CPSC 426 Final Project
JKZDB: Distributed Sharded Database with 2PC

Ross Johnson
Yale University

Matthew Zhang
Yale University

Surtaz Khan
Yale University

Abstract

JKZDB is a hash-sharded database over BoltDB that utilizes two-phase commit protocol to ensure atomic updates. JKZDB can be interacted with using a RESTful API implemented in Go that provides common features typically associated with bank accounts.

1 Introduction

For our final project, we built a sharded database that we call JKZDB. This database uses BoltDB as its underlying storage engine. We then built a custom RESTful API over JKZDB using Go Fiber. JKZDB adds global indexing functionality over the B+ tree implementation in BoltDB with hash sharding, and to ensure that updates to multiple shards are atomic, the two-phase commit (2PC) protocol is used. Two-phase commit protocol is a distributed algorithm that coordinates all the processes that are involved in a distributed atomic transaction on whether to commit or abort the transaction. The database schema in JKZDB is currently modeled after a banking model, storing information such as email, account balance, age, the date the account was opened at, and the last date the account was active. JKZDB supports account creations, updates to emails, deposits and withdrawals, and transaction between users. Email also serves as an index key along with the existing primary integer key in the database.

2 BoltDB as a Storage Engine

BoltDB is an embedded ACID database that stores key/-value pairs. It is written in Go and is meant to be used as a low-level piece of functionality. Hence, its implementation focuses on simplicity and serves as the ideal base for our database. Our implementation creates several underlying instances of BoltDB that are connected to different ports to act as the separate shards for JKZDB. This can be found in `/scripts/run_servers.go`. We then created functions that support standard CRUD operations by acting as wrappers over standard BoltDB functions such as `CreateBucket`, `Bucket`, `Delete`, and `Put` in `/db/db.go`. These sharded instances of JKZDB can be cleaned up by running `/scripts/stop_servers.go`.

3 Indexing and Sharding

Each bucket in BoltDB is a collection of unique keys that are associated with their corresponding values. These buckets are represented by using a B+ tree. In JKDZDB, we use hash sharding for our global indexing. This is done by the coordinator in `/coordinator/server/2pc_helpers.go` and later used by the API request handlers to interact with the database properly. In terms of indexing, we allow requests to JKZDB to look up data using both the primary key and the secondary key. Our primary key is an integer representing the account id and our secondary key is a user's email address. Based on which key is being used, we format the query using `CreateQuery` in `/coordinator/server/handlers.go` to create a common format of *index:key*. This resulting string is then hashed by the coordinator to determine which shards data should be placed on and located for future use. As a result, users are able to index into our database properly with either the primary key id or the secondary key email address. If the email address is used, we are able to retrieve the primary key internally by storing a mapping of *email address:id* on our shards. This mapping is stored upon the initial creation of our users. On all API requests, we check the type of index being used and fetch the necessary data that the JKZDB gRPC functions require.

4 Database Schema

Schema					
id	email	balance	age	opened	last used
int	str	int	int	stamp	stamp

Our database schema is intended to mock a user's account in a bank. This includes information about the user

such as their email address and age. Each user has an account balance that they can deposit to, withdraw from, and conduct transactions with (sending out money to another user or receiving money from another user). We also track the time at which the account was opened and when it was last used. Our data is stored in JSON format, and this model can be found in `/models/user.go`. To maintain simplicity, we decided to use the `int64` type for the account balance and the timestamps are measured as Unix timestamps. The following is an example of how this data might look like.

```
{
  "id": "Ross Johnson",
  "email": "rossjohnson@yale.edu",
  "balance": 1024,
  "age": 21,
  "account_opened": 1620776412,
  "last_used": 1652287216,
}
```

When any changes must be made to the data, we first unserialize it using `json.Unmarshal`. Then we can easily make the desired update to the requested field since the data has been transformed into an instance of our user model. Afterwards, we serialize the data again into JSON using `json.Marshal` and update the entry in the underlying database. By leveraging the serialization of JSON data, we were able to avoid having to write more fine-grained update functions over BoltDB.

