What is a blob store?

Blob store is a storage solution for unstructured data. We can store photos, audio, videos, binary executable codes, or other multimedia items in a blob store. Every type of data is stored as a blob. It follows a flat data organization pattern where there are no hierarchies, that is, directories, sub-directories, and so on.

Why do we use a blob store?

Blob store is an important component of many data-intensive applications, such as YouTube, Netflix, Facebook, and so on. The table below displays the blob storage used by some of the most well-known applications. These applications generate a huge amount of unstructured data every day. They require a storage solution that is easily scalable, reliable, and highly available, so that they can store large media files.

System	Blob Store
Netflix	S3
YouTube	Google Cloud Storage
Facebook	Tectonic

Requirements of Blob store design

Funtional Requirements

Create a container: The users should be able to create containers in order to group blobs. For example, if an application wants to store user-specific data, it should be able to store blobs for different user accounts in different containers. Additionally, a user may want to group video blobs and separate them from a group of image blobs. A single blob store user can create many containers, and each container can have many blobs, as shown in the following illustration. For the sake of simplicity, we assume that we can't create a container inside a container.

Multiple containers associated with a single storage account, and multiple blobs inside a single container
Put data: The blob store should allow users to upload blobs to the created containers.

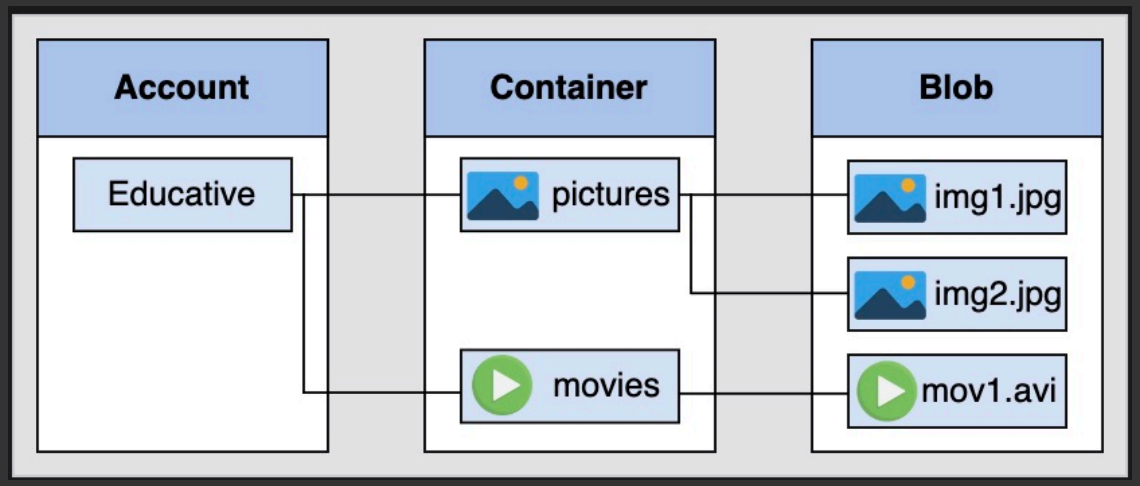
Get data: The system should generate a URL for the uploaded blob, so that the user can access that blob later through this URL.

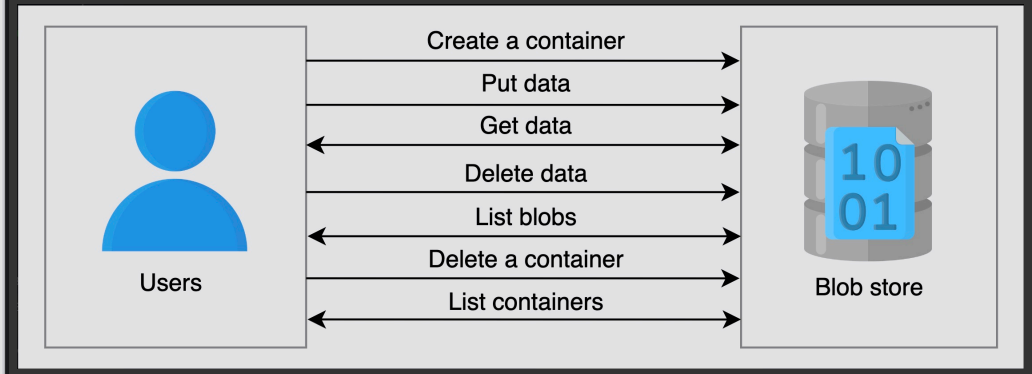
Delete data: The users should be able to delete a blob. If the user wants to keep the data for a specified period of time (retention time), our system should support this functionality.

List blobs: The user should be able to get a list of blobs inside a specific container.

