Designing and Implementing a Simplified Google Bigtable: Architecture, Trade-offs, and Insights

Yuzhe Ruan
Yale University

Chaoxu Wu
Yale University

1 Introduction

Google Bigtable stands out as a highly scalable, distributed storage solution that has underpinned numerous services within Google, including Search, Maps, and Gmail [2]. Inspired by Bigtable's success, this project aims to develop a simplified version that captures its core functionalities while maintaining manageability and educational value.

2 System Design and Implementation

2.1 Distributed System In Action

We explored the following aspects of the distributed system within this project

1. **Sharding Management:** A maximum table count (max_table_cnt) is established for each tablet server. In the event that tablet A is handling more than the maximum table count, signifying that it is overloaded, it will notify the master. The master will then select the currently least loaded tablet server B and instruct A to transfer a portion of its data to B
2. **Fault Tolerance:** Tablet server crashes are very common in practical scenarios. To address this issue, tablets continuously persist (snapshot) metadata (columns and rows) whenever updates occur. Actual data will also be persistent by LevelDB as well. When tablet server A crashes, the master will pick another available tablet B. Tablet B will then access the Google File System (in our project, LevelDB and files stored in different paths on the local disk to mimic the distributed storage) and commence the recovery process
3. **Service Discovery:** The master server will be initiated first. Subsequently, each tablet server will register with the master server upon startup. In this way, the master is enabled to maintain local caches and later provide the tablet address to the client for CRUD request routing. Additionally, the master will periodically send heartbeats to check whether any of the tablet servers have ceased to function.

2.2 Module Overview

Our simplified Bigtable implementation comprises three primary components:

1. **Master Server:** Oversees metadata management, tablet assignments, and coordination among tablet servers.
2. **Tablet Servers:** Handle data storage, CRUD operations, and respond to client requests.
3. **Client Library:** Provides an interface for applications to interact with the storage system, abstracting the underlying complexities.

2.2.1 Key Assumptions

1. The master server never crashes, ensuring that tablet servers don't need to be concerned about losing the address of the tablet. Thus, we can simply provide the master address to the tablet right from the start. In practice, this can be accomplished by converting the single master server into a Raft cluster.
2. No network problems, all gRPC calls are guaranteed to succeed. We did not incorporate any gRPC retry mechanism into our project.
3. Our Bigtable implementation also supports the multiple version read/write as described in the paper. The maximum number of versions is 3. This implies that you can access at most the last 3 versions of the data, while any updates prior to those will not be preserved.

4. LevelDB, along with files stored in different paths on the local disk, is used to act as a pseudo Google File System. The files are stored in a path format such as /<TabletserverAddress>/<tableName>/<...>. For instance, /localhost_9091/customers/..... When the CreateTable operation is executed, this specific path is created. When the DeleteTable is called, all the files in the path are removed. In the event that tablet A crashes and tablet B is responsible for recovering it, all the files under /localhost_tabletA are transferred to /localhost_tabletB.

2.2.2 How Modules Collaborate

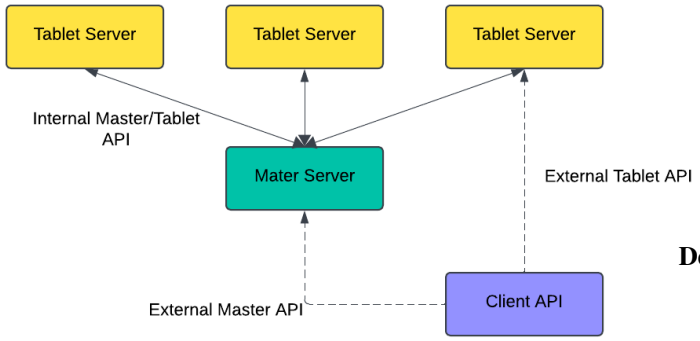

Figure 1: System Architecture Overview

1. External-API: There are six interfaces provided to the exterior client: CreateTable, DeleteTable, Read, Write, Delete, GetTabletLocation

   (a) CreateTable/DeleteTable: The client is required to call CreateTable first before performing any read, write, or delete operations. The client sends a CreateTable Request to the master. The master then validates the parameters and designates a tablet to create a table within the data storage. The DeleteTable operation follows a similar process flow.

   (b) Read, Write, Delete, GetTabletLocation: When the client intends to Read/Write/Delete, it will send a GetTabletLocation request to the server to obtain the address of the tablet responsible for the corresponding table. It will then connect with the tablet to perform the Read/Write/Delete operation.

2. Internal-API: Internal APIs are primarily responsible for sharding. Whenever a tablet A is managing more than the maximum number of tables, it will notify the master. The master will pick the tablet server B that is currently the least loaded and direct tablet server A to transfer a part of its data to B. Additionally, when the server detects a tablet crash via heartbeat, it will select and request another server to recover the tablet data.

