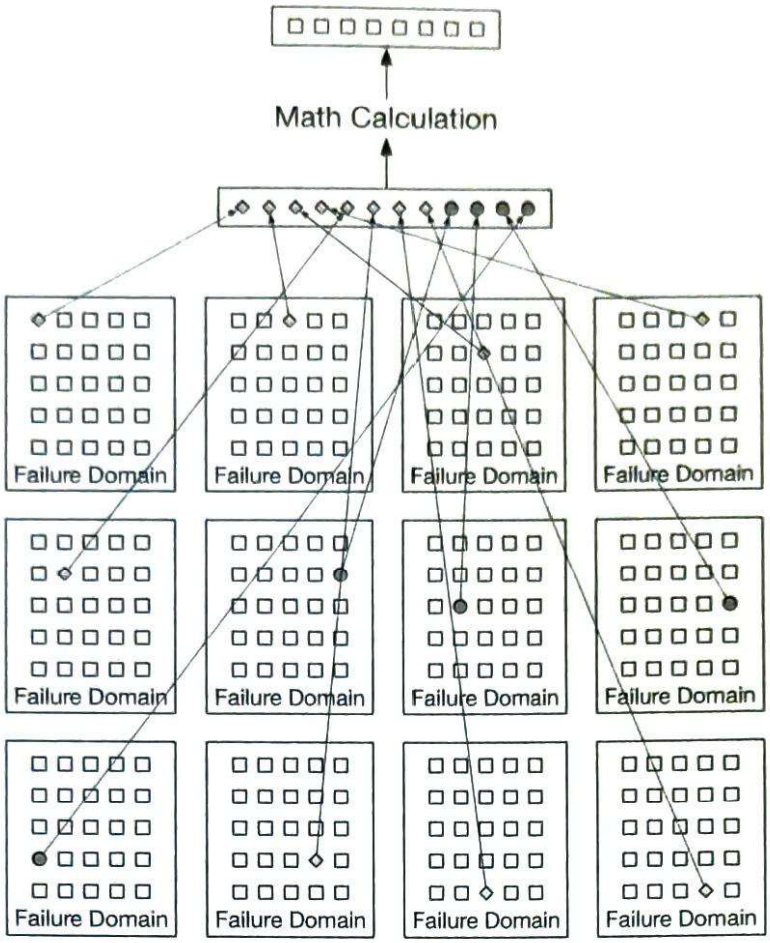


Figure 9.16: $(8 + 4)$ erasure coding

Compared to replication where the data router only needs to read data for an object from one healthy node, in erasure coding the data router has to read data from at least 8 healthy nodes. This is an architectural design tradeoff. We use a more complex solution with a slower access speed, in exchange for higher durability and lower storage cost. For object storage where the main cost is storage, this tradeoff might be worth it.

How much extra space does erasure coding need? For every two chunks of data, we need one parity block, so the storage overhead is 50% (Figure 9.17). While in 3-copy replication, the storage overhead is 200% (Figure 9.17).

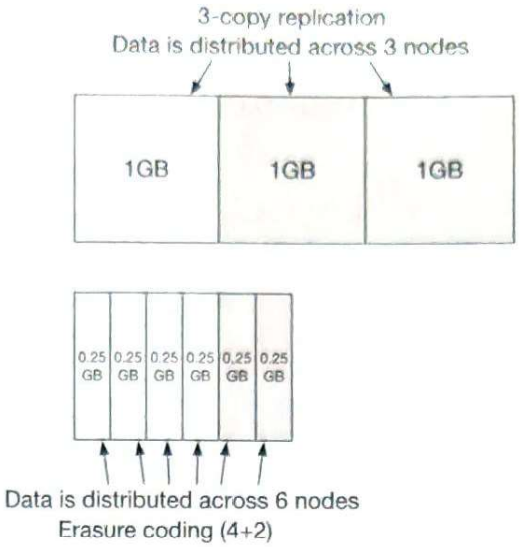


Figure 9.17: Extra space for replication and erasure coding

Does erasure coding increase data durability? Let's assume a node has a 0.81% annual failure rate. According to the calculation done by Backblaze [20], erasure coding can achieve 11 nines durability. The calculation requires complicated math. If you're interested, refer to [20] for details.

Table 9.5 compares the pros and cons of replication and erasure coding.

	Replication	**Erasure coding**
Durability	6 nines of durability (data copied 3 times)	11 nines of durability (8 + 4 erasure coding). **Erasure coding wins.**
Storage efficiency	200% storage overhead.	50% storage overhead. **Erasure coding wins.**
Compute resource	No computation. **Replication wins.**	Higher usage of computation resources to calculate parities.
Write performance	Replicating data to multiple nodes. No calculation is needed. **Replication wins.**	Increased write latency because we need to calculate parities before data is written to disk.
Read performance	In normal operation, reads are served from the same replica. Reads under a failure mode are not impacted because reads can be served from a non-fault replica. **Replication wins.**	In normal operation, every read has to come from multiple nodes in the cluster. Reads under a failure mode are slower because the missing data must be reconstructed first.

Table 9.5: Replication vs erasure coding

In summary, replication is widely adopted in latency-sensitive applications while erasure coding is often used to minimize storage cost. Erasure coding is attractive for its cost

efficiency and durability, but it greatly complicates the data node design. Therefore, for this design, we mainly focus on replication.

Correctness verification

Erasure coding increases data durability at comparable storage costs. Now we can move on to solve another hard challenge: data corruption.

If a disk fails completely and the failure can be detected, it can be treated as a data node failure. In this case, we can reconstruct data using erasure coding. However, in-memory data corruption is a regular occurrence in large-scale systems.

This problem can be addressed by verifying checksums [25] between process boundaries. A checksum is a small-sized block of data that is used to detect data errors. Figure 9.18 illustrates how the checksum is generated.

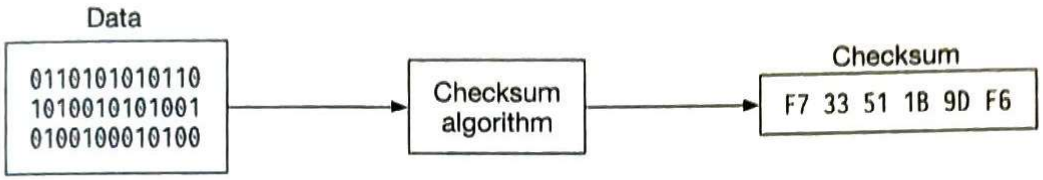


Figure 9.18: Generate checksum

If we know the checksum of the original data, we can compute the checksum of the data after transmission:

- If they are different, data is corrupted.
- If they are the same, there is a very high probability the data is not corrupted. The probability is not 100%, but in practice, we could assume they are the same.

Figure 9.19: Compare checksums

There are many checksum algorithms, such as MD5 [26], SHA1 [27], HMAC [28], etc. A good checksum algorithm usually outputs a significantly different value even for a small change made to the input. For this chapter, we choose a simple checksum algorithm such as MD5.

In our design, we append the checksum at the end of each object. Before a file is marked as read-only, we add a checksum of the entire file at the end. Figure 9.20 shows the layout.

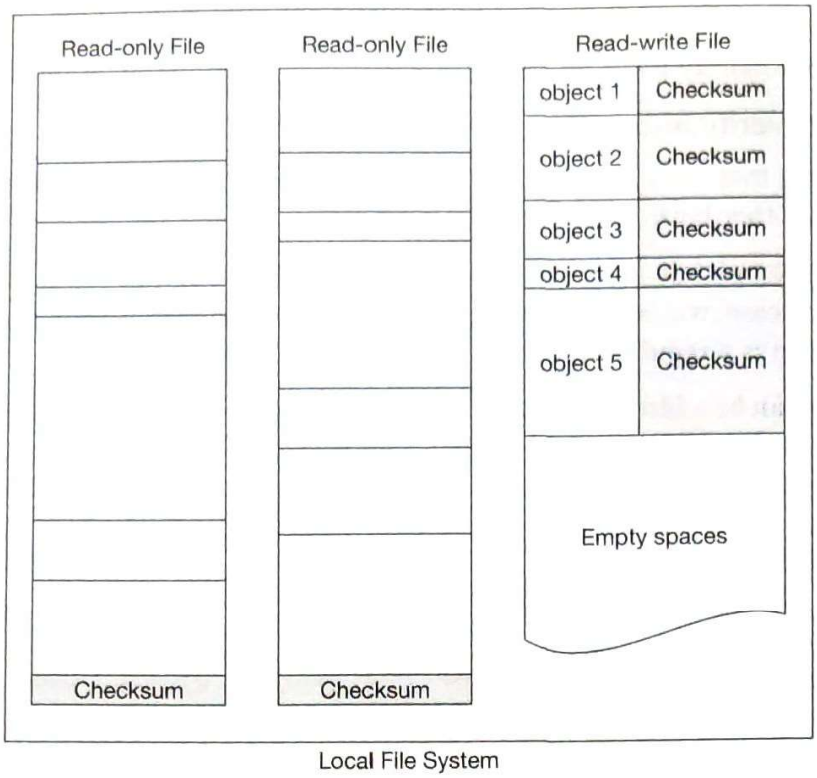


Figure 9.20: Add checksum to data node

With $(8 + 4)$ erasure coding and checksum verification, this is what happens when we read data:

1. Fetch the object data and the checksum.
2. Compute the checksum against the data received.
   (a) If the two checksums match, the data is error-free.
   (b) If the checksums are different, the data is corrupted. We will try to recover by reading the data from other failure domains.
3. Repeat steps 1 and 2 until all 8 pieces of data are returned. We then reconstruct the data and send it back to the client.

Metadata data model

In this section, we first discuss the database schema and then dive into scaling the database.

Schema

The database schema needs to support the following 3 queries:

Query 1: Find the object ID by object name.

Query 2: Insert and delete an object based on the object name.

Query 3: List objects in a bucket sharing the same prefix.

Figure 9.21 shows the schema design. We need two database tables: `bucket` and `object`.

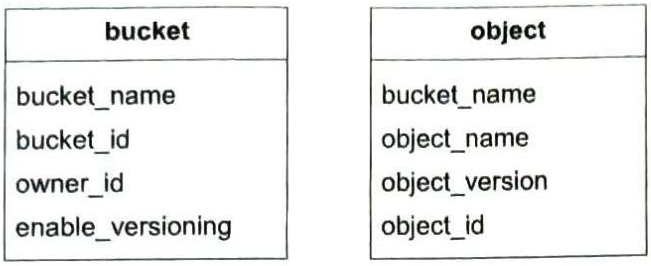


Figure 9.21: Database tables

Scale the `bucket` table

Since there is usually a limit on the number of buckets a user can create, the size of the `bucket` table is small. Let's assume we have 1 million customers, each customer owns 10 buckets and each record takes 1KB. That means we need 10GB (1 million $\times 10 \times$ 1KB) of storage space. The whole table can easily fit in a modern database server. However, a single database server might not have enough CPU or network bandwidth to handle all read requests. If so, we can spread the read load among multiple database replicas.

Scale the `object` table

The `object` table holds the object metadata. The dataset at our design scale will likely not fit in a single database instance. We can scale the object table by sharding.

One option is to shard by the `bucket_id` so all the objects under the same bucket are stored in one shard. This doesn't work because it causes hotspot shards as a bucket might contain billions of objects.

Another option is to shard by `object_id`. The benefit of this sharding scheme is that it evenly distributes the load. But we will not be able to execute query 1 and query 2 efficiently because those two queries are based on the URI.

We choose to shard by a combination of `bucket_name` and `object_name`. This is because most of the metadata operations are based on the object URI, for example, finding the object ID by URI or uploading an object via URI. To evenly distribute the data, we can use the hash of the `<bucket_name, object_name>` as the sharding key.

With this sharding scheme, it is straightforward to support the first two queries, but the last query is less obvious. Let's take a look.

Listing objects in a bucket

The object store arranges files in a flat structure instead of a hierarchy, like in a file system. An object can be accessed using a path in this format, `s3://bucket-name/object-name`. For example, `s3://mybucket/abc/d/e/f/file.txt` contains:

- Bucket name: `mybucket`
- Object name: `abc/d/e/f/file.txt`

To help users organize their objects in a bucket, S3 introduces a concept called 'prefixes'. A prefix is a string at the beginning of the object name. S3 uses prefixes to organize the data in a way similar to directories. However, prefixes are not directories. Listing a bucket by prefix limits the results to only those object names that begin with the prefix.

In the example above with the path `s3://mybucket/abc/d/e/f/file.txt`, the prefix is `abc/d/e/f/`.