Delete a container: The users should be able to delete a container and all the blobs inside it.

List containers: The system should allow the users to list all the containers under a specific account.





Non Functional Requirement

Availability: Our system should be highly available.

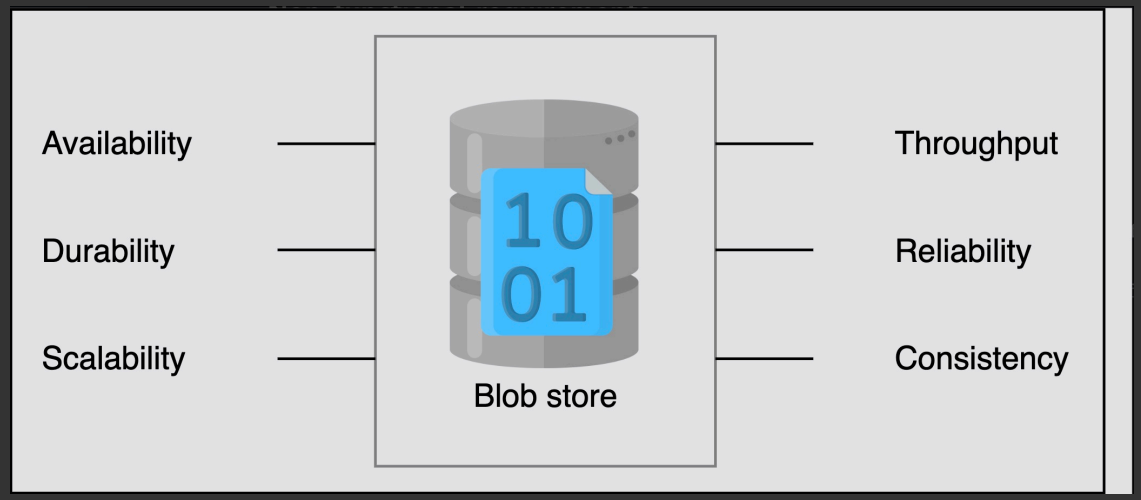
Durability: The data, once uploaded, shouldn't be lost unless users explicitly delete that data.

Scalability: The system should be capable of handling billions of blobs.

Throughput: For transferring gigabytes of data, we should ensure a high data throughput.

Reliability: Since failures are a norm in distributed systems, our design should detect and recover from failures promptly.

Consistency: The system should be strongly consistent. Different users should see the same view of a blob.



Resource Estimation

Let's estimate the total number of servers, storage, and bandwidth required by a blob storage system. Because blobs can have all sorts of data, mentioning all of those types of data in our estimation may not be practical.

The number of daily active users who upload or watch videos is five million.
The number of requests per second that a single blob store server can handle is 500.
The average size of a video is 50 MB.
The average size of a thumbnail is 20 KB.
The number of videos uploaded per day is 250,000.
The number of read requests by a single user per day is 20.

Numer of Server

↳ $\frac{\text{Number of active users}}{\text{Queries handled per server}} = 10\text{K servers.}$

Storage estimation

Total storage/day = No. of videos/day * (Storage/video + Storage/thumbnail)

Total Storage Required to Store Videos and Thumbnails Uploaded Per Day on YouTube

No. of videos per day	Storage per video (MB)	Storage per thumbnail (KB)	Total storage per day (TB)
250000	60	20	*f* 15.01

B/W estimation

$$\text{total BW} = \frac{\text{Total storage\_day}}{24 \times 60 \times 60}$$

Total storage per day (TB)	Seconds in a day	Bandwidth (Gb/s)
12.51	86400	*f* 1.16