2.3 Master Server

The Master Server is the linchpin of the system, responsible for:

- **Table Management:** Creation and deletion of tables, including the specification of column families.
- **Tablet Assignment:** Distributing data ranges (tablets) across available tablet servers.
- **Heartbeat Monitoring:** Ensuring the health and availability of tablet servers through periodic heartbeats.
- **Shard Management:** Handling dynamic data partitioning to balance load and optimize performance.

Design Choices and Trade-offs

- **Tablet Assignment Strategy:** We opted for a least-loaded approach, assigning new tablets to the tablet server with the fewest active tablets. This promotes load balancing but may introduce latency in tablet reassignment during high churn scenarios.
- **Heartbeat Mechanism:** Implemented a simple heartbeat protocol with timeouts to detect and handle server failures. While effective for our scale, more sophisticated mechanisms (e.g., gossip protocols) might be necessary for larger deployments to reduce overhead.

2.4 Tablet Servers

Tablet Servers are responsible for storing data, processing client requests, and maintaining data integrity. Each tablet server manages multiple tablets, each corresponding to a range of row keys within a table.

Core Functionalities

- **CRUD Operations and Data Storage:** Utilizing LevelDB as the underlying storage engine for efficient key-value storage
- **Shard Handling:** Migrating data to other tablet servers as instructed by the Master Server.

Design Choices and Trade-offs

- **Storage Engine Selection:** LevelDB was chosen for its simplicity and performance in handling key-value pairs. While LevelDB offers efficient storage, it lacks certain features like built-in replication and distributed consensus, which would require additional layers for a production-grade system.
- **Concurrency Control:** Employed Go's concurrency primitives (goroutines and mutexes) to handle simultaneous requests. This approach ensures thread safety but may require careful management to avoid contention in high-throughput scenarios.

2.5 Client Library

The Client Library abstracts the complexities of interacting with the distributed system, providing intuitive methods for table creation, data manipulation, and connection management.

Key Features

- **Table Caching:** Maintains a local cache of tablet ranges and their corresponding servers to minimize latency in request routing.

Design Choices and Trade-offs

- **Caching Strategy:** Implemented a simple range-based cache with binary search for efficient lookups. While effective for our use case, more advanced caching mechanisms (e.g., consistent hashing) could enhance performance and scalability.
- **Error Handling:** Emphasized robust error handling to gracefully manage network failures and server unavailability. However, in scenarios with frequent failures, the current retry logic might require enhancements to prevent cascading failures.

3 Related Work

Our project draws inspiration from Google Bigtable, which has significantly influenced the design of modern distributed storage systems [2]. Similar systems include:

- **Apache HBase:** An open-source implementation modeled directly after Bigtable, providing a distributed, scalable, and versioned key-value store on top of Hadoop's HDFS [4].
- **Apache Cassandra:** Designed for high availability and scalability, Cassandra employs a decentralized architecture and supports multi-datacenter replication [5].
- **Amazon DynamoDB:** A fully managed NoSQL database service that offers seamless scalability and high performance, inspired by the Dynamo system [3].
- **ScyllaDB:** A high-performance NoSQL database compatible with Cassandra's API, optimized for modern hardware and multi-core architectures [1].

Our implementation distinguishes itself by focusing on simplicity and educational value, providing a foundational understanding of distributed storage principles without the overhead of enterprise-grade features.

4 Conclusion

This writeup delineates the design and implementation of a simplified version of Google Bigtable, emphasizing core functionalities essential for a distributed, scalable, and reliable storage system. Through thoughtful architectural decisions and an exploration of design trade-offs, we achieved a balance between functionality and simplicity. Our work serves as both a learning tool and a foundation for further exploration into distributed storage systems, offering insights into the complexities and considerations inherent in such endeavors.

Future work may involve integrating advanced features such as replication, stronger consistency models, and more sophisticated shard management algorithms. Additionally, performance evaluations and benchmarking against existing systems could provide deeper insights into the strengths and limitations of our approach.

References

[1] Steve Cayzer, Andrew Jasienski, Luise V Lakshmanan, Sanjay Ghemawat, and Michael J Franklin. Scylladb: A high-performance nosql database compatible with cassandra. In *Proceedings of the VLDB Endowment*, volume 8, pages 1783–1786. VLDB Endowment, 2015.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chen, and Ralph E Gruber. Bigtable: A distributed storage system for structured data. In *ACM Transactions on Computer Systems (TOCS)*, volume 26, pages 1–26. ACM, 2008.

[3] Giuseppe DeCandia, Deniz Hastorun, Margaret Madden, Aditya Paul, John Rabkin, Ion Stoica, and George Kakulapati. Dynamo: amazon's highly available key-value

store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[4] Lars George, Jay Kreps, Jaideep Rao, and Jennifer Thoburn. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[5] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. In *Proceedings of the 2010 ACM SIGOPS Symposium on Operating Systems Principles*, pages 15–28. ACM, 2010.