The AWS S3 listing command has 3 typical uses:

1. List all buckets owned by a user. The command looks like this:

   ```
   aws s3 list-buckets
   ```

2. List all objects in a bucket that are at the same level as the specified prefix. The command looks like this:

   ```
   aws s3 ls s3://mybucket/abc/
   ```

   In this mode, objects with more slashes in the name after the prefix are rolled up into a common prefix. For example, with these objects in the bucket:

   ```
   CA/cities/losangeles.txt
   CA/cities/sanfranciso.txt
   NY/cities/ny.txt
   federal.txt
   ```

   Listing the bucket with the "/" prefix would return these results, with everything under `CA/` and `NY/` rolled up into them:

   ```
   CA/
   NY/
   federal.txt
   ```

3. Recursively list all objects in a bucket that shares the same prefix. The command looks like this:

   ```
   aws s3 ls s3://mybucket/abc/ --recursive
   ```

   Using the same example as above, listing the bucket with the `CA/` prefix would return these results:

   ```
   CA/cities/losangeles.txt
   CA/cities/sanfranciso.txt
   ```

Single database

Let's first explore how we would support the listing command with a single database. To list all buckets owned by a user, we run the following query:

```
SELECT * FROM bucket WHERE owner_id={id}
```

To list all objects in a bucket that share the same prefix, we run a query like this.

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LIKE `abc/%`
```

In this example, we find all objects with bucket_id equals to 123 that share the prefix abc/. Any objects with more slashes in their names after the specified prefix are rolled up in the application code as stated earlier in use case 2.

The same query would support the recursive listing mode, as stated in use case 3 previously. The application code would list every object sharing the same prefix, without performing any rollups.

Distributed databases

When the metadata table is sharded, it's difficult to implement the listing function because we don't know which shards contain the data. The most obvious solution is to run a search query on all shards and then aggregate the results. To achieve this, we can do the following:

1. The metadata service queries every shard by running the following query:

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LIKE `a/b/%`
```

2. The metadata service aggregates all objects returned from each shard and returns the result to the caller.

This solution works, but implementing pagination for this is a bit complicated. Before we explain why, let's review how pagination works for a simple case with a single database. To return pages of listing with 10 objects for each page, the SELECT query would start with this:

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LIKE `a/b/%`
ORDER BY object_name OFFSET 0 LIMIT 10
```

The OFFSET and LIMIT would restrict the results to the first 10 objects. In the next call, the user sends the request with a hint to the server, so it knows to construct the query for the second page with an OFFSET of 10. This hint is usually done with a cursor that the server returns with each page to the client. The offset information is encoded in the cursor. The client would include the cursor in the request for the next page. The server decodes the cursor and uses the offset information embedded in it to construct the query for the next page. To continue with the example above, the query for the second page looks like this:

```
SELECT * FROM metadata
WHERE bucket_id = "123" AND object_name LIKE `a/b/%`
ORDER BY object_name OFFSET 10 LIMIT 10
```

This client-server request loop continues until the server returns a special cursor that marks the end of the entire listing.

Now, let's explore why it's complicated to support pagination for sharded databases. Since the objects are distributed across shards, the shards would likely return a varying number of results. Some shards would contain a full page of 10 objects, while others

would be partial or empty. The application code would receive results from every shard, aggregate and sort them, and return only a page of 10 in our example. The objects that don't get included in the current round must be considered again for the next round. This means that each shard would likely have a different offset. The server must track the offsets for all the shards and associate those offsets with the cursor. If there are hundreds of shards, there will be hundreds of offsets to track.

We have a solution that can solve the problem, but there are some tradeoffs. Since object storage is tuned for vast scale and high durability, object listing performance is rarely a priority. In fact, all commercial object storage supports object listing with sub-optimal performance. To take advantage of this, we could denormalize the listing data into a separate table sharded by bucket ID. This table is only used for listing objects. With this setup, even buckets with billions of objects would offer acceptable performance. This isolates the listing query to a single database which greatly simplifies the implementation.

Object versioning

Versioning is a feature that keeps multiple versions of an object in a bucket. With versioning, we can restore objects that are accidentally deleted or overwritten. For example, we may modify a document and save it under the same name, inside the same bucket. Without versioning, the old version of the document metadata is replaced by the new version in the metadata store. The old version of the document is marked as deleted, so its storage space will be reclaimed by the garbage collector. With versioning, the object storage keeps all previous versions of the document in the metadata store, and the old versions of the document are never marked as deleted in the object store.

Figure 9.22 explains how to upload a versioned object. For this to work, we first need to enable versioning on the bucket.

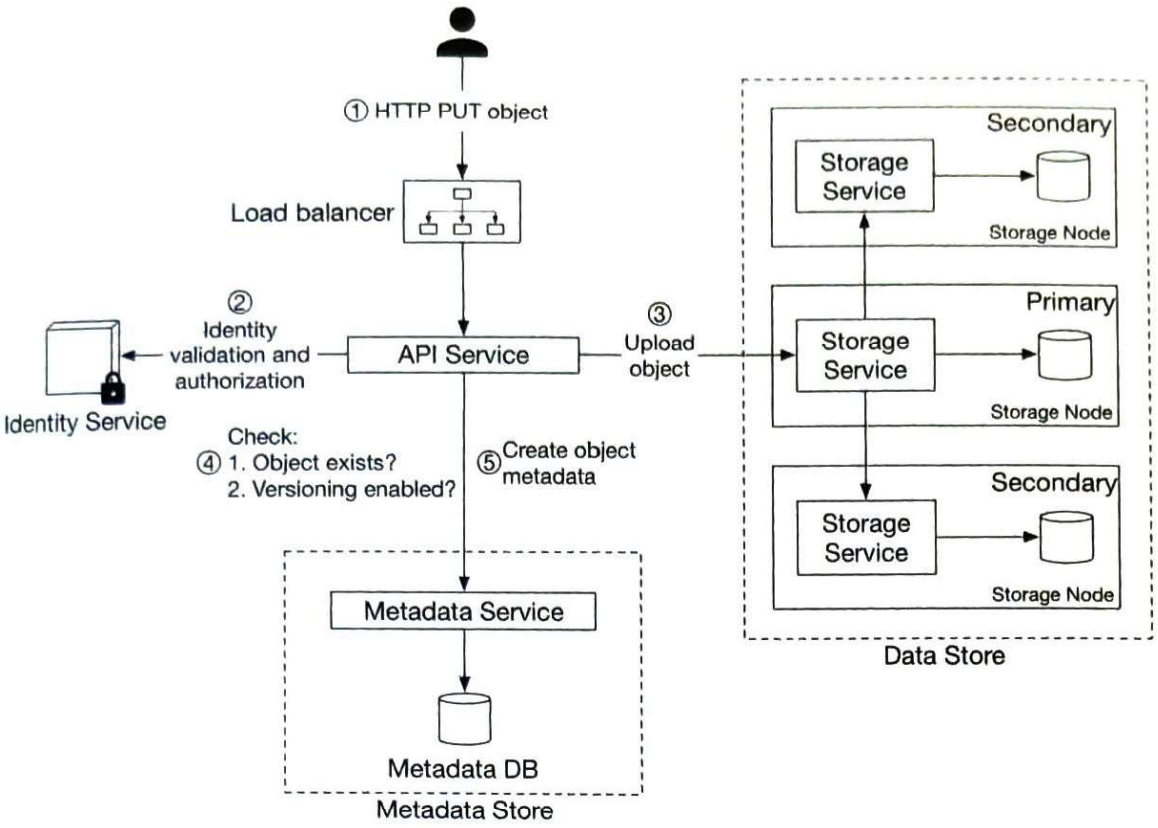


Figure 9.22: Object versioning

1. The client sends an `HTTP PUT` request to upload an object named `script.txt`.
2. The API service verifies the user's identity and ensures that the user has `WRITE` permission on the bucket.
3. Once verified, the API service uploads the data to the data store. The data store persists the data as a new object and returns a new UUID to the API service.
4. The API service calls the metadata store to store the metadata information of this object.
5. To support versioning, the object table for the metadata store has a column called `object_version` that is only used if versioning is enabled. Instead of overwriting the existing record, a new record is inserted with the same `bucket_id` and `object_name` as the old record, but with a new `object_id` and `object_version`. The `object_id` is the UUID for the new object returned in step 3. The `object_version` is a `TIMEUUID` [29] generated when the new row is inserted. No matter which database we choose for the metadata store, it should be efficient to look up the current version of an object. The current version has the largest `TIMEUUID` of all the entries with the same `object_name`. See Figure 9.23 for an illustration of how we store versioned metadata.

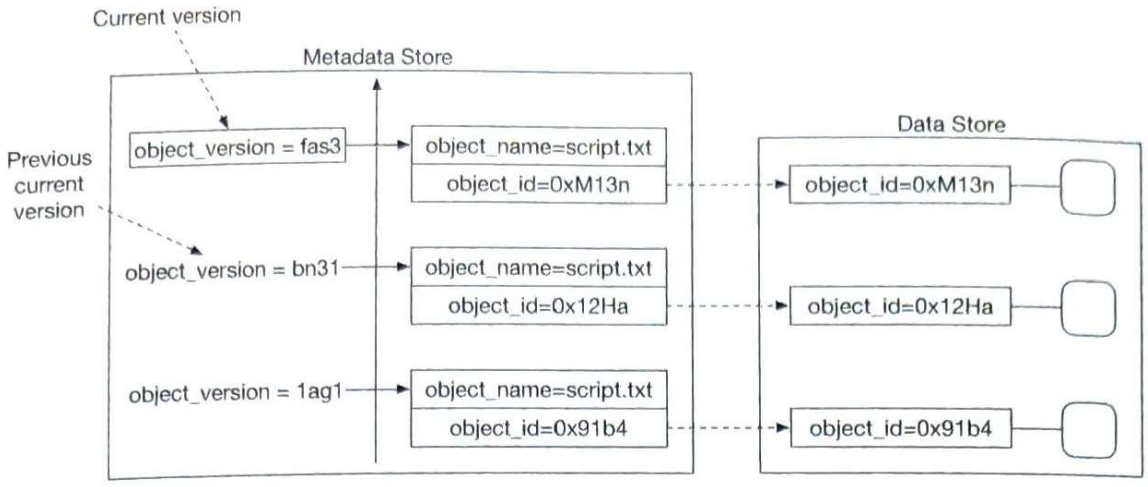


Figure 9.23: Versioned metadata

In addition to uploading a versioned object, it can also be deleted. Let's take a look.

When we delete an object, all versions remain in the bucket and we insert a delete marker, as shown in Figure 9.24.

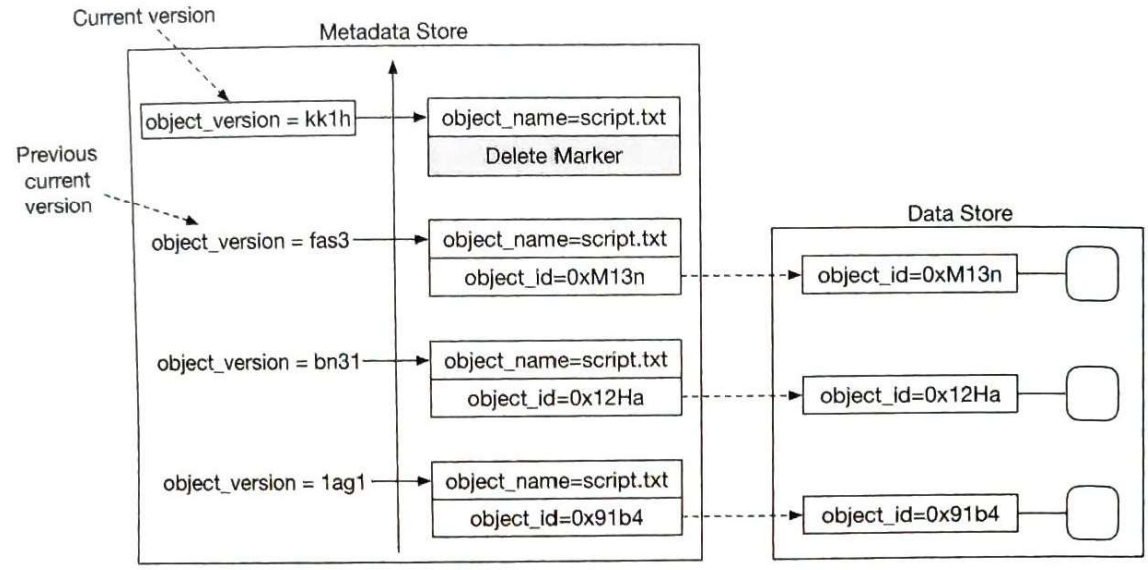


Figure 9.24: Delete object by inserting a delete marker

A delete marker is a new version of the object, and it becomes the current version of the object once inserted. Performing a `GET` request when the current version of the object is a delete marker returns a `404 Object Not Found` error.

Optimizing uploads of large files

In the back-of-the-envelope estimation, we estimated that 20% of the objects are large. Some might be larger than a few GBs. It is possible to upload such a large object file directly, but it could take a long time. If the network connection fails in the middle of the upload, we have to start over. A better solution is to slice a large object into smaller

parts and upload them independently. After all the parts are uploaded, the object store re-assembles the object from the parts. This process is called multipart upload.

Figure 9.25 illustrates how multipart upload works:

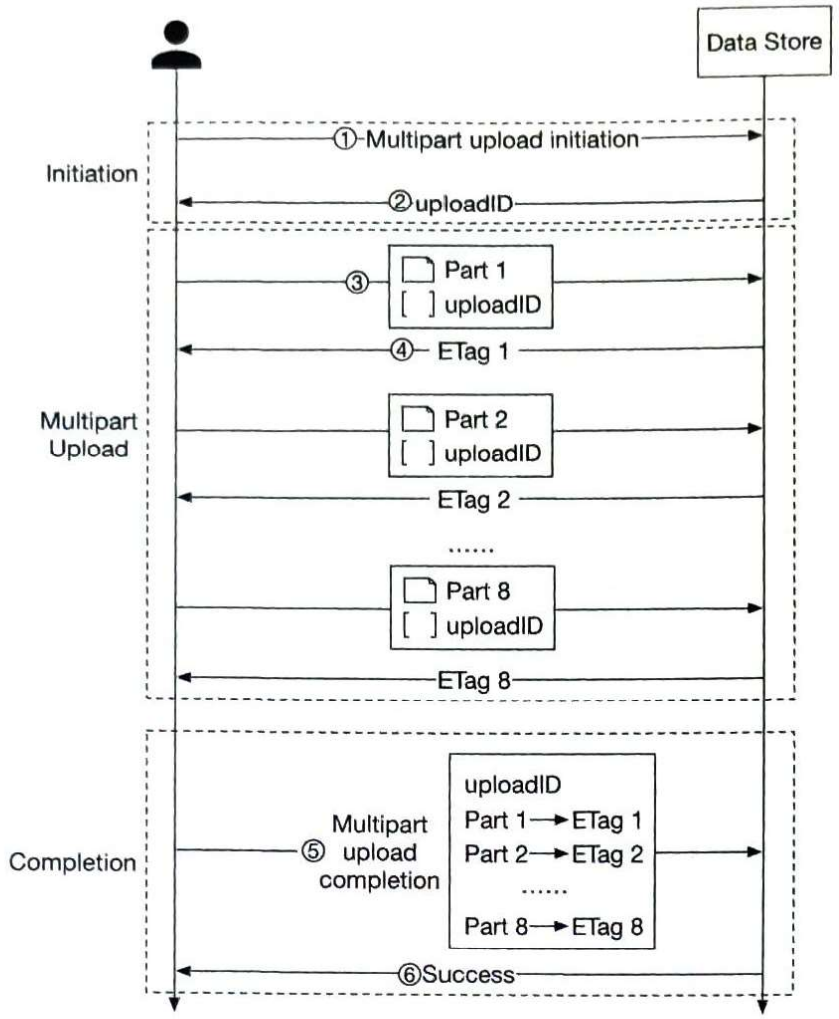


Figure 9.25: Multipart upload

1. The client calls the object storage to initiate a multipart upload.
2. The data store returns an `uploadID`, which uniquely identifies the upload.
3. The client splits the large file into small objects and starts uploading. Let's assume the size of the file is 1.6GB and the client splits it into 8 parts, so each part is 200MB in size. The client uploads the first part to the data store together with the uploadID it received in step 2.
4. When a part is uploaded, the data store returns an `ETag`, which is essentially the md5 checksum of that part. It is used to verify multipart uploads.
5. After all parts are uploaded, the client sends a complete multipart upload request, which includes the `uploadID`, part numbers, and `ETags`.
6. The data store reassembles the object from its parts based on the part number. Since

the object is really large, this process may take a few minutes. After reassembly is complete, it returns a success message to the client.

One potential problem with this approach is that old parts are no longer useful after the object has been reassembled from them. To solve this problem, we can introduce a garbage collection service responsible for freeing up space from parts that are no longer needed.

Garbage collection

Garbage collection is the process of automatically reclaiming storage space that is no longer used. There are a few ways that data might become garbage:

- Lazy object deletion. An object is marked as deleted at delete time without actually being deleted.
- Orphan data. For example, half uploaded data or abandoned multipart uploads.
- Corrupted data. Data that failed the checksum verification.

The garbage collector does not remove objects from the data store, right away. Deleted objects will be periodically cleaned up with a compaction mechanism.

The garbage collector is also responsible for reclaiming unused space in replicas. For replication, we delete the object from both primary and backup nodes. For erasure coding, if we use $(8 + 4)$ setup, we delete the object from all 12 nodes.

Figure 9.26 shows an example of how compaction works.

1. The garbage collector copies objects from `/data/b` to a new file named `/data/d`. Note the garbage collector skips "Object 2" and "Object 5" because the delete flag is set to true for both of them.
2. After all objects are copied, the garbage collector updates the `object_mapping` table. For example, the `obj_id` and `object_size` fields of "Object 3" remain the same, but `file_name` and `start_offset` are updated to reflect its new location. To ensure data consistency, it's a good idea to wrap the update operations to `file_name` and `start_offset` in a database transaction.

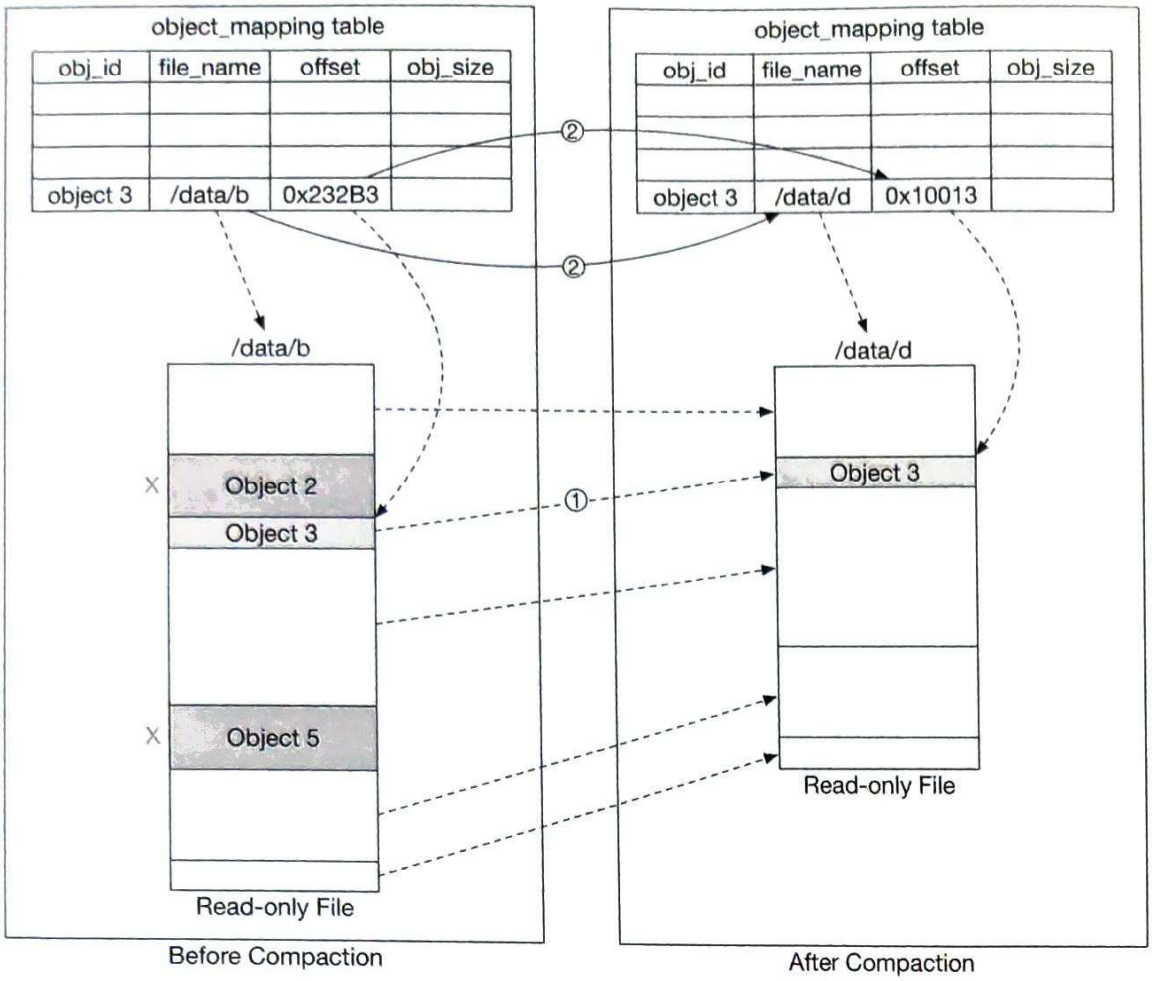


Figure 9.26: Compaction

As we can see from Figure 9.26, the size of the new file after compaction is smaller than the old file. To avoid creating a lot of small files, the garbage collector usually waits until there are a large number of read-only files to compact, and the compaction process appends objects from many read-only files into a few large new files.

Step 4 - Wrap Up

In this chapter, we described the high-level design for S3-like object storage. We compared the differences between block storage, file storage, and object storage.

The focus of this interview is on the design of object storage, so we listed how the uploading, downloading, listing objects in a bucket, and versioning of objects are typically done in object storage.

Then we dived deeper into the design. Object storage is composed of a data store and a metadata store. We explained how the data is persisted into the data store and discussed two methods for increasing reliability and durability: replication and erasure coding. For the metadata store, we explained how the multipart upload is executed and how to design the database schema to support typical use cases. Lastly, we explained how to shard the

Reference Material

[1] Fibre channel. https://en.wikipedia.org/wiki/Fibre_Channel.

[2] iSCSI. https://en.wikipedia.org/wiki/ISCSI.

[3] Server Message Block. https://en.wikipedia.org/wiki/Server_Message_Block.

[4] Network File System. https://en.wikipedia.org/wiki/Network_File_System.

[5] Amazon S3 Strong Consistency. https://aws.amazon.com/s3/consistency/.

[6] Serial Attached SCSI. https://en.wikipedia.org/wiki/Serial_Attached_SCSI.

[7] AWS CLI ls command. https://docs.aws.amazon.com/cli/latest/reference/s3/ls.html.

[8] Amazon S3 Service Level Agreement. https://aws.amazon.com/s3/sla/.

[9] Ambry. LinkedIn'sScalableGeo-DistributedObjectStore:https://assured-cloud-computing.illinois.edu/files/2014/03/Ambry-LinkedIns-Scalable-GeoDistributed-Object-Store.pdf.

[10] inode. https://en.wikipedia.org/wiki/Inode.

[11] Ceph's Rados Gateway. https://docs.ceph.com/en/pacific/radosgw/index.html.

[12] grpc. https://grpc.io/.

[13] Paxos. https://en.wikipedia.org/wiki/Paxos_(computer_science).

[14] Raft. https://raft.github.io/.

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

[16] RocksDB. https://github.com/facebook/rocksdb.

[17] SSTable. https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/.

[18] B+ tree. https://en.wikipedia.org/wiki/B%2B_tree.

[19] SQLite. https://www.sqlite.org/index.html.

[20] Data Durability Calculation. https://www.backblaze.com/blog/cloud-storage-durability/.

[21] Rack. https://en.wikipedia.org/wiki/19-inch_rack.

[22] Erasure Coding. https://en.wikipedia.org/wiki/Erasure_code.

[23] Reed–Solomon error correction. https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction.

[24] Erasure Coding Demystified. https://www.youtube.com/watch?v=Q5kVuM7zEUI.

[25] Checksum. https://en.wikipedia.org/wiki/Checksum.

[26] Md5. https://en.wikipedia.org/wiki/MD5.

[27] Sha1. https://en.wikipedia.org/wiki/SHA-1.

[28] Hmac. https://en.wikipedia.org/wiki/HMAC.

[29] TIMEUUID. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/timeuuid_functions_r.html.

10 Real-time Gaming Leaderboard

In this chapter, we are going to walk through the challenge of designing a leaderboard for an online mobile game.

What is a leaderboard? Leaderboards are common in gaming and elsewhere to show who is leading a particular tournament or competition. Users are assigned points for completing tasks or challenges, and whoever has the most points is at the top of the leaderboard. Figure 10.1 shows an example of a mobile game leaderboard. The leaderboard shows the ranking of the leading competitors and also displays the position of the user on it.

Figure 10.1: Leaderboard

Step 1 - Understand the Problem and Establish Design Scope

Leaderboards can be pretty straightforward, but there are a number of different matters that can add complexity. We should clarify the requirements.

Candidate: How is the score calculated for the leaderboard?
Interviewer: The user gets a point when they win a match. We can go with a simple point system in which each user has a score associated with them. Each time the user wins a match, we should add a point to their total score.

Candidate: Are all players included in the leaderboard?
Interviewer: Yes.

Candidate: Is there a time segment associated with the leaderboard?
Interviewer: Each month, a new tournament kicks off which starts a new leaderboard.

Candidate: Can we assume we only care about the top 10 users?
Interviewer: We want to display the top 10 users as well as the position of a specific user on the leaderboard. If time allows, let's also discuss how to return users who are four places above and below a specific user.

Candidate: How many users are in a tournament?
Interviewer: Average of 5 million daily active users (DAU) and 25 million monthly active users (MAU).

Candidate: How many matches are played on average during a tournament?
Interviewer: Each player plays 10 matches per day on average.

Candidate: How do we determine the rank if two players have the same score?
Interviewer: In this case, their ranks are the same. If time allows, we can talk about ways to break ties.

Candidate: Does the leaderboard need to be real-time?
Interviewer: Yes, we want to present real-time results, or as close as possible. It is not okay to present a batched history of results.

Now that we've gathered all the requirements, let's list the functional requirements.

- Display top 10 players on the leaderboard.
- Show a user's specific rank.
- Display players who are four places above and below the desired user (bonus).

Other than clarifying functional requirements, it's important to understand non-functional requirements.

Non-functional requirements

- Real-time update on scores.
- Score update is reflected on the leaderboard in real-time.
- General scalability, availability, and reliability requirements.

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.

With 5 million DAU, if the game had an even distribution of players during a 24-hour period, we would have an average of 50 users per second $\left(\frac{5{,}000{,}000 \text{ DAU}}{10^5 \text{ seconds}} =\sim 50\right)$. However, we know that usages most likely aren't evenly distributed, and potentially there are peaks during evenings when many people across different time zones have time to play. To account for this, we could assume that peak load would be 5 times the average.

Therefore we'd want to allow for a peak load of 250 users per second.

QPS for users scoring a point: if a user plays 10 games per day on average, the QPS for users scoring a point is: $50 \times 10 = \sim 500$. Peak QPS is 5x of the average: $500 \times 5 = 2{,}500$.

QPS for fetching the top 10 leaderboard: assume a user opens the game once a day and the top 10 leaderboard is loaded only when a user first opens the game. The QPS for this is around 50.

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

In this section, we will discuss API design, high-level architecture, and data models.

API design

At a high level, we need the following three APIs:

`POST /v1/scores`

Update a user's position on the leaderboard when a user wins a game. The request parameters are listed below. This should be an internal API that can only be called by the game servers. The client should not be able to update the leaderboard score directly.

Field	Description
`user_id`	The user who wins a game.
`points`	The number of points a user gained by winning a game.

Table 10.1: Request parameters

Response:

Name	Description
`200 OK`	Successfully updated a user's score.
`400 Bad Request`	Failed to update a user's score.

Table 10.2: Response

`GET /v1/scores`

Fetch the top 10 players from the leaderboard.

Sample response:

```
{
  "data": [
  {
    "user_id": "user_id1",
    "user_name": "alice",
    "rank": 1,
    "score": 976
  },
  {
    "user_id": "user_id2",
    "user_name": "bob",
    "rank": 2,
    "score": 965
  }
  ],
  ...
  "total": 10
}
```

GET /v1/scores/{:user_id}

Fetch the rank of a specific user.

Field	Description
user_id	The ID of the user whose rank we would like to fetch.

Table 10.3: Request parameters

Sample response:

```
{
  "user_info": {
    "user_id": "user5",
    "score": 940,
    "rank": 6,
  }
}
```

High-level architecture

The high-level design diagram is shown in Figure 10.2. There are two services in this design. The game service allows users to play the game and the leaderboard service creates and displays a leaderboard.

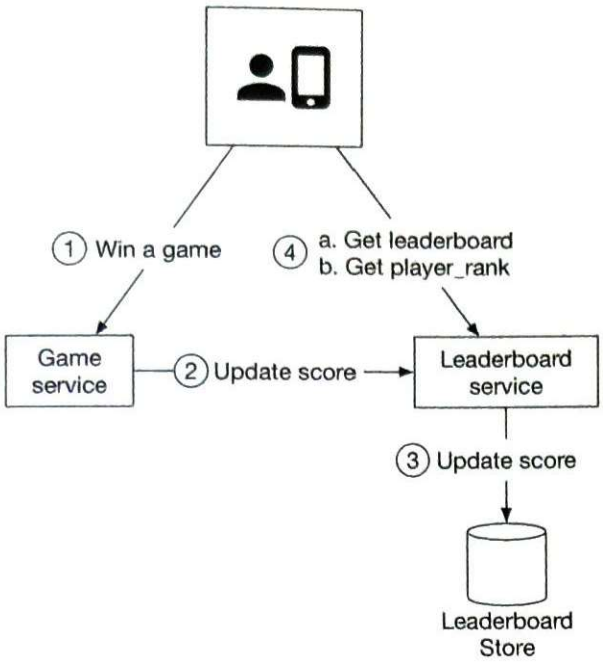


Figure 10.2: High-level design

1. When a player wins a game, the client sends a request to the game service.
2. The game service ensures the win is valid and calls the leaderboard service to update the score.
3. The leaderboard service updates the user's score in the leaderboard store.
4. A player makes a call to the leaderboard service directly to fetch leaderboard data, including:
   (a) top 10 leaderboard.
   (b) the rank of the player on the leaderboard.

Before settling on this design, we considered a few alternatives and decided against them. It might be helpful to go through the thought process of this and to compare different options.

Should the client talk to the leaderboard service directly?

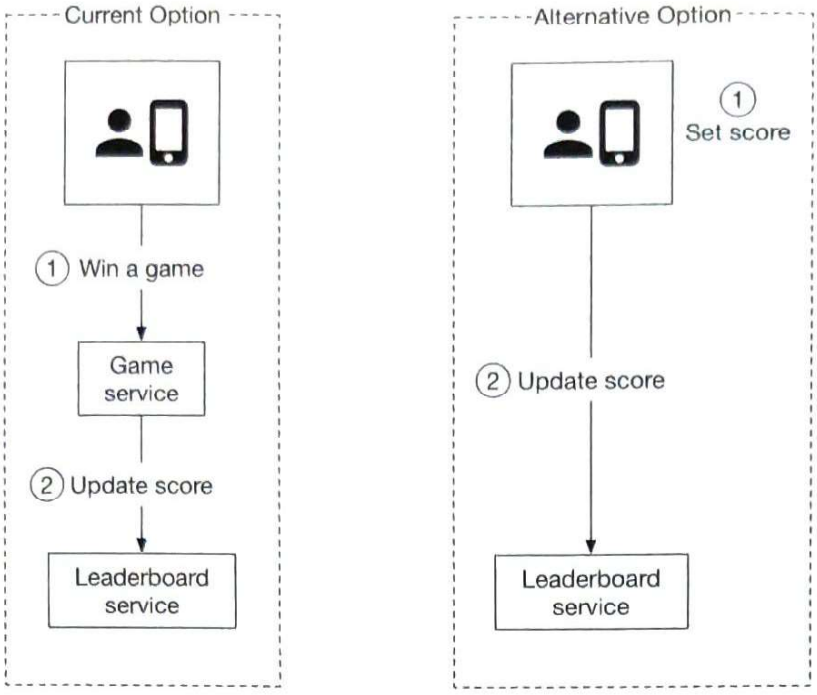


Figure 10.3: Who sets the leaderboard score

In the alternative design, the score is set by the client. This option is not secure because it is subject to man-in-the-middle attack [1], where players can put in a proxy and change scores at will. Therefore, we need the score to be set on the server-side.

Note that for server authoritative games such as online poker, the client may not need to call the game server explicitly to set scores. The game server handles all game logic, and it knows when the game finishes and could set the score without any client intervention.

Do we need a message queue between the game service and the leaderboard service?

The answer to this question highly depends on how the game scores are used. If the data is used in other places or supports multiple functionalities, then it might make sense to put data in Kafka as shown in Figure 10.4. This way, the same data can be consumed by multiple consumers, such as leaderboard service, analytics service, push notification service, etc. This is especially true when the game is a turn-based or multi-player game in which we need to notify other players about the score update. As this is not an explicit requirement based on the conversation with the interviewer, we do not use a message queue in our design.

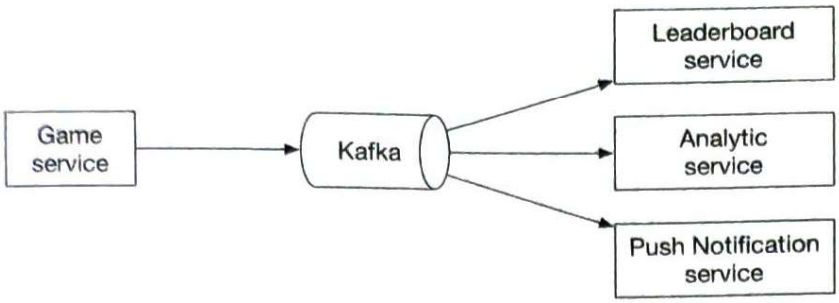


Figure 10.4: Game scores are used by multiple services

Data models

One of the key components in the system is the leaderboard store. We will discuss three potential solutions: relational database, Redis, and NoSQL (NoSQL solution is explained in deep dive section on page 309).

Relational database solution

First, let's take a step back and start with the simplest solution. What if the scale doesn't matter and we have only a few users?

We would most likely opt to have a simple leaderboard solution using a relational database system (RDS). Each monthly leaderboard could be represented as a database table containing user id and score columns. When the user wins a match, either award the user 1 point if they are new, or increase their existing score by 1 point. To determine a user's ranking on the leaderboard, we would sort the table by the score in descending order. The details are explained below.

Leaderboard DB table:

leaderboard	
user_id	varchar
score	int

Figure 10.5: Leaderboard table

In reality, the leaderboard table has additional information, such as a `game_id`, a timestamp, etc. However, the underlying logic of how to query and update the leaderboard remains the same. For simplicity, we assume only the current month's leaderboard data is stored in the leaderboard table.

A user wins a point:

Figure 10.6: A user wins a point

Assume every score update would be an increment of 1. If a user doesn't yet have an entry in the leaderboard for the month, the first insert would be:

```
INSERT INTO leaderboard (user_id, score) VALUES ('mary1934', 1)
  ;
```

An update to the user's score would be:

```
UPDATE leaderboard set score=score + 1 where user_id='mary1934';
```

Find a user's leaderboard position:

Figure 10.7: Find a user's leaderboard position

To fetch the user rank, we would sort the leaderboard table and rank by the score:

```
SELECT (@rownum := @rownum + 1) AS rank, user_id, score
FROM leaderboard
ORDER BY score DESC;
```

The result of the SQL query looks like this:

rank	user_id	score
1	happy_tomato	987
2	mallow	902
3	smith	870
4	mary1934	850

Table 10.4: Result sorted by score

This solution works when the data set is small, but the query becomes very slow when there are millions of rows. Let's take a look at why.

To figure out the rank of a user, we need to sort every single player into their correct spot on the leaderboard so we can determine exactly what the correct rank is. Remember that there can be duplicate scores as well, so the rank isn't just the position of the user in the list.

SQL databases are not performant when we have to process large amounts of continuously changing information. Attempting to do a rank operation over millions of rows is going to take 10s of seconds, which is not acceptable for the desired real-time approach. Since the data is constantly changing, it is also not feasible to consider a cache.

A relational database is not designed to handle the high load of read queries this implementation would require. An RDS could be used successfully if done as a batch operation, but that would not align with the requirement to return a real-time position for the user

on the leaderboard.

One optimization we can do is to add an index and limit the number of pages to scan with the LIMIT clause. The query looks like this:

```
SELECT (@rownum := @rownum + 1) AS rank, user_id, score
FROM leaderboard
ORDER BY score DESC
LIMIT 10
```

However, this approach doesn't scale well. First, finding a user's rank is not performant because it essentially requires a table scan to determine the rank. Second, this approach doesn't provide a straightforward solution for determining the rank of a user who is not at the top of the leaderboard.

Redis solution

We want to find a solution that gives us predictable performance even for millions of users and allows us to have easy access to common leaderboard operations, without needing to fall back on complex DB queries.

Redis provides a potential solution to our problem. Redis is an in-memory data store supporting key-value pairs. Since it works in memory, it allows for fast reads and writes. Redis has a specific data type called **sorted sets** that are ideal for solving leaderboard system design problems.

What are sorted sets?

A sorted set is a data type similar to a set. Each member of a sorted set is associated with a score. The members of a set must be unique, but scores may repeat. The score is used to rank the sorted set in ascending order.

Our leaderboard use case maps perfectly to sorted sets. Internally, a sorted set is implemented by two data structures: a hash table and a skip list [2]. The hash table maps users to scores and the skip list maps scores to users. In sorted sets, users are sorted by scores. A good way to understand a sorted set is to picture it as a table with score and member columns as shown in Figure 10.8. The table is sorted by score in descending order.

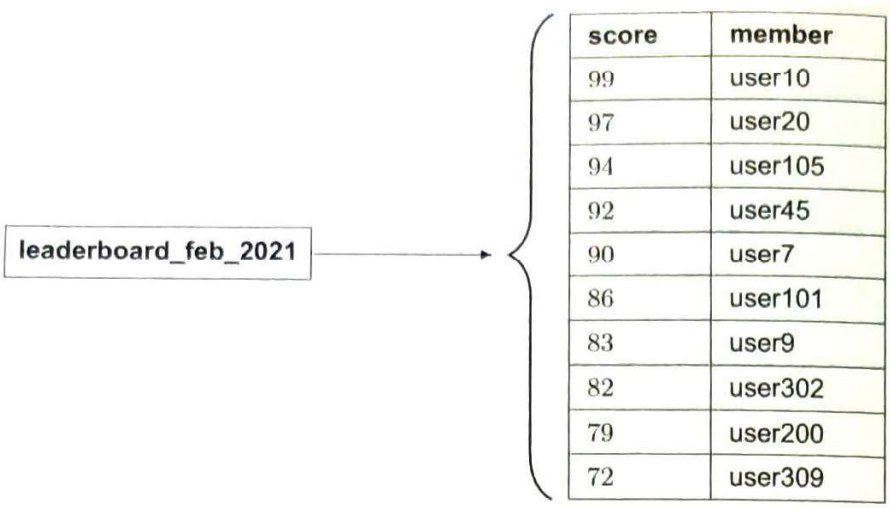


Figure 10.8: February leaderboard is represented by the sorted set

In this chapter, we don't go into the full detail of the sorted set implementation, but we do go over the high-level ideas.

A skip list is a list structure that allows for fast search. It consists of a base sorted linked list and multi-level indexes. Let's take a look at an example. In Figure 10.9, the base list is a sorted singly-linked list. The time complexity of insertion, removal, and search operations is $O(n)$.

How can we make those operations faster? One idea is to get to the middle quickly, as the binary search algorithm does. To achieve that, we add a level 1 index that skips every other node, and then a level 2 index that skips every other node of the level 1 indexes. We keep introducing additional levels, with each new level skipping every other nodes of the previous level. We stop this addition when the distance between nodes is $\frac{n}{2} - 1$, where n is the total number of nodes. As shown in Figure 10.9, searching for number 45 is a lot faster when we have multi-level indexes.

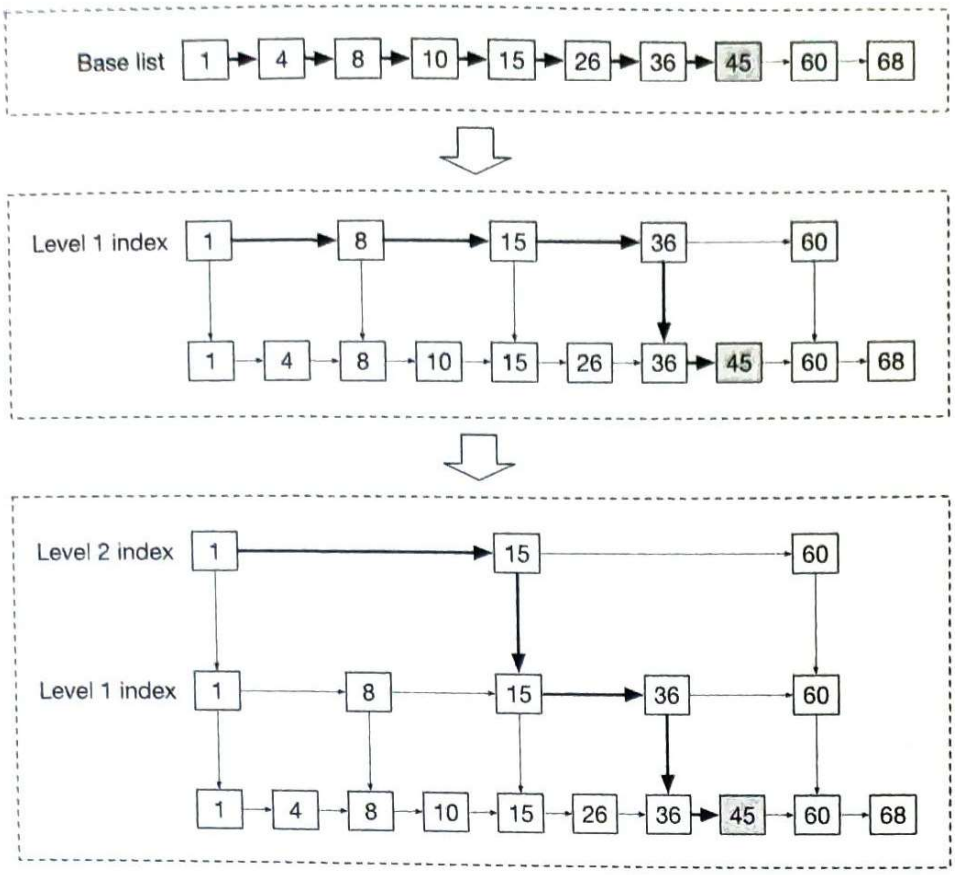


Figure 10.9: Skip list

When the data set is small, the speed improvement using the skip list isn't obvious. Figure 10.10 shows an example of a skip list with 5 levels of indexes. In the base linked list, it needs to travel 62 nodes to reach the correct node. In the skip list, it only needs to traverse 11 nodes [3].

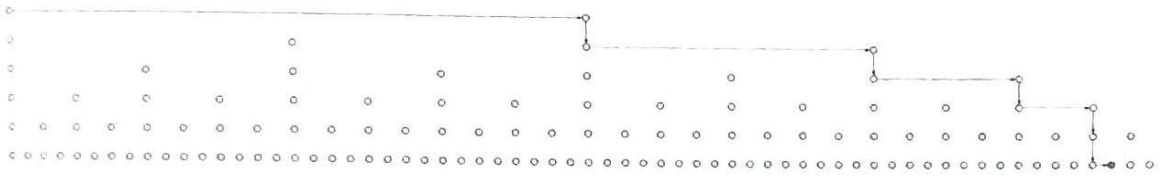


Figure 10.10: Skip list with 5 levels of indexes

Sorted sets are more performant than a relational database because each element is automatically positioned in the right order during insert or update, as well as the fact that the complexity of an add or find operation in a sorted set is logarithmic: $O(\log(n))$.

In contrast, to calculate the rank of a specific user in a relational database, we need to run nested queries:

```
SELECT *,(SELECT COUNT(*) FROM leaderboard lb2
WHERE lb2.score >= lb1.score) RANK
FROM leaderboard lb1
WHERE lb1.user_id = {:user_id};
```

Implementation using Redis sorted sets

Now that we know sorted sets are fast, let's take a look at the Redis operations we will use to build our leaderboard [4] [5] [6] [7]:

- `ZADD`: insert the user into the set if they don't yet exist. Otherwise, update the score for the user. It takes $O(\log(n))$ to execute.
- `ZINCRBY`: increment the score of the user by the specified increment. If the user doesn't exist in the set, then it assumes the score starts at 0. It takes $O(\log(n))$ to execute.
- `ZRANGE/ZREVRANGE`: fetch a range of users sorted by the score. We can specify the order (range vs. revrange), the number of entries, and the position to start from. This takes $O(\log(n)+m)$ to execute, where m is the number of entries to fetch (which is usually small in our case), and n is the number of entries in the sorted set.
- `ZRANK/ZREVRANK`: fetch the position of any user sorting in ascending/descending order in logarithmic time.

Workflow with sorted sets

1. A user scores a point

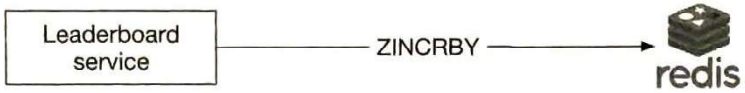


Figure 10.11: A user scores a point

Every month we create a new leaderboard sorted set and the previous ones are moved to historical data storage. When a user wins a match, they score 1 point; so we call `ZINCRBY` to increment the user's score by 1 in that month's leaderboard, or add the user to the leaderboard set if they weren't already there. The syntax for `ZINCRBY` is:

```
ZINCRBY <key> <increment> <user>
```

The following command adds a point to user `mary1934` after they win a match.

```
ZINCRBY leaderboard_feb_2021 1 'mary1934'
```

2. A user fetches the top 10 global leaderboard

Figure 10.12: Fetch top 10 global leaderboard

We will call ZREVRANGE to obtain the members in descending order because we want the highest scores, and pass the WITHSCORES attribute to ensure that it also returns the total score for each user, as well as the set of users with the highest scores. The following command fetches the top 10 players on the Feb-2021 leaderboard.

```
ZREVRANGE leaderboard_feb_2021 0 9 WITHSCORES
```

This returns a list like this:

```
[(user2,score2),(user1,score1),(user5,score5)...]
```

3. A user wants to fetch their leaderboard position

Figure 10.13: Fetch a user's leaderboard position

To fetch the position of a user in the leaderboard, we will call ZREVRANK to retrieve their rank on the leaderboard. Again, we call the rev version of the command because we want to rank scores from high to low.

```
ZREVRANK leaderboard_feb_2021 'mary1934'
```

4. Fetch the relative position in the leaderboard for a user. An example is shown in Figure 10.14.

Rank	Player	Points
267	Aquaboys	876
258	B team	845
259	Berlin's Angels	832
360	GrendelTeam	799
361	Mallow007	785
362	Woo78	743
363	milan~114	732
364	G3^^^^2	726
365	Mailso_91_	712

Figure 10.14: Fetch 4 players above and below

While not an explicit requirement, we can easily fetch the relative position for a user by leveraging `ZREVRANGE` with the number of results above and below the desired player. For example, if user `Mallow007`'s rank is 361 and we want to fetch 4 players above and below them, we would run the following command.