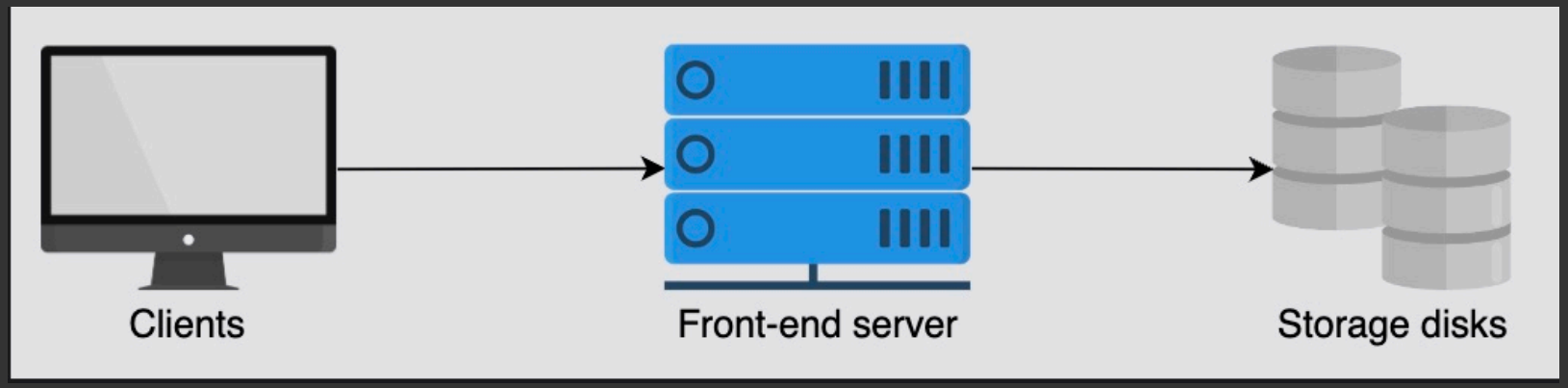
Outgoing traffic

$$Total_{bandwidth} = \frac{No.\ of\ active\ users_{/day} \times No.\ of\ requests_{/user/day} \times Total_{data\_size}}{Seconds\ in\ a\ day}$$

No. of active users per day	No. of requests per user	Data size (MB)	Bandwidth required (Gb/s)
5000000	20	50	462.96

Design of blob storage

1. High level design



API design

`Create container`

`The createContainer operation creates a new container under the logged-in account from which this request is being generated.`

Parameter	Description
`containerName`	This is the name of the container. It should be unique within a storage account.

Upload blobs

The client's data is stored in the form of Bytes in the blob store. The data can be put into a container with the following code:

```
putBlob(containerPath, blobName, data)
```

Parameter	Description
`containerPath`	This is the path of the container in which we upload the blob. It consists of the `accountID` and `containerID`.
`blobName`	This is the name of the blob. It should be unique within a container, otherwise our system will give the blob that was uploaded later a version number.
`data`	This is a file that the user wants to upload to the blob store.

Download blobs

Blobs are identified by their unique name or ID.

```
getBlob(blobPath)
```

Parameter	Description
`blobPath`	This is the fully qualified path of the data or file, including its unique ID.

Delete blob

The deleteBlob operation marks the specified blob for deletion. The actual blob is deleted during garbage collection.

```
deleteBlob(blobPath)
```

Parameter	Description
`blobPath`	This is the path of the blob that the user wants to delete.

List blobs

The listBlobs operation returns a list of blobs under the specified container or path.

listBlobs(containerPath)

Parameter	Description
`containerPath`	This is the path to the container from which the user wants to get the list of blobs.

Delete container

The deleteContainer operation marks the specified container for deletion. The container and any blobs in it are deleted later during garbage collection.

deleteContainer(containerPath)

Parameter	Description
`containerPath`	This is the path to the container that the user wants to delete.

List containers

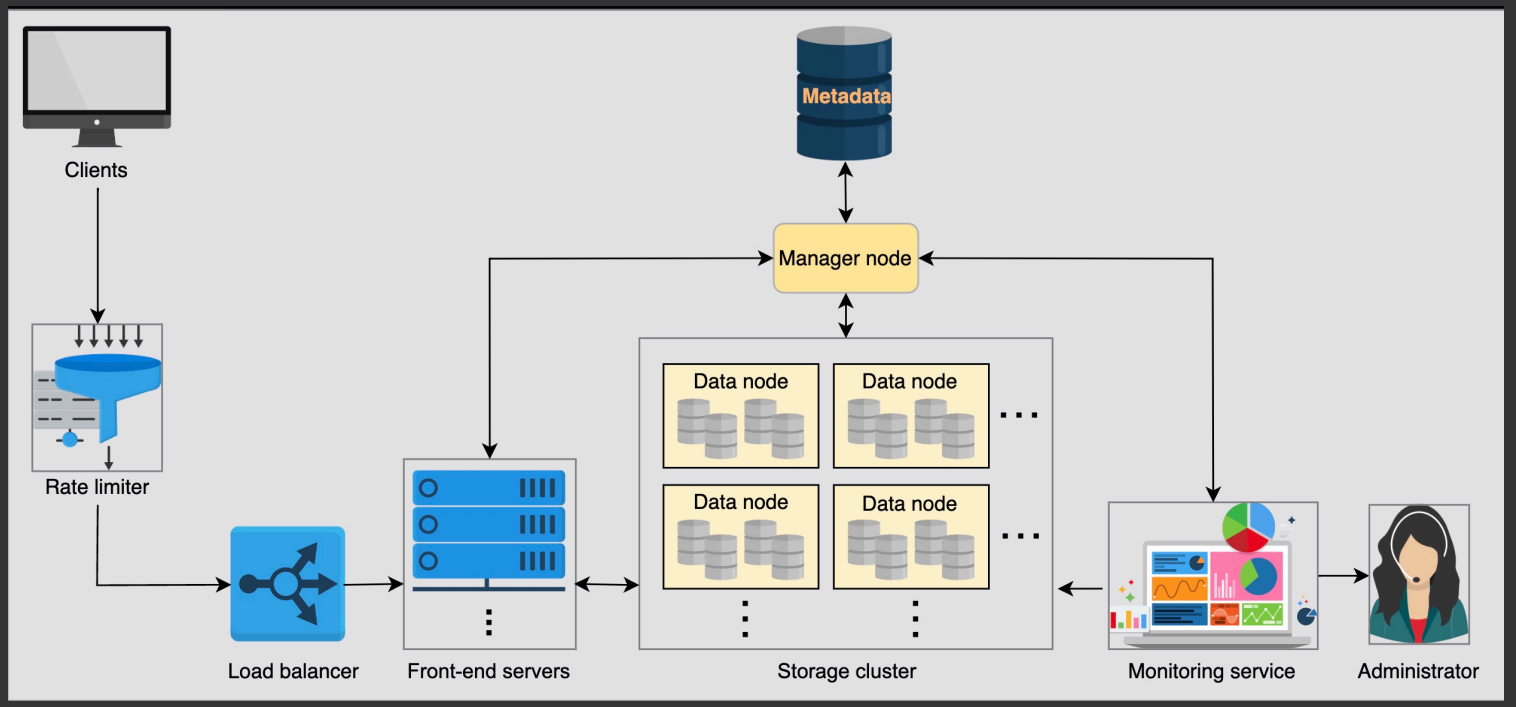
The listContainers operation returns a list of the containers under the specified user's blob store account.

