### **System Design: A Blob Store - Summary**

#### **Introduction to Blob Store**

A blob store is a storage solution designed for unstructured data, such as photos, audio, videos, binary executable codes, and other multimedia items. Data in a blob store is stored as a "blob," following a flat data organization pattern without hierarchical structures like directories and subdirectories.

#### **Why Use a Blob Store?**

Blob stores are crucial for data-intensive applications like YouTube, Netflix, and Facebook, which generate massive amounts of unstructured data daily. These applications require a storage solution that is:

* **Easily scalable**: To handle the ever-increasing volume of data.
* **Reliable**: Ensures data integrity and availability.
* **Highly available**: Data needs to be accessible at all times, replicated across multiple data centers and regions.

For instance, YouTube requires more than a petabyte of additional storage per day, with each video stored in multiple resolutions and replicated several times for availability.

#### **Design of a Blob Store System**

The design process of a blob store is divided into five key lessons and a quiz:

1. **Requirements**:
   * **Functional Requirements**:
     + Store and retrieve unstructured data.
     + Support large-scale data storage and access.
   * **Non-functional Requirements**:
     + Scalability.
     + Reliability.
     + High availability.
   * **Resource Estimation**: Estimate the storage, network bandwidth, and computational resources required for the blob store system.
2. **High-Level Design**:
   * **API Design**: Define the interfaces for interacting with the blob store.
   * **Component Design**: Detail the various components, such as storage nodes, metadata services, and client interfaces.
   * **Workflow**: Describe the data flow from ingestion to retrieval.
3. **Design Considerations**:
   * **Database Schema**: Organize and index metadata for efficient data retrieval.
   * **Partitioning Strategy**: Distribute data across multiple storage nodes to balance load and ensure scalability.
   * **Blob Indexing and Pagination**: Implement efficient indexing and pagination mechanisms for managing large datasets.
   * **Replication**: Ensure data redundancy and availability by replicating blobs across multiple locations.
4. **Evaluation**:
   * Assess the design based on the identified requirements.
   * Ensure that the blob store meets scalability, reliability, and availability criteria.
5. **Quiz**:
   * Test understanding of the blob store design concepts and principles.

#### **Conclusion**

A blob store is an essential component for managing unstructured data in modern, data-intensive applications. Its design requires careful consideration of scalability, reliability, and availability to handle the vast amounts of data generated daily by platforms like YouTube and Netflix.

This summary provides a detailed overview of the key elements involved in the design and evaluation of a blob store system, as outlined in the Grokking Modern System Design Interview for Engineers & Managers course.

### **Requirements of a Blob Store's Design - Summary**

#### **Functional Requirements**

1. **Create a Container**:
   * Users should be able to create containers to group blobs. Containers can help organize user-specific data or different types of blobs (e.g., videos vs. images).
   * A single blob store user can create multiple containers, and each container can have many blobs. Containers cannot be nested within each other.
2. **Put Data**:
   * Users should be able to upload blobs to the created containers.
3. **Get Data**:
   * The system should generate a URL for each uploaded blob, allowing users to access it later.
4. **Delete Data**:
   * Users should be able to delete a blob. The system should support retention times, allowing users to keep data for a specified period before deletion.
5. **List Blobs**:
   * Users should be able to list all blobs within a specific container.
6. **Delete a Container**:
   * Users should be able to delete a container and all the blobs inside it.
7. **List Containers**:
   * Users should be able to list all containers under a specific account.

#### **Non-functional Requirements**

1. **Availability**:
   * The system should be highly available to ensure that data can be accessed whenever needed.
2. **Durability**:
   * Once uploaded, data should not be lost unless explicitly deleted by the user.
3. **Scalability**:
   * The system should handle billions of blobs, allowing for seamless scalability as data volume grows.
4. **Throughput**:
   * The system should ensure high data throughput for efficient data transfers, especially for large media files.
5. **Reliability**:
   * The design should detect and recover from failures promptly, ensuring system reliability.
6. **Consistency**:
   * The system should provide strong consistency, ensuring that different users see the same view of a blob.