```
ZREVRANGE leaderboard_feb_2021 357 365
```

Storage requirement

At a minimum, we need to store the user id and score. The worst-case scenario is that all 25 million monthly active users have won at least one game, and they all have entries in the leaderboard for the month. Assuming the id is a 24-character string and the score is a 16-bit integer (or 2 bytes), we need 26 bytes of storage per leaderboard entry. Given the worst-case scenario of one leaderboard entry per MAU, we would need 26 bytes $\times 25$ million $= 650$ million bytes or $\sim$ 650MB for leaderboard storage in the Redis cache. Even if we double the memory usage to account for the overhead of the skip list and the hash for the sorted set, one modern Redis server is more than enough to hold the data.

Another related factor to consider is CPU and I/O usage. Our peak QPS from the back-of-the-envelope estimation is 2500 updates/sec. This is well within the performance envelope of a single Redis server.

One concern about the Redis cache is persistence, as a Redis node might fail. Luckily, Redis does support persistence, but restarting a large Redis instance from disk is slow. Usually, Redis is configured with a read replica, and when the main instance fails, the read replica is promoted, and a new read replica is attached.

Besides, we need to have 2 supporting tables (user and point) in a relational database like MySQL. The user table would store the user ID and user's display name (in a real-world application, this would contain a lot more data). The point table would contain the user id, score, and timestamp when they won a game. This can be leveraged for other game functions such as play history, and can also be used to recreate the Redis leaderboard in the event of an infrastructure failure.

As a small performance optimization, it may make sense to create an additional cache of the user details, potentially for the top 10 players since they are retrieved most frequently. However, this doesn't amount to a large amount of data.

Step 3 - Design Deep Dive

Now that we've discussed the high-level design, let's dive into the following:

- Whether or not to use a cloud provider
  - Manage our own services
  - Leverage cloud service providers like Amazon Web Services (AWS)
- Scaling Redis
- Alternative solution: NoSQL
- Other considerations

To use a cloud provider or not

Depending on the existing infrastructure, we generally have two options for deploying our solution. Let's take a look at each of them.

Manage our own services

In this approach, we will create a leaderboard sorted set each month to store the leaderboard data for that period. The sorted set stores member and score information. The rest of the details about the user, such as their name and profile image, are stored in MySQL databases. When fetching the leaderboard, besides the leaderboard data, API servers also query the database to fetch corresponding users' names and profile images to display on the leaderboard. If this becomes too inefficient in the long term, we can leverage a user profile cache to store users' details for the top 10 players. The design is shown in Figure 10.15.

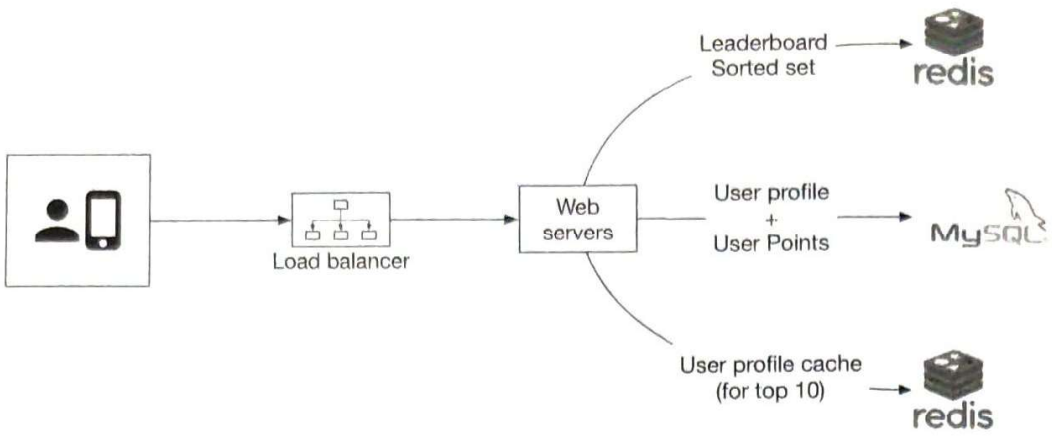


Figure 10.15: Manage our own services

Build on the cloud

The second approach is to leverage cloud infrastructures. In this section, we assume our existing infrastructure is built on AWS and that it's a natural fit to build the leaderboard on the cloud. We will use two major AWS technologies in this design: Amazon API Gateway and AWS Lambda function [8]. The Amazon API gateway provides a way to define the HTTP endpoints of a RESTful API and connect it to any backend services. We use it to connect to our AWS lambda functions. The mapping between Restful APIs and Lambda functions is shown in Table 10.5.

APIs	Lambda function
`GET /v1/scores`	`LeaderboardFetchTop10`
`GET /v1/scores/{:user_id}`	`LeaderboardFetchPlayerRank`
`POST /v1/scores`	`LeaderboardUpdateScore`

Table 10.5: Lambda functions

AWS Lambda is one of the most popular serverless computing platforms. It allows us to run code without having to provision or manage the servers ourselves. It runs only when needed and will scale automatically based on traffic. Serverless is one of the hottest topics in cloud services and is supported by all major cloud service providers. For example, Google Cloud has Google Cloud Functions [9] and Microsoft has named its offering Microsoft Azure Functions [10].

At a high level, our game calls the Amazon API Gateway, which in turn invokes the appropriate lambda functions. We will use AWS Lambda functions to invoke the appropriate commands on the storage layer (both Redis and MySQL), return the results back to the API Gateway, and then to the application.

We can leverage Lambda functions to perform the queries we need without having to spin up a server instance. AWS provides support for Redis clients that can be called from the Lambda functions. This also allows for auto-scaling as needed with DAU growth. Design diagrams for a user scoring a point and retrieving the leaderboard are shown below:

Use case 1: scoring a point

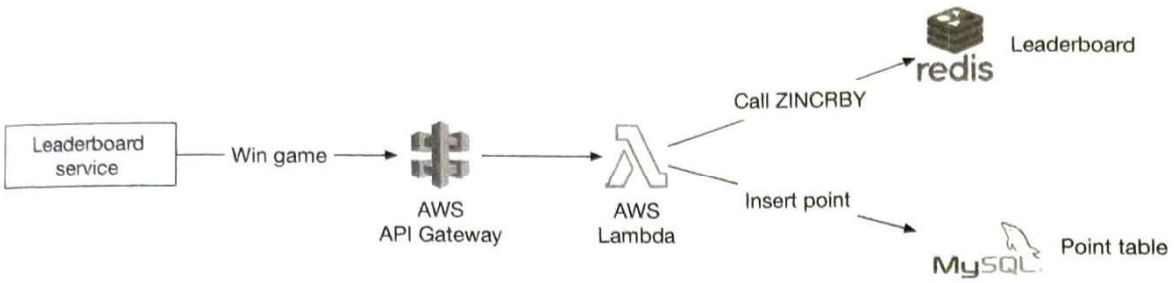


Figure 10.16: Score a point

Use case 2: retrieving leaderboard

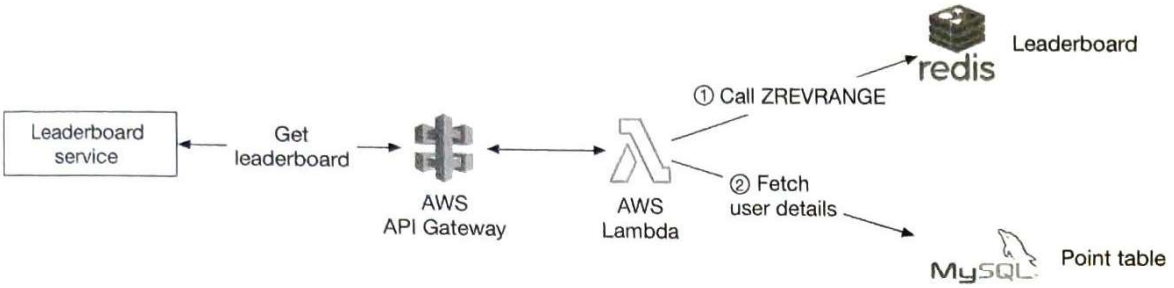


Figure 10.17: Retrieve leaderboard

Lambdas are great because they are a serverless approach, and the infrastructure will take care of auto-scaling the function as needed. This means we don't need to manage the scaling and environment setup and maintenance. Given this, we recommend going with a serverless approach if we build the game from the ground up.

Scaling Redis

With 5 million DAU, we can get away with one Redis cache from both a storage and QPS perspective. However, let's imagine we have 500 million DAU, which is 100 times our original scale. Now our worst-case scenario for the size of the leaderboard goes up to 65GB (650MB $\times$ 100), and our QPS goes up to 250,000 (2,500 $\times$ 100) queries per second. This calls for a sharding solution.

Data sharding

We consider sharding in one of the following two ways: fixed or hash partitions.

Fixed partition

One way to understand fixed partitions is to look at the overall range of points on the leaderboard. Let's say that the number of points won in one month ranges from 1 to 1000, and we break up the data by range. For example, we could have 10 shards and each shard would have a range of 100 scores (For example, $1 \sim 100$, $101 \sim 200$, $201 \sim 300$, ...) as shown in Figure 10.18.

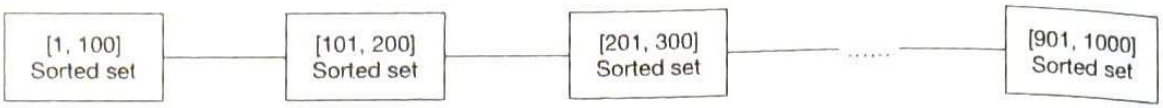


Figure 10.18: Fixed partition

For this to work, we want to ensure there is an even distribution of scores across the leaderboard. Otherwise, we need to adjust the score range in each shard to make sure of a relatively even distribution. In this approach, we shard the data ourselves in the application code.

When we are inserting or updating the score for a user, we need to know which shard they are in. We could do this by calculating the user's current score from the MySQL database. This can work, but a more performant option is to create a secondary cache to store the mapping from user ID to score. We need to be careful when a user increases their score and moves between shards. In this case, we need to remove the user from their current shard and move them to the new shard.

To fetch the top 10 players in the leaderboard, we would fetch the top 10 players from the shard (sorted set) with the highest scores. In Figure 10.18, the last shard with scores [901, 1000] contains the top 10 players.

To fetch the rank of a user, we would need to calculate the rank within their current shard (local rank), as well as the total number of players with higher scores in all of the shards. Note that the total number of players in a shard can be retrieved by running the `info keyspace` command in $O(1)$ [11].

Hash partition

A second approach is to use the Redis cluster, which is desirable if the scores are very clustered or clumped. Redis cluster provides a way to shard data automatically across multiple Redis nodes. It doesn't use consistent hashing but a different form of sharding, where every key is part of a **hash slot**. There are 16384 hash slots [12] and we can compute the hash slot of a given key by doing `CRC16(key)` %16384 [13]. This allows us to add and remove nodes in the cluster easily without redistributing all the keys. In Figure 10.19, we have 3 nodes, where:

- The first node contains hash slots [0, 5500].
- The second node contains hash slots [5501, 11000].
- The third node contains hash slots [11001, 16383].

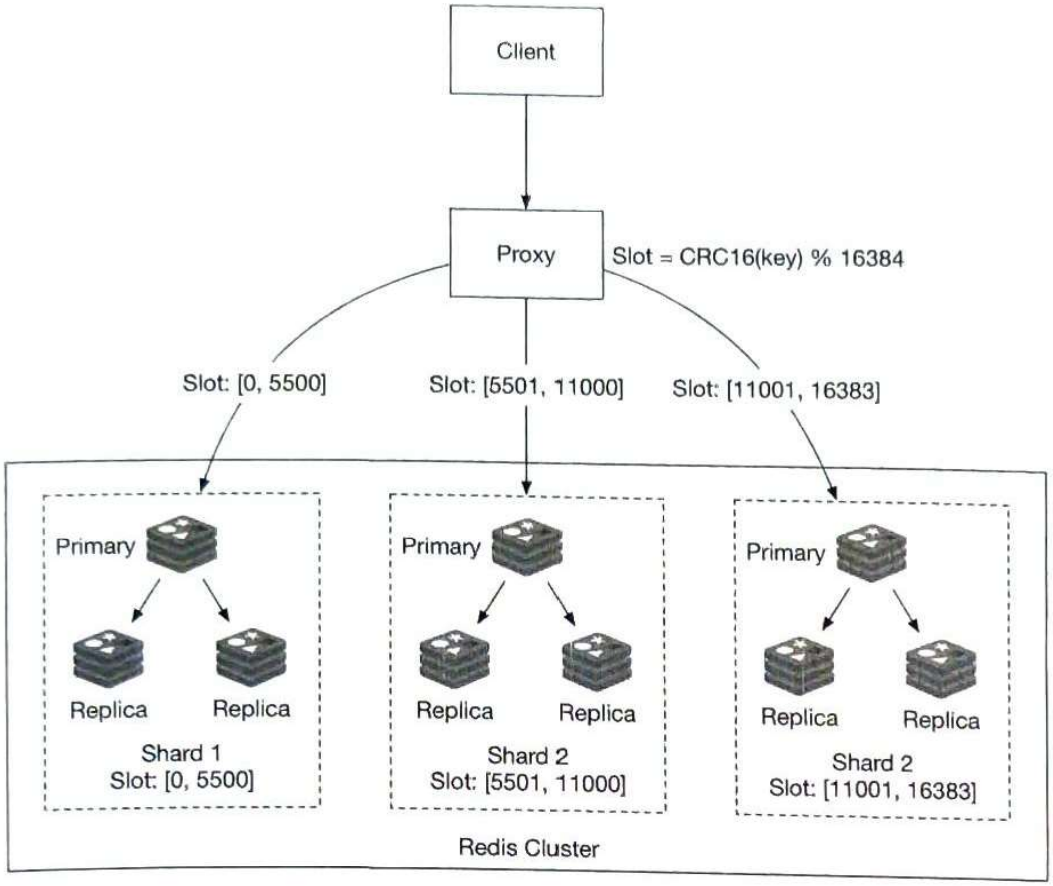


Figure 10.19: Hash partition

An update would simply change the score of the user in the corresponding shard (determined by `CRC16(key) %16384`). Retrieving the top 10 players on the leaderboard is more complicated. We need to gather the top 10 players from each shard and have the application sort the data. A concrete example is shown in Figure 10.20. Those queries can be parallelized to reduce latency.

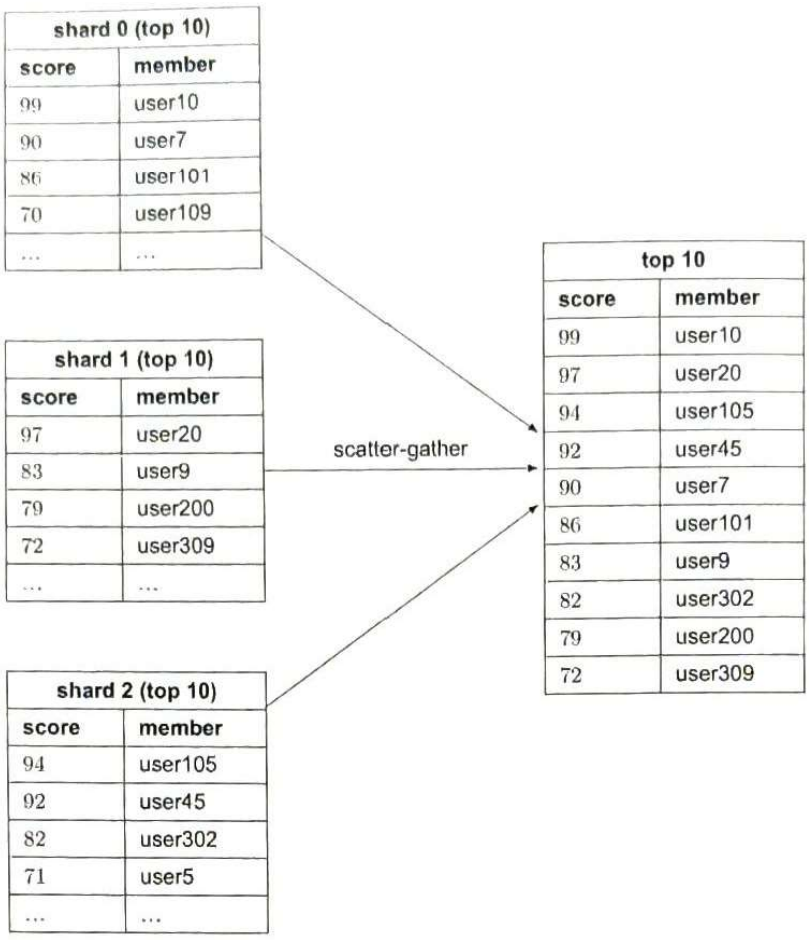


Figure 10.20: Scatter-gather

This approach has a few limitations:

- When we need to return top k results (where k is a very large number) on the leaderboard, the latency is high because a lot of entries are returned from each shard and need to be sorted.
- Latency is high if we have lots of partitions because the query has to wait for the slowest partition.
- Another issue with this approach is that it doesn't provide a straightforward solution for determining the rank of a specific user.

Therefore, we lean towards the first proposal: fixed partition.