listContainers(accountID)

Parameter	Description
`accountID`	This is the ID of the user who wants to list their containers.

Components of blob storage

- Client
- Rate limiter
- Load balancer
- Front-end servers
- Data Nodes
- Manager Nodes
- Metadata storage
- Monitoring service
- Administrator.



What does the manager node do if a user concurrently writes two blobs with the same name inside the same container?

Hide Answer
The manager node serializes such operations and assigns a version number to the blob that's uploaded later.

Client: This is a user or program that performs any of the API functions that are specified.

Rate limiter: A rate limiter limits the number of requests based on the user's subscription or limits the number of requests from the same IP address at the same time. It doesn't allow users to exceed the predefined limit.

Load balancer: A load balancer distributes incoming network traffic among a group of servers. It's also used to reroute requests to different regions depending on the location of the user, different data centers within the same region, or different servers within the same data center. DNS load balancing can be used to reroute the requests among different regions based on the location of the user.

Front-end servers: Front-end servers forward
the users' requests for adding or deleting data to the appropriate storage servers.

Data nodes: Data nodes hold the actual blob data. It's also possible that they contain a part of the blob's data. Blobs are split into small, fixed-size pieces called chunks. A data node can accommodate all of the chunks of a blob or at least some of them.

Manager node: A manager node is the core component that manages all data nodes. It stores information about storage paths and the access privileges of blobs. There are two types of access privileges: private and public. A private access privilege means that the blob is only accessible by the account containing that blob. A public access privilege means that anyone can access that blob.

Metadata storage: Metadata storage is a distributed database that's used by the manager node to store all the metadata. Metadata consists of account metadata, container metadata, and blob metadata.
Account metadata contains the account information for each user and the containers held by each account.
Container metadata consists of the list of the blobs in each container.

Blob metadata consists of where each blob is stored. The blob metadata is discussed in detail in the next lesson.
Monitoring service: A monitoring service monitors the data nodes and the manager node. It alerts the administrator in case of disk failures that require human intervention. It also gets information about the total available space left on the disks to alert administrators to add more disks.

Administrator: An administrator is responsible for handling notifications from the monitoring services and conducting routine checkups of the overall service to ensure reliability.

Suppose the manager node moves data from one data node to another because of an impending disk failure. The user will now have stale information if they use the cached metadata to access the data. How do we handle such situations?

Hide Answer
In such cases, the client calls fail. The client then flushes the cache and fetches new metadata information from the manager node.

Can the manager node be considered a single point of failure? If yes, then how can we cope with this problem?

Hide Answer
Yes, because the manager node is the central point of a blob store and is a single point of failure. Therefore, we have to have a backup or shadow server in place of a manager node.

The technique that we use for this is called checkpointing, meaning we take snapshots of the data at different time intervals. A snapshot captures the state, data, hardware configuration of the running manager node, and messages in transit between the manager and data nodes. It maintains the operation log in an external storage area or snapshot repository. If the manager node fails, an automated system or the administrator uses the snapshot to restart that manager node from the state it failed at and replays the operation log.