#### **Resource Estimation**

1. **Number of Servers Estimation**:
   * Based on assumptions about daily active users and server capabilities, the number of servers required is calculated. For example, with 5 million daily active users and each server handling 500 requests per second, approximately 10,000 servers are needed.
2. **Storage Estimation**:
   * Estimations are made for the total storage required per day. For instance, with 250,000 videos uploaded daily, each averaging 50 MB, and thumbnails averaging 20 KB, the total storage required is approximately 12.51 TB per day.
3. **Bandwidth Estimation**:
   * **Incoming Traffic**: Calculated based on the total data uploaded per day. For instance, 12.51 TB of data per day translates to a bandwidth requirement of 1.16 Gb/s.
   * **Outgoing Traffic**: Since the blob store is read-intensive, most bandwidth is required for outgoing traffic. With 5 million daily active users, each making 20 requests for 50 MB videos, the bandwidth requirement is approximately 462.96 Gb/s.

#### **Building Blocks**

1. **Rate Limiter**:
   * Controls user interactions with the system to prevent overload.
2. **Load Balancer**:
   * Distributes the request load across different servers to ensure efficient resource utilization and prevent bottlenecks.
3. **Database**:
   * Stores metadata information for blobs, facilitating efficient data management and retrieval.
4. **Monitoring**:
   * Inspects storage devices and tracks available space to add storage as needed, ensuring continuous system operation.

This detailed summary outlines the critical requirements and estimations involved in designing a blob store system, addressing both functional and non-functional aspects to ensure a scalable, reliable, and efficient storage solution for unstructured data.

### **Design of a Blob Store - Summary**

#### **High-Level Design**

The blob store system is composed of several key components:

* **Clients**: Initiate requests to the blob store.
* **Front-end Servers**: Receive and process client requests, storing blobs onto storage disks.
* **Storage Disks**: Store the actual blob data.

The client requests are handled by front-end servers, which then interact with storage disks to store the data.

#### **API Design**

The API design for the blob store includes the following operations:

1. **Create Container**:  
   python

listContainers(accountID)

* + Lists all containers under a user's account.
  + accountID: ID of the user.

#### **Detailed Design**

The detailed design includes the following components:

1. **Client**: User or program performing API functions.
2. **Rate Limiter**: Limits the number of requests based on user subscription or IP address to prevent overload.
3. **Load Balancer**: Distributes incoming traffic among servers, reroutes requests to different regions or data centers.
4. **Front-end Servers**: Forward user requests to storage servers.
5. **Data Nodes**: Hold the actual blob data, split into smaller chunks.
6. **Master Node**: Manages data nodes, stores information about storage paths and access privileges, assigns unique IDs to blobs, and determines storage space.
7. **Metadata Storage**: Distributed database storing account, container, and blob metadata.
8. **Monitoring Service**: Monitors data nodes and master node, alerts administrators in case of issues, tracks available space.
9. **Administrator**: Handles notifications from the monitoring service, conducts routine checks for reliability.

#### **Workflow**

1. **Writing a Blob**:
   * The client requests to upload a blob, which is processed through the rate limiter and load balancer to a front-end server.
   * The front-end server contacts the master node for data node assignment.
   * The master node assigns a unique ID, splits the blob into chunks, and determines storage allocation.
   * The front-end servers write chunks to the data nodes, replicate chunks for redundancy, and store blob metadata.
   * The client receives the fully qualified path of the blob.
2. **Reading a Blob**:
   * The client requests to read a blob, and the front-end server asks the master node for metadata.
   * The master node checks access privileges, retrieves chunk mappings, and returns this information to the client.
   * The client reads the chunk data from the data nodes.
3. **Deleting a Blob**:
   * Upon a delete request, the master node marks the blob as deleted in metadata and frees space during garbage collection.