5 Two-Phase Commit (2PC) Protocol

Two-phase commit is a mechanism by which to achieve atomicity in committing transactions in a distributed system, as opposed to related consensus algorithms that seek to achieve agreement on a particular value. When a client wishes to open a transaction on multiple database nodes, the initial request is first routed to a coordinator, and the coordinator sends a `PrepareEntry` message to all database nodes. [3] Each database node replies with a success or failure response to this `PrepareEntry` indicating its intention to commit or not commit the message back to the coordinator. The transaction coordinator waits until it receives all `PrepareEntry` responses from all nodes. In the case that the transaction coordinator receives n success messages from n nodes, it subsequently sends a `CommitEntry` message to all nodes instructing them to now actually commit the message and release their locks. In the case that n success responses are not received, the coordinator views this as a commit failure and aborts the commit of the message. [3]

In this project, we incorporate two-phase commit into JKZDB by creating a JKZDBServer that implements the required functions (`SetEntryPrepare`, `SetEntryCommit`, `SetEntryAbort`). These functions can be used by the coordinator by creating a `JKZDBClient` instance that relies on the gRPC framework to call the two-phase commit operations. To ensure idempotency, we also generate a unique value in the `JKZDBClient` and send it alongside the `SetEntryPrepare` requests. The `JKZDBServer` then stores the key to use for bookkeeping. When the `SetEntryCommit` request is processed, we make sure that the idempotency key matches so that the correct operation is only executed once.

Based on our design architecture, we needed to include batching APIs (`SendPrepareBatchRPC` and `SendCommittBatchRPC`) in our model. In a given update query, there may be situations where two different keys both exist on the same shard. However, based on two-phase commit transaction handling, a given `PrepareEntry` or `CommitEntry` message necessarily acquires system resources. Therefore, a simultaneous update of multiple components on the same shard would lead to deadlock. Our batching API combines these separate requests into a single transaction that can occur under one lock on a given shard, thus avoiding deadlock on both the prepare or commit phase in our protocol. One potential disadvantage of two-phase commit is that the transaction coordinator is a single point of failure. Specifically, if the coordinator fails after the first preparation phase, the database nodes remain uncertain about whether to abort or commit a message. It is only after the coordinator recovers that the database nodes are able to move forward. Thus, until then, the database nodes must remain locked and results in a significant decline in performance.

6 API Endpoints

Here is the list of the API endpoints that JKZDB supports.

- **GET** `/api/user?index=id&key=<pk>`
- **GET** `/api/user?index=email&key=<email>`
- **POST** `/api/user`
- **POST** `/transaction?index=id&from=<from_pk>&to=<to_pk>&amount=<amount>`
- **POST** `/transaction?index=email&from=<from_email>&to=<to_email>&amount=<amount>`
- **PUT** `/api/email?old-email=<old>&new-email=<new>`
- **PUT** `/api/withdraw?amount=<amount>&index=id&key=<pk>`
- **PUT** `/api/withdraw?amount=<amount>&index=email&key=<email>`
- **PUT** `/api/deposit?amount=<amount>&index=id&key=<pk>`
- **PUT** `/api/deposit?amount=<amount>&index=email&key=<email>`
- **DELETE** `/api/user?index=id&key=<pk>`

- **DELETE** `/api/user?index=email&key=<email>`

In the initial POST request to `/api/user`, JSON data representing the user's email, age, and account balance should be included. Users can be retrieved by id or email using the GET requests, have their emails or balances updated using the PUT requests, conduct transactions with the POST requests, and delete their data using the DELETE requests.

7 Results and Discussion

To test our implementation of JKZDB, we rely on an external API platform such as Postman or Thunder Client for Visual Studio Code. We can start the different shards of JKZDB by running `/scripts/run_servers.go`. We start the API listener by running `/coordinator/server.go`. Then we can simply send API requests to the provided host URL and observe the changes.

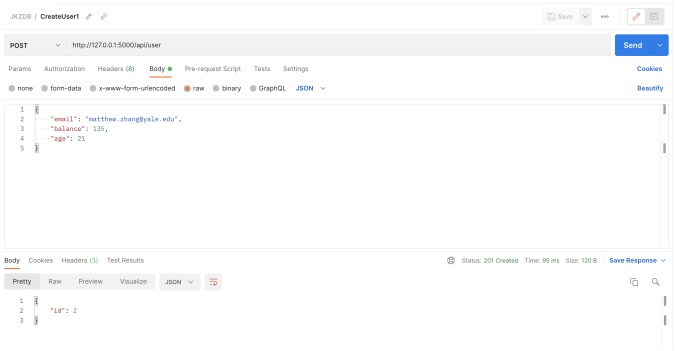

Figure 1: Example POST request to create a user.