Summary of the Lesson

Section	Purpose
Blob metadata	This is the metadata that's maintained to ensure efficient storage and retrieval of blobs.
Partitioning	This determines how blobs are partitioned among different data nodes.
Blob indexing	This shows us how to efficiently search for blobs.
Pagination	This teaches us how to conceive a method for the retrieval of a limited number of blobs to ensure improved readability and loading time.
Replication	This teaches us how to replicate blobs and tells us how many copies we should maintain to improve availability.
Garbage collection	This teaches us how to delete blobs without sacrificing performance.
Streaming	This teaches us how to stream large files chunk-by-chunk to facilitate interactivity for users.
Caching	This shows us how to improve response time and throughput.

```
User account: Users uniquely get identified on this layer through their account_ID. Blobs uploaded by users are maintained in their containers.

Container: Each user has a set of containers that are all uniquely identified by a container_ID. These containers contain blobs.

Blob: This layer contains information about blobs that are uniquely identified by their blob_ID. This layer maintains information about the metadata of blobs that's vital for achieving the availability and reliability of the system.
```

Layered Information

Level	Uniquely identified by	Information	Sharded by	Mapping
User's blob store account	`account_ID`	list of `containers_ID` values	`account_ID`	Account -> list of containers
Container	`container_ID`	List of `blob_ID` values	`container_ID`	Container -> list of blobs
Blob	`blob_ID`	{list of chunks, chunkInfo: data node ID's,.. }	`blob_ID`	Blob -> list of chunks

When a user uploads a blob, it's split into small-sized chunks in order to be able to support the storage of large files that can't fit in one contiguous location, in one data node, or in one block of a disk associated with that data node. The chunks for a single blob are then stored on different data nodes that have enough storage space available to store these chunks.