#### **Considerations and Points to Ponder**

* Concurrent write operations with the same name are managed by the master node.
* Handling stale information due to data node movements involves updating cached metadata.
* Addressing single points of failure in the master node involves redundancy and failover mechanisms.

### **Design Considerations of a Blob Store - Summary**

#### **Introduction**

Blob stores are crucial for managing unstructured data, such as multimedia files, in data-intensive applications like YouTube and Netflix. Key design considerations include efficient storage, retrieval, replication, indexing, and deletion of blobs to ensure scalability, reliability, and high availability.

#### **Key Design Considerations**

1. **Blob Metadata**
   * **Purpose**: Metadata ensures efficient storage and retrieval of blobs.
   * **Details**:
     + When a user uploads a blob, it is split into chunks stored on different data nodes.
     + Metadata includes chunk IDs, data node IDs, and replica IDs for redundancy.
     + Example: A 128 MB blob is split into two 64 MB chunks, with each chunk having multiple replicas for reliability​
     + ​.

#### **Conclusion**

These design considerations ensure that a blob store can efficiently handle large volumes of unstructured data while maintaining scalability, reliability, and high availability. By addressing metadata management, data partitioning, indexing, pagination, replication, garbage collection, streaming, and caching, the blob store can meet the demanding requirements of modern data-intensive applications.

5. Evaluation of a Blob Store's Design.pdf

PDF

As an expert in system design, summarize the following document in great detail.

ChatGPT

### **Evaluation of a Blob Store's Design - Summary**

#### **Introduction**

The evaluation of a blob store design focuses on key non-functional requirements: availability, durability, scalability, throughput, reliability, and consistency. These aspects ensure the blob store meets the demands of modern, data-intensive applications.

#### **Availability**

* **Replication Strategy**: The system maintains four replicas for each blob, distributing request loads across replicas to ensure availability even if one node fails.
* **Data Center and Region Failures**: The replication strategy is designed to handle complete data center failures and regional unavailability due to disasters.
* **Monitoring Service**: Ensures that replicas are always available by making additional copies if the number of failed replicas exceeds a threshold.
* **Write Requests**: Data is replicated within the cluster in a fault-tolerant manner, and the system quickly responds to write requests.
* **Master Node Backup**: The master node’s state is backed up, and a new instance can be started from the saved state in case of failure​
* ​.

#### **Durability**

* **Replication and Monitoring**: Synchronously replicating data within a storage cluster ensures durability. If data is lost at one node, it can be recovered from another.
* **Disk Monitoring**: The monitoring service alerts administrators to replace failed disks and instructs the master node to copy content from failed disks to new ones​
* ​.

#### **Scalability**

* **Partitioning and Chunking**: Blobs are split into smaller chunks and partitioned across different servers, allowing the system to handle billions of blob requests. This partitioning also facilitates automatic load balancing.
* **Horizontal Scaling**: The system can scale horizontally by adding more data nodes as storage needs increase. However, the master node may become a bottleneck, handling up to 10,000 queries per second (QPS)​
* ​.

#### **Throughput**

* **Parallel Fetching**: Saving blob chunks on different data nodes allows parallel fetching, distributing requests for a blob across multiple machines to achieve high throughput.
* **Caching**: Implemented at various layers—client-side, front-end servers, and master node—caching improves throughput and reduces latency​
* ​.

#### **Reliability**

* **Monitoring Techniques**: The heartbeat protocol keeps the master node updated on the status of data nodes, ensuring only reliable nodes are used. The failure of a node triggers the master node to request an additional replica node.
* **Administrator Alerts**: Monitoring services alert administrators to replace faulty hardware, such as failed disks or broken network links. The master node also tracks available disk space, alerting administrators to add new disks as needed​
* ​.

