ID2221 - Data Intensive Computing - Review Questions 1

Deepak Shankar Reethika Ambatipudi
deepak | reethika @kth.se

September 8, 2023

Q1. Explain how a file region can be left in an undefined state in GFS?

In GFS, a file region can be left in an undefined state under certain conditions, primarily due to the distributed and fault-tolerant nature of the system. Here are some scenarios in which this can happen:

1. Partial Writes: GFS allows clients to write data to a file in multiple, potentially overlapping, chunks. When a client writes data to a chunk, it sends the data to a primary chunk server responsible for that chunk. If a write operation to a chunk is interrupted or fails before completion (e.g., due to a network failure or client crash), the data may be partially written to the chunk. In such cases, the file region covered by the partial write will be in an undefined state, containing a mix of old and new data.

2. Concurrent Writes: GFS supports multiple clients writing to the same file concurrently. When two or more clients write to overlapping regions of a file simultaneously, the resulting data in the overlapping region may be inconsistent and undefined. GFS does not provide built-in mechanisms for coordinating concurrent writes between clients.

3. Chunk Server Failures: GFS replicates chunks across multiple chunk servers for fault tolerance. If one or more chunk servers hosting a particular chunk fail and there are not enough replicas available to maintain data integrity, then the data in that chunk may become inaccessible or corrupted, leaving a file region in an undefined state.

4. Master Server Failures: The GFS master server maintains metadata information about files, including the location of chunks. If the master server fails or becomes unreachable, clients may not be able to determine the location of the chunks for a particular file. This can result in undefined states when clients cannot access or modify the file's data.

Q2. Briefly explain the read operation in GFS? What's the role of the primary node?

In Google File System (GFS), the read operation involves retrieving data from a file's chunk and delivering it to the client requesting the read. Here's a brief overview of the read operation in GFS:

1. Client Request: When a client application wants to read data from a file in GFS, it contacts the GFS master server to obtain metadata about the file, including the location of the relevant data chunks.

2. Chunk Location: The GFS master server provides the client with information about the chunk servers that hold the data chunks needed for the read operation. This information includes the locations of the primary and replica chunk servers for each chunk.

3. Primary Node: The primary node, also known as the primary chunk server, is responsible for coordinating read operations for a specific chunk of data. The primary node plays a crucial role in the read operation in GFS. Its responsibilities include:

   - Handling Read Requests: The client contacts the primary node directly to request the data chunk. The primary node is responsible for processing the read request and sending the requested data back to the client.

   - Data Consistency: The primary node ensures data consistency by coordinating any concurrent writes and maintaining a consistent view of the data chunk. It keeps track of which version of the data is the most up-to-date.

   - Replica Coordination: If the primary node detects that one or more replicas of the chunk have become stale or inaccessible (e.g., due to chunk server failures), it coordinates the process of creating new replicas or choosing a different replica for reading.

4. Data Retrieval: Once the client contacts the primary node, the primary node retrieves the requested data from its local storage and sends it back to the client. The client can then use the retrieved data for its intended purpose.

Overall, the primary node in GFS ensures that read operations are coordinated, consistent, and reliable by managing access to the data chunk and coordinating with replica chunk servers when necessary. If the primary node becomes unavailable, the client can contact one of the replica chunk servers to perform the read operation, but the primary node typically serves as the primary point of access for data retrieval to maintain consistency.

Q3. Using one example show that in the CAP theorem, if we have Consistency and Partition Tolerance, we cannot provide Availability at the same time.

The CAP theorem, formulated by computer scientist Eric Brewer, describes the fundamental trade-offs that distributed systems must make between Consistency, Availability, and Partition Tolerance. According to the theorem, a distributed system can have at most two out of the three properties simultaneously. Let's illustrate this using an example:

Suppose we are designing a distributed database system for an e-commerce platform where customers can place orders. In this system:

1. Consistency: Ensures that all nodes in the system see the same data at the same time. In other words, when a write operation is completed, all subsequent read operations will return the most recent written value.
2. Availability: Guarantees that every request (read or write) made to the system receives a response without any errors, even if it cannot guarantee that the response contains the most up-to-date data.
3. Partition Tolerance: Ensures that the system continues to operate even in the presence of network partitions or communication failures between nodes. Partition tolerance is crucial for distributed systems to handle network issues gracefully.

If we prioritize Consistency and Partition Tolerance, This means that we want all nodes to have the same, up-to-date data, and the system should continue to operate even when network partitions occur.

Imagine a network partition temporarily isolates a subset of your database nodes from the rest of the system. In this case, the isolated nodes cannot communicate with the others, and they cannot process new updates. However, since we've chosen strong Consistency, no new data can be written or read on the isolated nodes until they reconnect to the network and synchronize with the others. This ensures that data remains consistent across the entire system.

While Consistency and Partition Tolerance are maintained, Availability is compromised because the system cannot respond to read or write requests on the isolated nodes during the network partition.