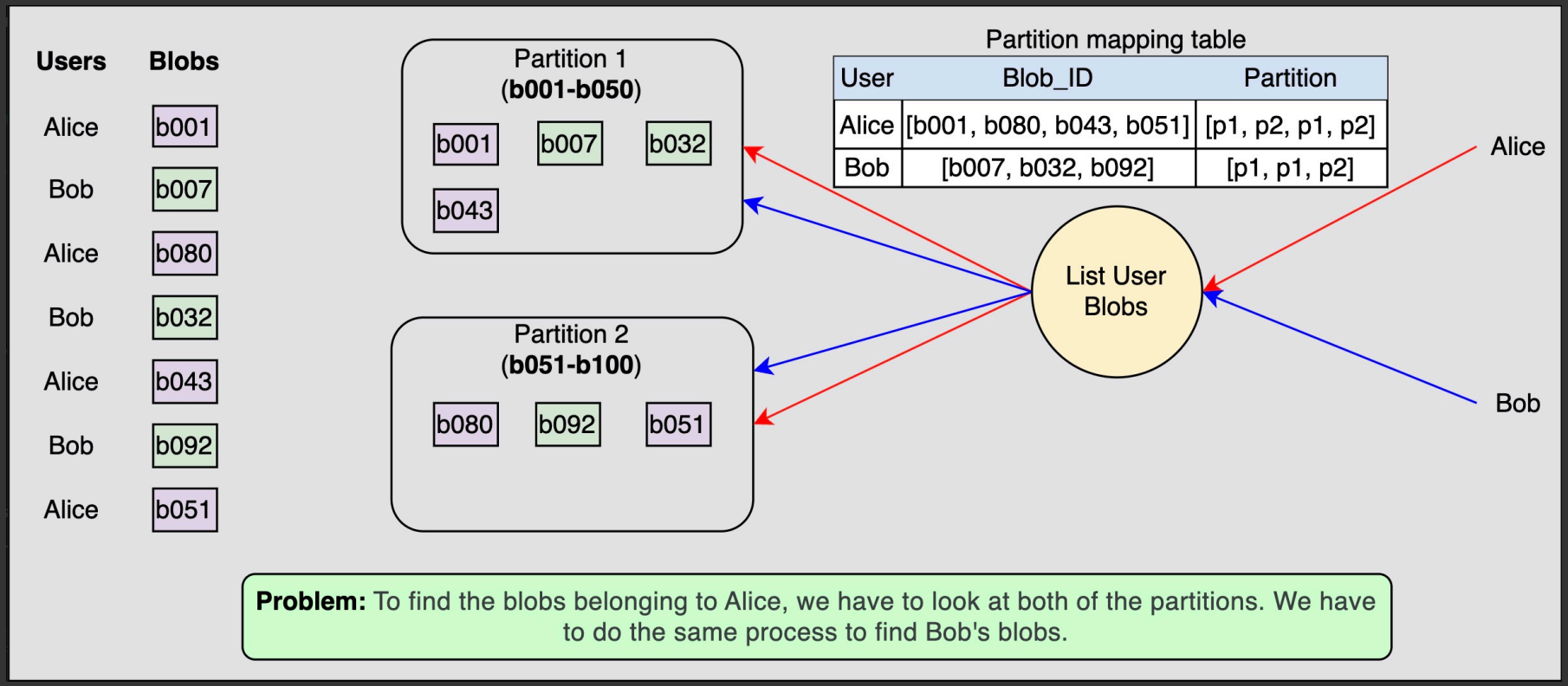
Blob Metadata

Chunk	Datanode ID	Replica 1 ID	Replica 2 ID	Replica 3 ID
1	d1b1	r1b1	r2b1	r3b1
2	d1b2	r1b2	r2b2	r3b2

What if the blob size isn't a multiple of our configured chunk size? How does the manager node know how many Bytes to read for the last chunk?

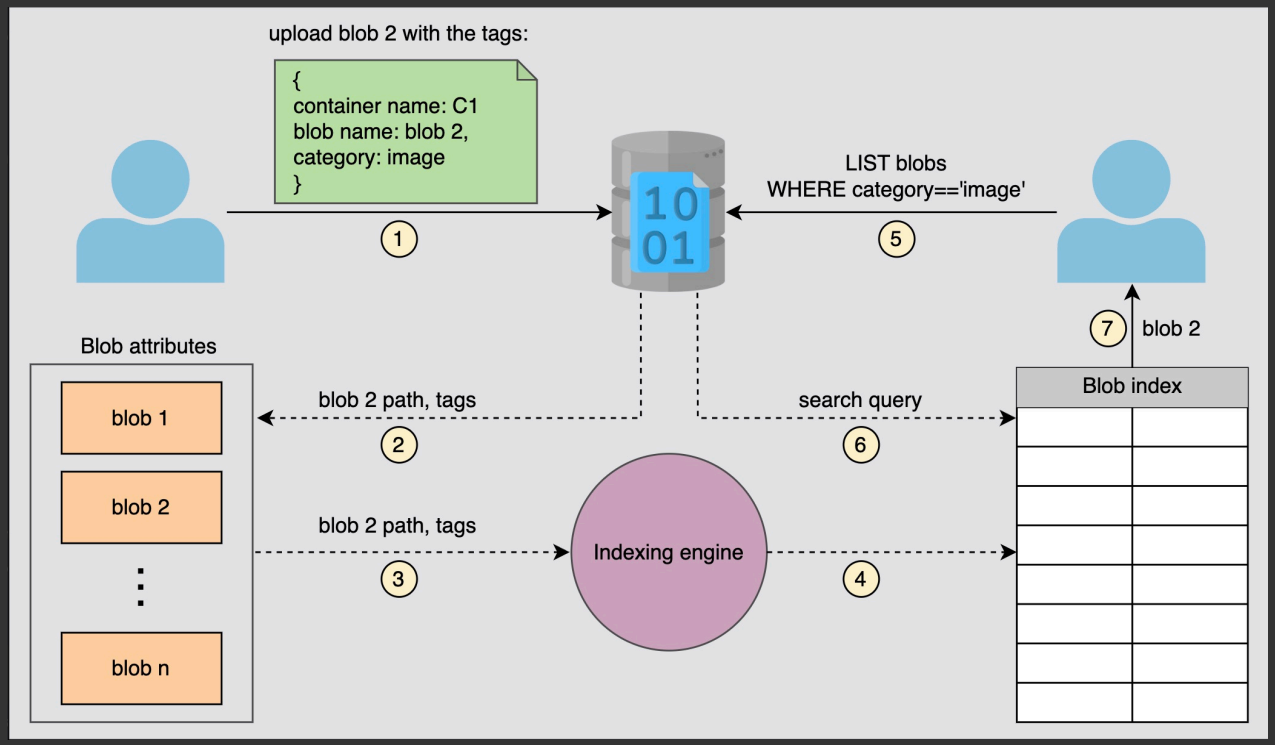
If the blob size isn't a multiple of the chunk size, the last chunk won't be full.

The manager node also keeps the size of each blob to determine the number of Bytes to read for the last chunk.



Blob indexing

Finding specific blobs in a sea of blobs becomes more difficult and time-consuming with an increase in the number of blobs that are uploaded to the storage. The blob index solves the problem of blob management and querying.



Pagination for listing

Listing is about returning a list of blobs to the user, depending on the user's entered prefix. A prefix is a character or string that returns the blobs whose name begins with that particular character or string.