#### **Consistency**

* **Synchronous Replication**: Disk data blocks are synchronously replicated within a storage cluster upon a write request, ensuring strong consistency. Subsequent read requests are served from the same storage cluster until the data is replicated to other clusters.
* **Asynchronous Replication**: After initial replication, blobs are asynchronously replicated across different data centers and regions to ensure availability​
* ​.

#### **Conclusion**

The blob store is designed to handle large-sized and unstructured data, making it suitable for applications like YouTube, Facebook, Instagram, and Twitter. The evaluation demonstrates that the design meets the non-functional requirements of availability, durability, scalability, throughput, reliability, and consistency​

​.

This detailed evaluation ensures that the blob store can effectively support modern data-intensive applications by providing a reliable and efficient storage solution for unstructured data.

ChatGPT can make mistakes. Check important info.

ChatGPT

createContainer(containerName)

* Creates a new container under the logged-in account.
* containerName: Unique name within the storage account.

**Upload Blobs**:  
python

putBlob(containerPath, blobName, data)

* Uploads data as blobs into a specified container.
* containerPath: Path of the container.
* blobName: Unique name within the container.
* data: File to be uploaded.

**Download Blobs**:  
python

getBlob(blobPath)

* Retrieves a blob by its unique path.
* blobPath: Fully qualified path of the blob.

**Delete Blob**:  
python

deleteBlob(blobPath)

* Marks a blob for deletion.
* blobPath: Path of the blob to be deleted.

**List Blobs**:  
python

listBlobs(containerPath)

* Lists all blobs within a specified container.
* containerPath: Path to the container.

**Delete Container**:  
python

deleteContainer(containerPath)

* Marks a container and all its blobs for deletion.
* containerPath: Path to the container.

**List Containers**:  
python

* + ​.

**Partitioning Data**

* **Purpose**: Efficiently manage and access large volumes of blobs.
* **Details**:
  + Blobs are partitioned based on a combination of account ID, container ID, and blob ID.
  + This co-locates blobs for a single user on the same partition server, enhancing performance.
  + Partition mappings are maintained by the master node and stored in distributed metadata storage​
  + ​.

**Blob Indexing**

* **Purpose**: Efficiently search and retrieve blobs.
* **Details**:
  + Indexes are created using key-value tags like container name, blob name, upload date, etc.
  + A blob indexing engine reads new tags, indexes them, and makes them searchable​
  + ​.

**Pagination for Listing**

* **Purpose**: Handle large lists of blobs efficiently.
* **Details**:
  + Pagination divides long lists of blobs into manageable parts.
  + A continuation token is used as a pointer for subsequent queries.
  + Example: A list of 2,000 blobs can be paginated to return five results at a time​
  + ​.

**Replication**

* **Purpose**: Ensure data availability and strong consistency.
* **Details**:
  + **Synchronous Replication**: Within a storage cluster to maintain data durability and consistency.
  + **Asynchronous Replication**: Across different data centers and regions to protect against local and regional failures.
  + Example: Four copies of a blob are maintained—one local, one in another data center in the same region, and one in a different region​
  + ​.

**Garbage Collection while Deleting a Blob**

* **Purpose**: Efficiently manage storage space after blob deletion.
* **Details**:
  + Blobs are marked as "DELETED" in metadata and removed later by a garbage collector.
  + This approach ensures real-time latency optimization for user delete requests​
  + ​.

**Streaming a File**

* **Purpose**: Facilitate interactive user experience by streaming large files in chunks.
* **Details**:
  + Files are read in defined byte ranges, allowing chunk-by-chunk streaming.
  + Example: Reading the first X bytes starting from byte 0, then the next X bytes, and so on​
  + ​.

**Caching the Blob Store**

* **Purpose**: Improve response time and throughput.
* **Details**:
  + Caching occurs at multiple levels: client-side metadata caching, front-end server partition map caching, and master node chunk caching.
  + Caching reduces disk I/O and enhances performance.
  + CDN caching is used for publicly accessible blobs with TTL defined by the origin server​