In summary, when we prioritize both Consistency and Partition Tolerance, we may experience reduced Availability during network partitions because maintaining strong Consistency requires waiting for all nodes to agree on the data's state, which can be delayed by partitioned nodes. This illustrates the trade-off described by the CAP theorem: you can't have all three properties (Consistency, Availability, and Partition Tolerance) simultaneously in a distributed system when network partitions occur.

Q4. Explain how the consistent hashing works.

Consistent hashing is a technique used in distributed systems and distributed databases to efficiently distribute and manage data across multiple nodes or servers while allowing for dynamic scaling and fault tolerance. It addresses the challenge of load balancing and data distribution in a distributed environment. Here's an explanation of how consistent hashing works:

1. Data Distribution: In a distributed system, data is typically divided into smaller units, often referred to as shards or partitions. Each shard is associated with a unique identifier, such as a key or hash value. The goal is to distribute these shards across a set of nodes or servers in a way that is both load-balanced and resilient to node failures.

2. Node Assignment: Consistent hashing assigns each node in the system a position on a virtual ring or circle, which is typically represented as a numerical range, such as 0 to 2 power32. Each node is mapped to one or more positions on this ring using a hash function. The hash function maps a node's identifier (e.g., its IP address or name) to a point on the ring.

3. Shard Mapping: To determine which node should be responsible for a given shard, consistent hashing applies the same hash function to the shard's identifier (e.g., a unique key or name). This maps the shard's identifier to a point on the ring as well.

4. Node Lookup: To find the node responsible for a specific shard, the system looks for the first node encountered in a clockwise direction on the ring from the position of the shard's identifier. This node becomes the shard's owner.

Overall, consistent hashing is a powerful technique for managing data distribution in distributed systems, ensuring load balancing and fault tolerance while maintaining predictability and scalability. It's widely used in distributed databases, content delivery networks (CDNs), and other distributed computing scenarios.

Q5. Explain the finding process of tablets in BigTable.

In Google Bigtable, a distributed, scalable, and high-performance NoSQL database, the process of finding a tablet involves a hierarchical structure and a combination of metadata storage mechanisms to efficiently locate and access data tablets. Here's an explanation of the finding process for tablets in Bigtable:

1. Three-Level Hierarchy: - Bigtable organizes its data into a three-level hierarchy: Tables, Tablets, and Rows. - Tables contain the actual data and are divided into multiple tablets for efficient distribution. - Rows are individual records within tablets.

2. Root Tablet Location: - At the top of the hierarchy is a special tablet known as the "root tablet." It contains information about the location of all tablets within the system. - The location of the root tablet is stored in a distributed lock service called Chubby, which ensures that only one process can access the root tablet's location information at a time.

3. METADATA Table: - The root tablet contains metadata about all the tablets in the system, including their locations. - Specifically, the root tablet holds information about a special system table known as the "METADATA table." - The METADATA table is a system-level table that stores metadata about all the user-defined tables and their tablets in the Bigtable instance. - Each row in the METADATA table corresponds to a specific tablet. - The METADATA table contains information about the tablets' location, such as the tablet's start and end key ranges and the location of the tablet servers responsible for that tablet.

4. Client Library Caching: - To optimize the tablet-finding process and reduce the need for frequent lookups, the Bigtable client library employs a caching mechanism. - The client library caches tablet locations, including the information from the METADATA table. - When a client application wants to read or write data to a specific tablet, it consults its local cache to quickly determine the tablet's location without needing to make network requests for every operation.

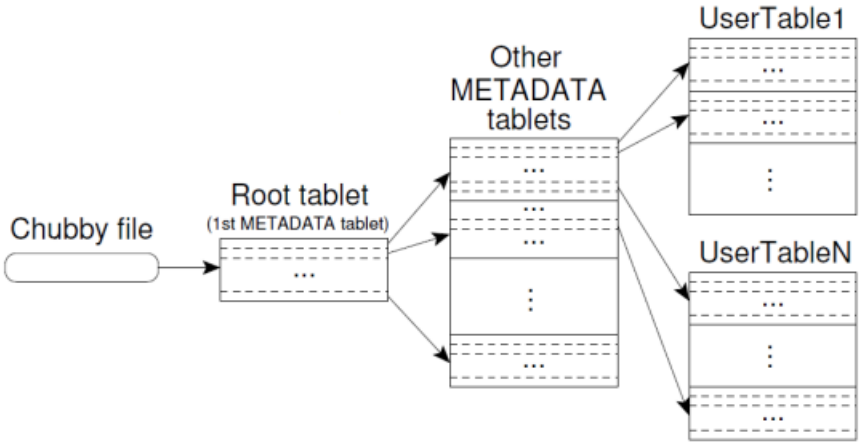

Figure 1: Finding process of tablets in Big Table

In summary, the tablet-finding process in Bigtable relies on a hierarchical structure, starting with a root tablet that contains information about all tablets in the system. The metadata for individual tablets is stored in a

system-level METADATA table. Client applications utilize a caching mechanism to efficiently locate tablets, reducing the need for frequent lookups and network requests, which is crucial for achieving high performance and scalability in Bigtable.