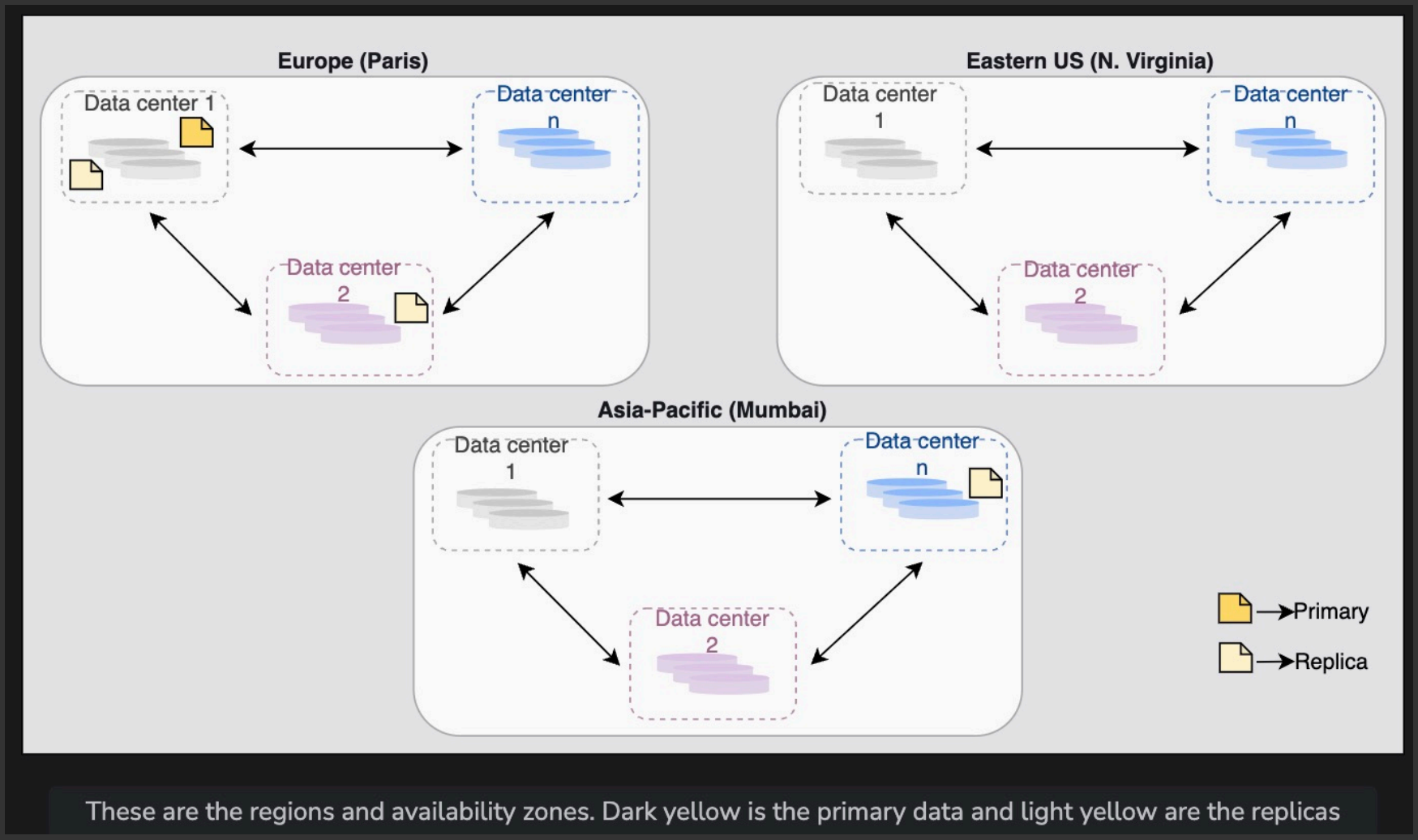
How do we decide which five blobs to return first out of the 2,000 blobs total?

Hide Answer
Here, we utilize indexing to sort and categorize the blobs. We should do this beforehand, while we store the blobs. Otherwise, it becomes challenging at the time of returning the list to the user. There could be millions or billions of blobs and we can't sort them quickly when the list request is received.

Replication is carried out on two levels to support availability and strong consistency. To keep the data strongly consistent, we synchronously replicate data among the nodes that are used to serve the read requests, right after performing the write operation

These are the two levels of replication:

Synchronous replication within a storage cluster.
Asynchronous replication across data centers and regions.



These are the regions and availability zones. Dark yellow is the primary data and light yellow are the replicas

Garbage collection while deleting a blob

Since the blob chunks are placed at different data nodes, deleting from many different nodes takes time, and holding a client until that's done is not a viable option. Due to real-time latency optimization, we don't actually remove the blob from the blob store against a delete request. Instead, we just mark a blob as "DELETED" in the metadata to make it inaccessible to the user. The blob is removed later after responding to the user's delete request.