Sizing a Redis node

There are multiple things to consider when sizing the Redis nodes [14]. Write-heavy applications require much more available memory, since we need to be able to accommodate all of the writes to create the snapshot in case of a failure. To be safe, allocate twice the amount of memory for write-heavy applications.

Redis provides a tool called Redis-benchmark that allows us to benchmark the perfor-

mance of the Redis setup, by simulating multiple clients executing multiple queries and returning the number of requests per second for the given hardware. To learn more about Redis-benchmark, see [15].

Alternative solution: NoSQL

An alternative solution to consider is NoSQL databases. What kind of NoSQL should we use? Ideally, we want to choose a NoSQL that has the following properties:

- Optimized for writes.
- Efficiently sort items within the same partition by score.

NoSQL databases such as Amazon's DynamoDB [16], Cassandra, or MongoDB can be a good fit. In this chapter, we use DynamoDB as an example. DynamoDB is a fully managed NoSQL database that offers reliable performance and great scalability. To allow efficient access to data with attributes other than the primary key, we can leverage global secondary indexes [17] in DynamoDB. A global secondary index contains a selection of attributes from the parent table, but they are organized using a different primary key. Let's take a look at an example.

The updated system diagram is shown in Figure 10.21. Redis and MySQL are replaced with DynamoDB.

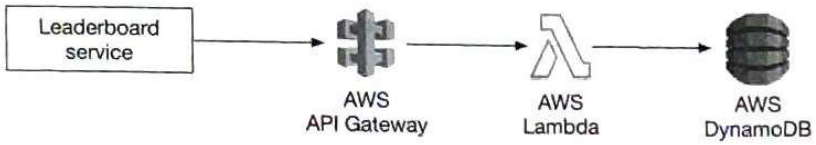


Figure 10.21: DynamoDB solution

Assume we design the leaderboard for a chess game and our initial table is shown in Figure 10.22. It is a denormalized view of the leaderboard and user tables and contains everything needed to render a leaderboard.

Primary key	Attributes			
user_id	score	email	profile_pic	leaderboard_name
lovelove	309	love@test.com	https://cdn.example/3.png	chess#2020-02
i_love_tofu	209	test@test.com	https://cdn.example/p.png	chess#2020-02
golden_gate	103	gold@test.com	https://cdn.example/2.png	chess#2020-03
pizza_or_bread	203	piz@test.com	https://cdn.example/31.png	chess#2021-05
ocean	10	oce@test.com	https://cdn.example/32.png	chess#2020-02
...	...	...	...	...

Figure 10.22: Denormalized view of the leaderboard and user tables

This table scheme works, but it doesn't scale well. As more rows are added, we have to scan the entire table to find the top scores.

linear scan, we need to add indexes. Our first attempt is to use year-month} as the partition key and the score as the sort key, as gure 10.23.

condary Index		Attributes		
ey	Sort key (score)	user_id	email	profile_pic
-02	309	lovelove	love@test.com	https://cdn.example/3.png
02	209	i_love_tofu	test@test.com	https://cdn.example/p.png
03	103	golden_gate	gold@test.com	https://cdn.example/2.png
02	203	pizza_or_bread	piz@test.com	https://cdn.example/31.png
02	10	ocean	oce@test.com	https://cdn.example/32.png
	...	...	...	...

Figure 10.23: Partition key and sort key

t runs into issues at a high load. DynamoDB splits data across multiple sistent hashing. Each item lives in a corresponding node based on its 'e want to structure the data so that data is evenly distributed across r table design (Figure 10.23), all the data for the most recent month n one partition and that partition becomes a hot partition. How can we n?

ta into n partitions and append a partition number (`user_id % ions`) to the partition key. This pattern is called write sharding. Write nplexity for both read and write operations, so we should consider the y.

ion we need to answer is, how many partitions should we have? It rite volume or DAU. The important thing to remember is that there is n load on partitions and read complexity. Because data for the same enly across multiple partitions, the load for a single partition is much to read items for a given month, we have to query all the partitions lts, which adds read complexity.

oks something like this: `game_name#{year-month}#p{partition_number}`. the updated schema table.

Global Secondary Index		
Partition key (PK)	Sort key (score)	
ess#2020-02#p0	309	
ess#2020-02#p1	209	
ess#2020-03#p2	103	
ess#2020-02#p1	203	
ess#2020-02#p2	10	
	...	

Figu

global secondary index us
tion key and the score as th
sorted within their own par
in order to fetch the top 10
ter" mentioned earlier. We
is the "scatter" portion), and
ng all the partitions (this is
5

top 10 from partition 0 (scatte

rtition key (PK)	Sort key (score)	use
ss#2020-02#p0	309	lovelo
	...	...

top 10 from partition 1 (scatter)

rtition key (PK)	Sort key (score)	user_id
ss#2020-02#p1	209	i_love_to
ss#2020-02#p1	203	pizza_or
	...	...

top 10 from partition 2 (scatter)

rtition key (PK)	Sort key (score)	user_id
ss#2020-02#p2	10	ocean
	...	...

Figure

Global Secondary Index		Attributes		
Partition key (PK)	Sort key (score)	user_id	email	profile_pic
chess#2020-02#p0	309	lovelove	love@test.com	https://cdn.example/3.png
chess#2020-02#p1	209	i_love_tofu	test@test.com	https://cdn.example/p.png
chess#2020-03#p2	103	golden_gate	gold@test.com	https://cdn.example/2.png
chess#2020-02#p1	203	pizza_or_bread	piz@test.com	https://cdn.example/31.png
chess#2020-02#p2	10	ocean	oce@test.com	https://cdn.example/32.png
...	...	...	...	...

Figure 10.24: Updated partition key

The global secondary index uses `game_name#{year-month}#p{partition_number}` as the partition key and the score as the sort key. What we end up with are n partitions that are all sorted within their own partition (locally sorted). If we assume we had 3 partitions, then in order to fetch the top 10 leaderboard, we would use the approach called "scatter-gather" mentioned earlier. We would fetch the top 10 results in each of the partitions (this is the "scatter" portion), and then we would allow the application to sort the results among all the partitions (this is the "gather" portion). An example is shown in Figure 10.25.

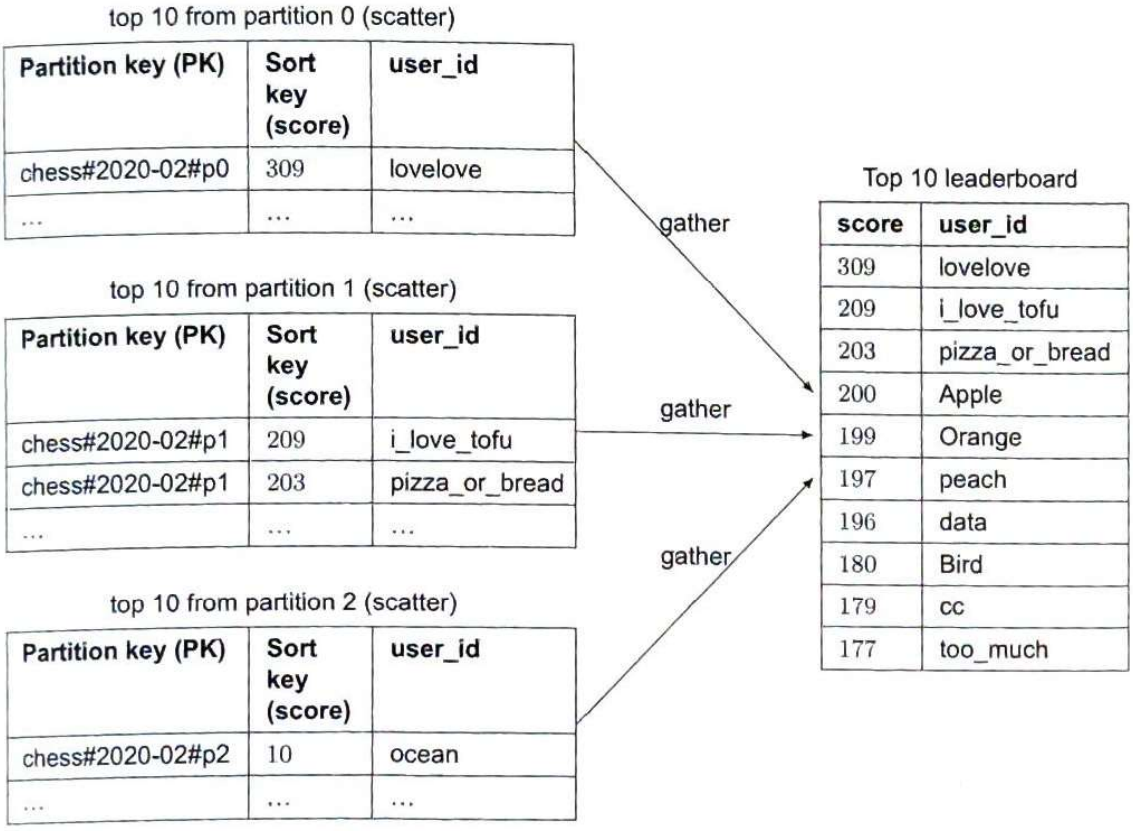


Figure 10.25: Scatter-gather

How do we decide on the number of partitions? This might require some careful benchmarking. More partitions decrease the load on each partition but add complexity, as we need to scatter across more partitions to build the final leaderboard. By employing benchmarking, we can see the trade-off more clearly.

However, similar to the Redis partition solution mentioned earlier, this approach doesn't provide a straightforward solution for determining the relative rank of a user. But it is possible to get the percentile of a user's position, which could be good enough. In real life, telling a player that they are in the top $10 \sim 20\%$ might be better than showing the exact rank at eg. 1,200,001. Therefore, if the scale is large enough that we needed to shard, we could assume that the score distributions are roughly the same across all shards. If this assumption is true, we could have a cron job that analyzes the distribution of the score for each shard, and caches that result.

The result would look something like this:

$$\begin{aligned} &\text{10th percentile} = \text{score} < 100 \\ &\text{20th percentile} = \text{score} < 500 \\ &\quad\vdots \qquad\qquad\quad \vdots \\ &\text{90th percentile} = \text{score} < 6500 \end{aligned}$$

Then we could quickly return a user's relative ranking (say 90th percentile).

Step 4 - Wrap Up

In this chapter, we have created a solution for building a real-time game leaderboard with the scale of millions of DAU. We explored the straightforward solution of using a MySQL database and rejected that approach because it does not scale to millions of users. We then designed the leaderboard using Redis sorted sets. We also looked into scaling the solution to 500 million DAU, by leveraging sharding across different Redis caches. We also proposed an alternative NoSQL solution.

In the event you have some extra time at the end of the interview, you can cover a few more topics:

Faster retrieval and breaking tie

A Redis Hash provides a map between string fields and values. We could leverage a hash for 2 use cases:

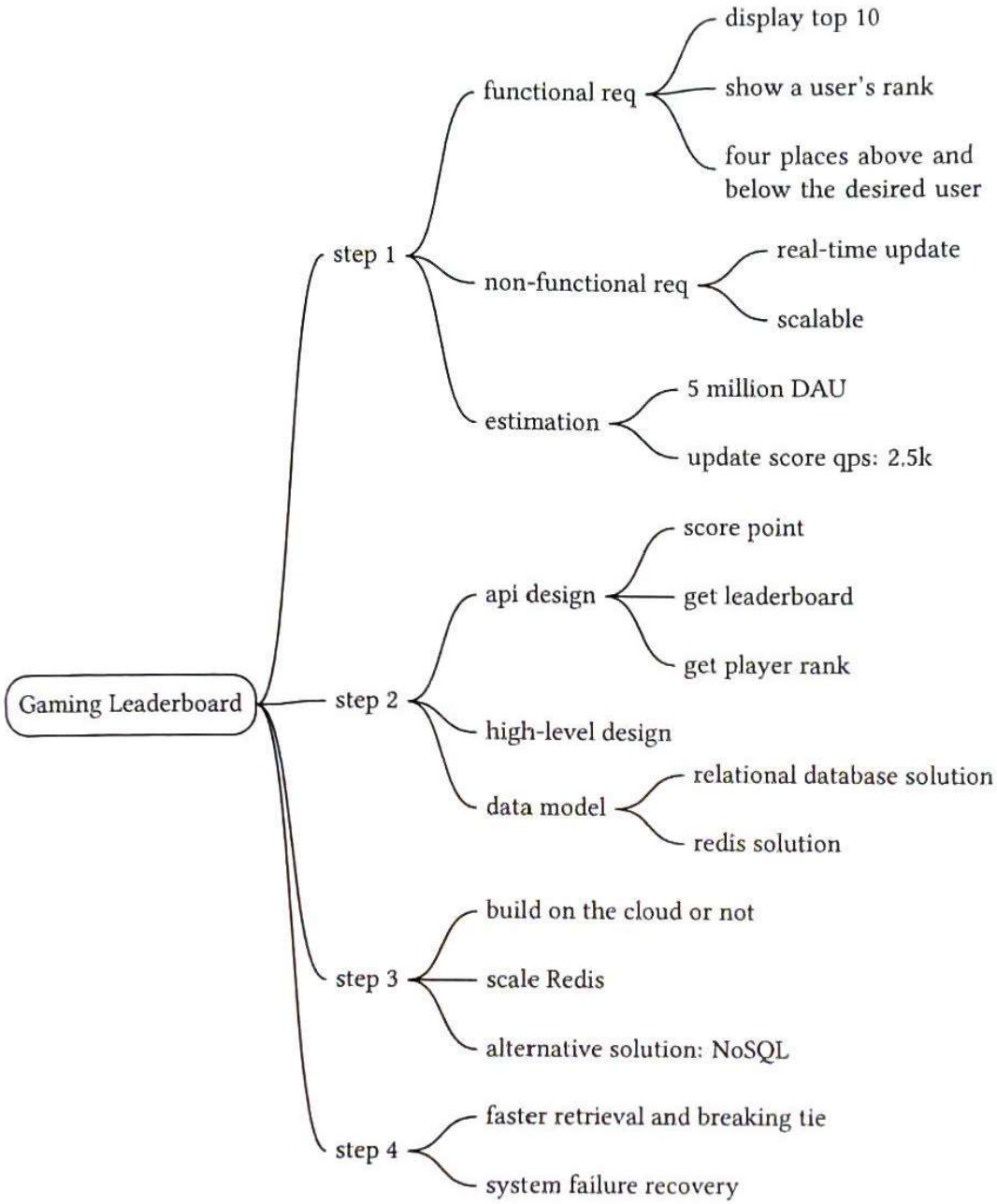
1. To store a map of the user id to the user object that we can display on the leaderboard. This allows for faster retrieval than having to go to the database to fetch the user object.
2. In the case of two players having the same scores, we could rank the users based on who received that score first. When we increment the score of the user, we can also store a map of the user id to the timestamp of the most recently won game. In the case of a tie, the user with the older timestamp ranks higher.

System failure recovery

The Redis cluster can potentially experience a large-scale failure. Given the design above, we could create a script that leverages the fact that the MySQL database records an entry with a timestamp each time a user won a game. We could iterate through all of the entries for each user, and call `ZINCRBY` once per entry, per user. This would allow us to recreate the leaderboard offline if necessary, in case of a large-scale outage.

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

Chapter Summary



Reference Material

[1] Man-in-the-middle attack. https://en.wikipedia.org/wiki/Man-in-the-middle_attack.

[2] Redis Sorted Set source code. https://github.com/redis/redis/blob/unstable/src/t_zset.c.

[3] Geekbang. https://static001.geekbang.org/resource/image/46/a9/46d283cd82c987153b3fe0c76dfba8a9.jpg.

[4] Building real-time Leaderboard with Redis. https://medium.com/@sandeep4.verma/building-real-time-leaderboard-with-redis-82c98aa47b9f.