Through log statements, we were able to verify that Postman requests succeeded for regular and batched two-phase commit operations. This indicates that the coordinator communicated with the shards correctly and updates to JKZDB were atomic despite data being spread around multiple shards. We were able to create users, update their email and balances, and conduct transactions. We also encountered expected errors for invalid operations (withdrawing or sending money that exceeds the balance). If we updated a user's email, this change was reflected on all the shards. The original email mapping is deleted, the user data is updated, and a new mapping from the current email to the original primary key id is created. We observed in the output that these operations occur successfully on different shards, and on the same shard with the batch RPCs. Moreover, we saw the same results were returned when indexing with id or email, which is the intended behavior. For more information on how we tested JKZDB, please refer to the demo video provided in the submission.

8 Future Work and Areas of Improvement

Moving forward, we see many avenues for future work as well as improvements to existing functionality. One of our stretch goals in our proposed plan was range sharding so that we can expand the types of queries our database can support. Range sharding would lead to faster and more flexible data querying. In general, we could attempt to generalize our data model and the provided functionality so that we can expand to different types of queries such as search and sort requests for all types of data. A more advanced and generalized database would help serve more use cases.

Another area of improvement is related to striping. We could stripe our shard to have more concurrency with less shards (i.e. more than one bucket per shard and having a lock for each of those). Overall, we could utilize more fine-grained locking to reduce potential latency concerns. Moreover, we could also have a bucket per indexed key. This would allow our keys to be shorter, which will help JKZDB use less memory. There will be less IOs when searching for a particular key because grabbing the correct bucket to begin with will already exclude all unrelated keys on a particular shard.

To verify scalability goals, our immediate next steps would also include creating a robust stress testing mechanism that tests all potential failure states, such as transaction coordinator failure, DB failure, elevated volume of concurrent update requests, and malicious network delays and corrupted messages. To demonstrate this, we would send plenty of reads and writes, and ensure that after every write, the values are consistent between secondary key data and primary key data. While stress testing was in our plans, we experienced delays in our timeline due to debugging other features which prevented us from adding it to the project as of now.

As more failure states are discovered through stress testing, more robust error handling and status codes could be introduced. This may enable new features such as dynamic timeout modulation and exponential backoff for intelligent retries. Our architecture may even benefit from more significant changes like the introduction of a cache for efficient retrieval, especially as our sample model of a transaction-based banking system is more generalized.

While two-phase commit certainly has limitations, especially around coordinator failure and corrupted state, many large-scale systems still use two-phase commit. In future work, we may benefit from combining two-phase commit with consensus solutions seen in systems like Kafka, like leader election and replication, to maintain resiliency in failure states.

Ultimately, given the scope of a class final project and the limited time, we still have many features to work on and

implement. Eventually, as we continue to work on improving JKZDB, we can also work on deploying the service through Docker and Kubernetes.

9 Related Work

There is literature on potential improvements on two-phase commit, such as how Spanner implements 2PC over Paxos to alleviate bottleneck problems and mitigates availability problems across zones. Having a client drive two-phase commit further avoids sending data twice across wide-area links. [1]

Other literature reveals how standard 2PC requires multiple messages in multiple phases, which in certain cases can negatively affect system throughput, increase response time, and cause substantial delays. However, optimizations can be made regarding presumptions about when a given entry is prepared or committed given 2PC invariants, leading to significantly less log additions with the expense of storing crash-related information in-memory forever. [2]

10 Acknowledgements

We would like to thank our professor Richard Yang and our lecturers from Facebook, Scott Pruett and Xiao Shi, for guiding our knowledge of distributed systems this semester.

References

[1] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[2] Butler W. Lampson and David B. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, page 630–640, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[3] Antoni Wolski and Jari Veijalainen. 2pc agent method: achieving serializability in presence of failures in a heterogeneous multidatabase. pages 321 – 330, 04 1990.