Therefore, we have a service called a garbage collector that cleans up metadata inconsistencies later. The deletion of a blob causes the chunks associated with that blob to be freed. However, there could be an appreciable time delay between the time a blob is deleted by a user and the time of the corresponding increase in free space in the blob store.

Stream a file

To stream a file, we need to define how many Bytes are allowed to be read at one time. Let's say we read X number of Bytes each time. The first time we read the first X Bytes starting from the 0th Byte (0 to $X - 1$) and the next time, we read the next X Bytes (X to $2X - 1$).

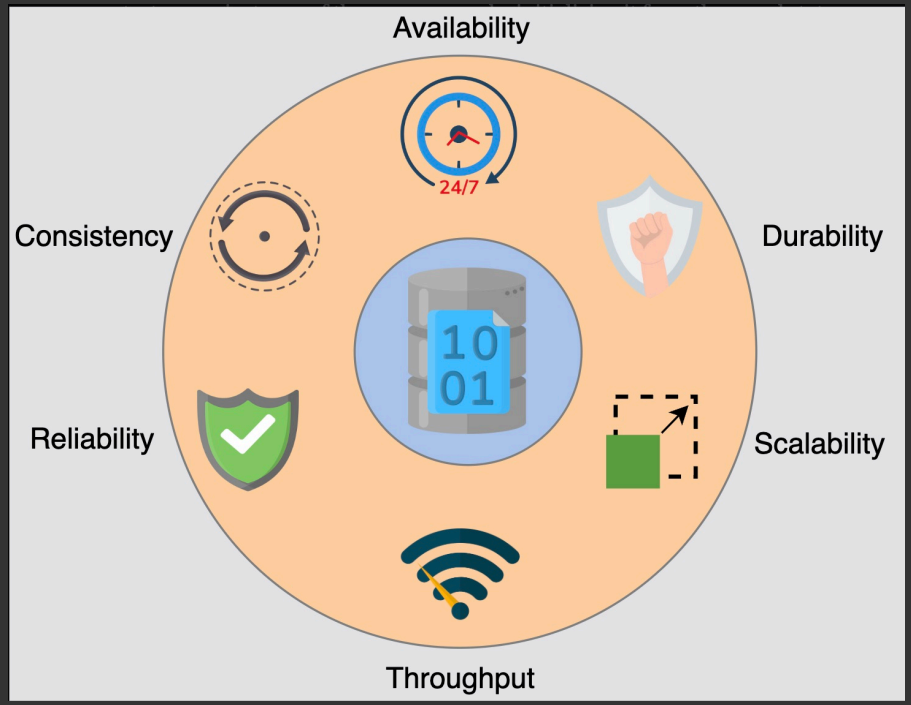
How do we know which Bytes we have read first and which Bytes we have to read next?

Hide Answer
We can use an offset value to keep track of the Byte from which we need to start reading again.

Evaluation of blob store design

Availability
The replication part of our design makes the system available. For reading the data, we keep four replicas for each blob. Having replicas, we can distribute the request load. If one node fails, the other replica node can serve the request. Moreover, our replica placement strategy handles a whole data center failure and can even handle the situation of region unavailability due to natural disasters.

Durability
The replication and monitoring services ensure the durability of the data. The data, once uploaded, is synchronously replicated within a storage cluster. If data loss occurs at one node, we can recover the data from the other nodes. The monitoring service monitors the storage disks.



Scalability
The partitioning and splitting of blobs into small-sized chunks helps us scale for billions of blob requests. Blobs are partitioned into separate ranges and served by different partition servers. The partition mappings specify which partition server will serve which particular blob range requests.

How can we further scale when our manager server hits its limits and we can’t improve its computational abilities by vertical scaling?

Hide Answer
We can make two independent instances of our system. Each instance will have its own manager node and a set of data nodes. Deployment of a system similar to ours has been shown to scale up to a few petabytes. Therefore, making additional instances can help us scale further.

For further scaling inside a single instance, we need a new, more complicated design.

Throughput
We save chunks of a blob on different data nodes that distribute the requests for a blob to multiple machines. Parallel fetching of chunks from multiple data nodes helps us achieve high throughput.

Reliability
We achieve reliability through our monitoring techniques. For example, the heartbeat protocol keeps the manager node updated on the status of data nodes. This enables the manager node to request data from reliable nodes only.
Furthermore, it takes necessary precautions to ensure reliable service. For example, the failure of a node triggers the manager node to request an additional replica node.

Consistency
We synchronously replicate the disk data blocks inside a storage cluster upon a write request, making the data strongly consistent inside the storage cluster.