[5] Build a real-time gaming leaderboard with Amazon ElastiCache for Redis. https://aws.amazon.com/blogs/database/building-a-real-time-gaming-leaderboard-with-amazon-elasticache-for-redis.

[6] How we created a real-time Leaderboard for a million Users. https://levelup.gitconnected.com/how-we-created-a-real-time-leaderboard-for-a-million-users-555aaa3ccf7b.

[7] Leaderboards. https://redislabs.com/solutions/use-cases/leaderboards/.

[8] Lambda. https://aws.amazon.com/lambda/.

[9] Google Cloud Functions. https://cloud.google.com/functions.

[10] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[11] Info command. https://redis.io/commands/INFO.

[12] Why redis cluster only have 16384 slots. https://stackoverflow.com/questions/36203532/why-redis-cluster-only-have-16384-slots.

[13] Cyclic redundancy check. https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[14] Choosing your node size. https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/nodes-select-size.html.

[15] How fast is Redis? https://redis.io/topics/benchmarks.

[16] Using Global Secondary Indexes in DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html.

[17] Leaderboard & Write Sharding. https://www.dynamodbguide.com/leaderboard-write-sharding/.

11 Payment System

In this chapter, we design a payment system. E-commerce has exploded in popularity across the world in recent years. What makes every transaction possible is a payment system running behind the scenes. A reliable, scalable, and flexible payment system is essential.

What is a payment system? According to Wikipedia, "a payment system is any system used to settle financial transactions through the transfer of monetary value. This includes the institutions, instruments, people, rules, procedures, standards, and technologies that make its exchange possible" [1].

A payment system is easy to understand on the surface but is also intimidating for many developers to work on. A small slip could potentially cause significant revenue loss and destroy credibility among users. But fear not! In this chapter, we demystify payment systems.

Step 1 - Understand the Problem and Establish Design Scope

A payment system can mean very different things to different people. Some may think it's a digital wallet like Apple Pay or Google Pay. Others may think it's a backend system that handles payments such as PayPal or Stripe. It is very important to determine the exact requirements at the beginning of the interview. These are some questions you can ask the interviewer:

Candidate: What kind of payment system are we building?
Interviewer: Assume you are building a payment backend for an e-commerce application like Amazon.com. When a customer places an order on Amazon.com, the payment system handles everything related to money movement.

Candidate: What payment options are supported? Credit cards, PayPal, bank cards, etc?
Interviewer: The payment system should support all of these options in real life. However, in this interview, we can use credit card payment as an example.

Candidate: Do we handle credit card payment processing ourselves?

Interviewer: No, we use third-party payment processors, such as Stripe, Braintree, Square, etc.

Candidate: Do we store credit card data in our system?
Interviewer: Due to extremely high security and compliance requirements, we do not store card numbers directly in our system. We rely on third-party payment processors to handle sensitive credit card data.

Candidate: Is the application global? Do we need to support different currencies and international payments?
Interviewer: Great question. Yes, the application would be global but we assume only one currency is used in this interview.

Candidate: How many payment transactions per day?
Interviewer: 1 million transactions per day.

Candidate: Do we need to support the pay-out flow, which an e-commerce site like Amazon uses to pay sellers every month?
Interviewer: Yes, we need to support that.

Candidate: I think I have gathered all the requirements. Is there anything else I should pay attention to?
Interviewer: Yes. A payment system interacts with a lot of internal services (accounting, analytics, etc.) and external services (payment service providers). When a service fails, we may see inconsistent states among services. Therefore, we need to perform reconciliation and fix any inconsistencies. This is also a requirement.

With these questions, we get a clear picture of both the functional and non-functional requirements. In this interview, we focus on designing a payment system that supports the following.

Functional requirements

- Pay-in flow: payment system receives money from customers on behalf of sellers.
- Pay-out flow: payment system sends money to sellers around the world.

Non-functional requirements

- Reliability and fault tolerance. Failed payments need to be carefully handled.
- A reconciliation process between internal services (payment systems, accounting systems) and external services (payment service providers) is required. The process asynchronously verifies that the payment information across these systems is consistent.

Back-of-the-envelope estimation

The system needs to process 1 million transactions per day, which is 1,000,000 transactions $/10^5$ seconds = 10 transactions per second (TPS). 10 TPS is not a big number for a typical database, which means the focus of this system design interview is on how to

correctly handle payment transactions, rather than aiming for high throughput.

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

At a high level, the payment flow is broken down into two steps to reflect how flows:

- Pay-in flow
- Pay-out flow

Take the e-commerce site, Amazon, as an example. After a buyer places a money flows into Amazon's bank account, which is the pay-in flow. Althoug is in Amazon's bank account, Amazon does not own all of the money. Th a substantial part of it and Amazon only works as the money custodian fo when the products are delivered and money is released, the balance after fe from Amazon's bank account to the seller's bank account. This is the pay- simplified pay-in and pay-out flows are shown in Figure 11.1.

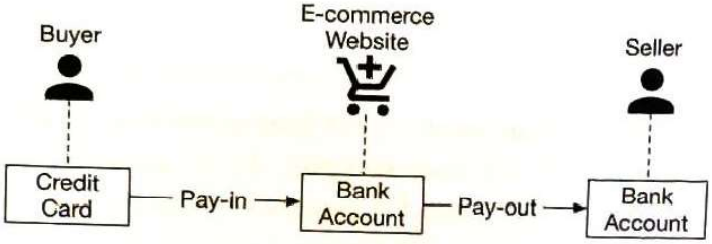


Figure 11.1: Simplified pay-in and pay-out flow

Pay-in flow

The high-level design diagram for the pay-in flow is shown in Figure 11.2. Let's take a look at each component of the system.

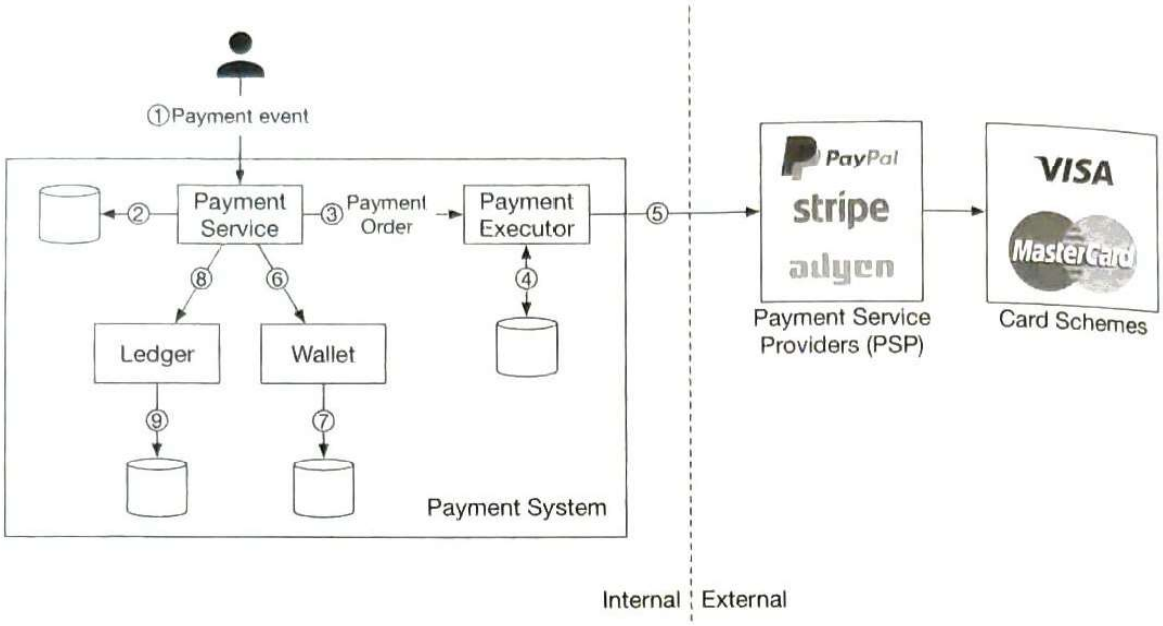


Figure 11.2: Pay-in flow

Payment service

The payment service accepts payment events from users and coordinates the payment process. The first thing it usually does is a risk check, assessing for compliance with regulations such as AML/CFT [2], and for evidence of criminal activity such as money laundering or financing of terrorism. The payment service only processes payments that pass this risk check. Usually, the risk check service uses a third-party provider because it is very complicated and highly specialized.

Payment executor

The payment executor executes a single payment order via a Payment Service Provider (PSP). A payment event may contain several payment orders.

Payment Service Provider (PSP)

A PSP moves money from account A to account B. In this simplified example, the PSP moves the money out of the buyer's credit card account.

Card schemes

Card schemes are the organizations that process credit card operations. Well known card schemes are Visa, MasterCard, Discovery, etc. The card scheme ecosystem is very complex [3].

Ledger

The ledger keeps a financial record of the payment transaction. For example, when a user pays the seller $1, we record it as debit $1 from the user and credit $1 to the seller. The ledger system is very important in post-payment analysis, such as calculating the total revenue of the e-commerce website or forecasting future revenue.

Wallet

The wallet keeps the account balance of the merchant. It may also record how much a given user has paid in total.

As shown in Figure 11.2, a typical pay-in flow works like this:

1. When a user clicks the "place order" button, a payment event is generated and sent to the payment service.
2. The payment service stores the payment event in the database.
3. Sometimes, a single payment event may contain several payment orders. For example, you may select products from multiple sellers in a single checkout process. If the e-commerce website splits the checkout into multiple payment orders, the payment service calls the payment executor for each payment order.
4. The payment executor stores the payment order in the database.
5. The payment executor calls an external PSP to process the credit card payment.
6. After the payment executor has successfully processed the payment, the payment service updates the wallet to record how much money a given seller has.
7. The wallet server stores the updated balance information in the database.
8. After the wallet service has successfully updated the seller's balance information, the payment service calls the ledger to update it.
9. The ledger service appends the new ledger information to the database.

APIs for payment service

We use the RESTful API design convention for the payment service.

`POST /v1/payments`

This endpoint executes a payment event. As mentioned above, a single payment event may contain multiple payment orders. The request parameters are listed below:

Field	Description	Type
`buyer_info`	The information of the buyer	json
`checkout_id`	A globally unique ID for this checkout	string
`credit_card_info`	This could be encrypted credit card information or a payment token. The value is PSP-specific.	json
`payment_orders`	A list of the payment orders	list

Table 11.1: API request parameters (execute a payment event)

The `payment_orders` look like this: