Distributed SQLite - Replicated SQLite Service powered by Raft

Arun Rajan, Ethan (Zhexin) Qiu, Esther (Cuiping) Fang, Purva Kamat

Abstract

SQLite is one of the most popular database engines that have been installed on billions of devices[1]. Although SQLite is lightweight & consists of a rich set of features for portability and performance, it does not come with replication support out of the box. The motivation of this project is to build a reliable, distributed, relational datastore that leverages the Raft algorithm[2] to realize replication, consistency, and fault tolerance. In addition to the distributed constructs like Leader Election, Log replication we provide support for : Dynamic cluster membership change, Snapshot & Log Compaction, partial support for Non-Deterministic functions, multi raft group transaction support using 2 Phase Commit[4]. The final deliverable includes a distributed Raft server implementation based on open source project sofajraft[3], a REPL client for CLI interactions, a visualization layer implemented in Angular.js, a Spring Boot controller service to orchestrate client-server requests and a coordinator for 2 Phase commit orchestration. The components are deployed through Docker containers.

1. Introduction

SQLite is a popular choice for a relational database engine that is widely used by several widespread browsers, operating systems, and embedded systems. Unlike most other SQL databases, SQLite does not have a separate server process. It is lightweight and contained in a single disk file. However, it does not support replication and thus does not provide fault tolerance for the data nodes. We use the idea of leveraging Raft algorithm to build a distributed SQL database. In this paper, we would first describe why we chose Raft in section 2. In section 3, we discuss DSQLite in detail. Furthermore, in section 4 we describe comparison of this service with a similar project - RQLITE[6] and the original SQLite. Finally, we conclude with the discussion on limitations, future work and our learnings from this project.

2. Raft

In this section, we briefly describe the Raft consensus protocol and the SOFAJRaft library we leveraged to build the feature-rich distributed SQLite service.

2.1 Raft consensus protocol

The Raft consensus protocol is equivalent to multi-Paxos but with understandability as its prime goal in design. As a result, this is an ideal candidate to serve as the foundational algorithm to enhance the fault tolerance and scalability of the original SQLite implementation given the project time constraints. The central properties of the Raft consensus protocol is a strong leader and leader election[2]. Our service strictly follows this design to realize the serializability and consistency of the database read/write operations.

2.2 SOFAJRaft

The open source project SOFAJRaft is developed by Ant Financial Services Group as a library to enable building distributed applications on top of Raft. We leverage the stability and abstraction provided by the library to focus on the realization of a distributed version of SQLite with a rich set of features.

3. Distributed SQlite Service

In this section, we discuss the architecture & implementation details of our distributed SQLite service and various supported features.

3.1 Architecture

The DSQLite service supports horizontal scale out architecture. A user can add/remove nodes - where each node means a Raft server handling its own state machine - a SQLite database. To provide ease of use, we have provided a GUI. The GUI client talks to the DSQLite cluster through a middleware - a Rest Controller. This controller handles REST requests and converts them into RPC requests for the DSQLite Leader. From a developer friendly perspective, we have also provided a CLI tool that directly connects with the DSQLite leader using RPCs. Please see Figure 1 for more information.

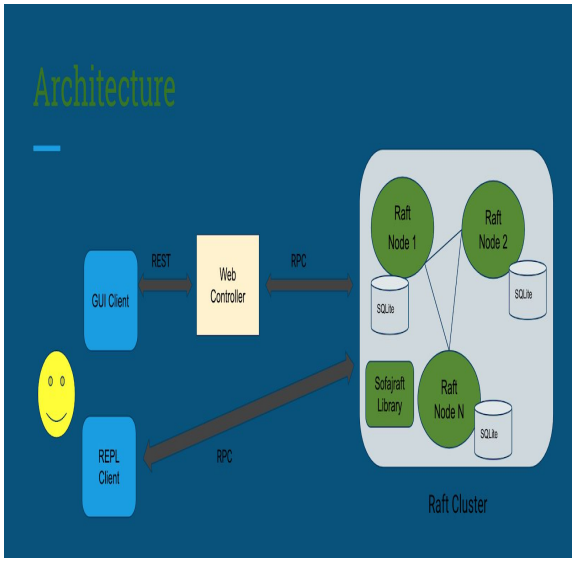


Figure 1 - Architecture of DSQlite service

3.2 Read Requests

Although each node can service a read request, our implementation sends all the requests to the leader. Even at the leader, if we execute read at its own state machine, we run the risk of returning stale data as there could be a new leader due to network partition. To avoid this problem, one way is to treat each read request as a write request. However, this introduces a new problem - reads become expensive. To overcome this problem, one optimization that SOFAJRaft library provides is as follows[3]:

- The leader records the commitIndex of its current log into a local variable readIndex.
- It sends heartbeat messages to the followers.
- It waits for its state machine to execute so that applyIndex exceeds readIndex.
- It executes the read request and returns results to the client.

We, however, provide support for multiple read modes, 'Local' or 'Strong'. If a client chooses to accept potential stale entries, it can provide 'Local' as read consistency level in the request. By default, we use 'Strong'.

3.3. Update Requests

All SQL requests that can potentially change the state of the SQLite database are considered update requests. Such update requests follow the path of usual Raft log replication. The leader receives the request and propagates it to all the followers and every node applies it to its own state machine - SQLite database.

3.4 Nondeterministic Function Handling

An SQL statement is considered to be nondeterministic if it returns different results even with the same input values. We partially support some of the nondeterministic functions like date(), datetime(), time(), Julianday(), random(), order by random(). We evaluate non deterministic functions locally at the leader and replace the nondeterministic functions with evaluated values and broadcast the processed query from the leader. We have special handling for order by random. We find the table that we need to sort by regular expression and find the total number of rows in that table. We assign a random number (need to be smaller than the total number of the rows in that table) to each of the rows and order each row using the random value.

3.5 Snapshot and Log Compaction

Realistic systems are always resource bound in terms of storage. For a log replication system implemented in Raft, there is a physical limit for how many logs a single server can persist. Even though storage is becoming progressively cheaper in the modern era, servers in a replicated state machine system still need to consume excessive computing power in order to process complex queries during log replaying. Snapshotting is a simple approach suggested by the Raft paper to allow log compaction and reduce the need for excessive log replays[2]. This will reduce the time for initializing rebooted nodes or configuring new nodes. The general idea is to take a snapshot of the current system state and write into stable storage such that the entire log up to that point can be discarded from the system.

In our distributed SQLite service implementation, each individual SQLite Raft node has a periodically running snapshot executor service to manage the snapshots stored on the system. A separate log manager process will monitor the snapshot progress in the system and clean up logs that are no longer needed. The doSnapshot(...) method in SnapshotExecutor service will decide whether to take a snapshot based a set of preconditions (e.g. if the server stopped, is the server loading another snapshot file, is the current state machine already synched with the last applied log index). When a new DSQLite node is added to the Raft cluster, the leader will send the latest snapshot to the new follower and a InstallSnapshot event will be triggered. The new node will initialize its state machine and the leader node will keep sending AppendEntries requests that contain logs after the log index that is captured in the snapshot.

One improvement we did is to leverage the natural property of the SQLite (or any database management engine) which already stores the state machine state in stable storage. Instead of creating actual snapshots, we just created a soft link to the already existing SQLite database files. This avoids the need for duplication and significantly improves the time for snapshot creation and deletion. Refer to Figure 2 for more information.

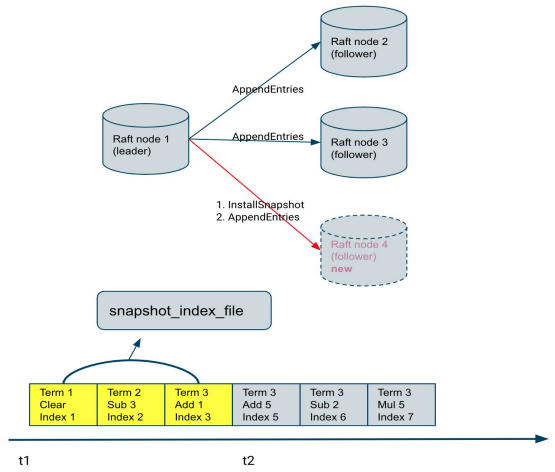


Figure 2 - Snapshot and log compaction

3.6 Distributed transaction between raft clusters

Distributed SQLite also supports distributed transactions between databases in different raft groups. The distributed transaction is performed using Two Phase commit (2PC) protocol[5]. ACID property is guaranteed during the distributed transaction.

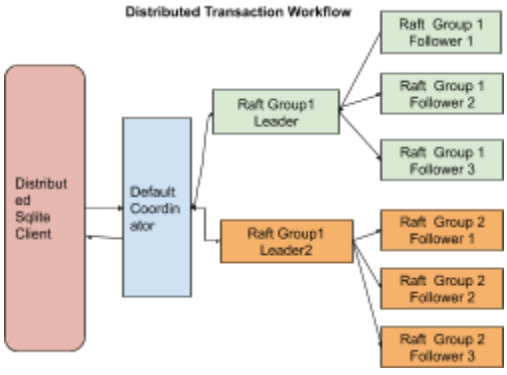
3.6.1 Implementation of Two phase commit in Distributed SQLite

It follows the basic algorithm of two phase commit protocol which has two phases. We have implemented a *Coordinator* to receive distributed transaction requests from clients and coordinate transactions between different distributed sqlite databases managed by different raft groups.

Below we describe a typical workflow of a distributed transaction initiated by a client.

1. Distributed sqlite clients enter the Raft group id involved in the distributed transaction and the corresponding sql commands for each Raft group. It will create a Distributed Transaction request and the request will be sent to the coordinator.

2. The coordinator will send a PREPARE message to the leaders of Raft groups involved in the transaction. By using Raft, the sql commands received by the leader will be replicated by all followers in the group.
3. After sending VOTE requests to participants.
    a. The coordinator records this transaction into the local log.
    b. Upon receiving a PREPARE request from coordinator, all participants will record it and execute the SQL queries in the state machine without committing in the database. If it fails, the participant will respond No to the coordinator, otherwise it will return Yes. And participants will record the vote result in the local log.
4. After receiving votes from participants, The coordinator will decide the outcome and send it to participants and record this in the local *log*.
5. Upon receiving a Decision, participants will commit or abort the transaction accordingly and record the outcome in the log.
6. Once the coordinator reaches a decision, it will reply to the client with the outcome of distributed transaction (COMMIT or ABORT)



. Figure 2 - Architecture of multi raft txn support

3.6.2 Fault tolerance

In case of machine failure, the coordinator and participants use local log (*LogRecordRepository*) to recover all the unfinished transactions. *LogRecordRepository* stores all the transaction information.

If a participant votes 'Yes' and crashes after sending 'Yes' to the coordinator, it is highly likely that a leader election would have happened before the server recovers from the crash. Because of the consistency guarantee of the *Raft* consensus, the next leader is guaranteed to know PREPARE operation. The next leader has to check the local *LogRecordRepository* for any transaction not in final state (not in Abort or Commit). If the participant votes 'No' for the transaction then it can safely record the decision as Abort, otherwise it needs to contact the coordinator to get the final decision of the transaction and apply it on its *state machine.* If the coordinator fails, participants will abort any transactions that are not in final state.

4. Benchmark and Performance

We performed benchmarking against Rqlite and SQLite. Rqlite is also a replicated sqlite database[6] built using Raft.

We have included our benchmark datasets and commands to perform testing in the git repository. The tests that compare with Rqlite and perform distributed transactions were done locally on a personal machine with 16 GB memory & 3.1 GHz Intel Core i7 processor. The comparison with SQLite was performed on a personal machine with 8 GB memory & 1.6 GHz processor.

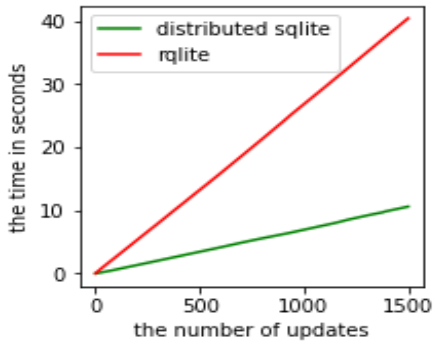
4.1 Comparing DSQLite with Rqlite

The cluster size for rqlite and distributed sqlite (DSQlite) is 3 and all the nodes run on the single host.

4.1.1 Write Test

CREATE TABLE bar (id INTEGER NOT NULL PRIMARY KEY,name TEXT);
For i in range (0, 1500)

Insert Into bar(name) VALUES(“Fiona" + str(i) + ");

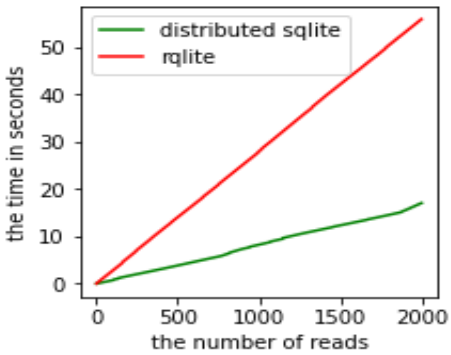


We observe that Distributed SQLite outperforms Rqlite in terms of speed of handling write requests. We believe the difference in time is mainly caused by optimization done in the underlying raft library that we used[2].

4.1.2 Strong Read Test

For i in range(0, 2000):

select * from foo where id like \"fiona" + str(i) + "\"



From the graph, we observe Distributed SQLite is twice faster than Rqlite. The reason is that we used read index optimization which is offered by SofaJraft.

4.2 Comparing DSQLite with SQLite

A single instance of SQLite DB was compared against a 3 node distributed sqlite (DSQlite).

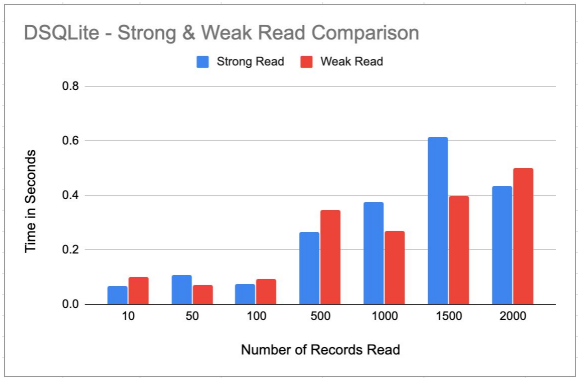
4.2.1 Write Test

CREATE TABLE Employee (ID numeric, FIRST_NAME text, LAST_NAME text, STATE text, DATE_OF_BIRTH text)

INSERT INTO Employee (ID, FIRST_NAME, LAST_NAME, STATE, DATE_OF_BIRTH) VALUES (87098,'Valrie','Gormley','CO','02/28/2005')

Number of Records	10	50	100	500	1000	1500
SQLite	0.0006	0.0019	0.0029	0.0186	0.0464	0.0812
DSQ Lite	0.7724	4.0046	7.758	33.7103	64.9713	95.1153

4.3 Comparing Strong Read and Weak Read in Distributed SQLite



From the graph we observed that Weak Read is relatively faster than Strong Read as it does not need to go through the regular Raft log replication process.

4.4 Distributed Transaction Test

number of nodes	1	2	3
Time (in seconds)	0.059547699	0.405190655	0.458951374

We can tell from the graph that as the number of the nodes involved in the transaction increases , the runtime of transactions would increase as well. However, it is not a linear relationship, because all the rpc calls from coordinator to participants are sent in parallel. The runtime would also increase if the queries that involve the transactions are complicated and take more

time to prepare or the geo distance from the coordinator to participants increases.

5. Future work

Although Distributed SQLite was built with fault tolerance and consistency in mind, the performance is not the primary goal for the project. In the future, we would like to adopt some optimization techniques to improve the DSQLite service's overall latency and throughput for handling database operation requests.

One common technique is to apply sharding in the system. There are various ways to manage shards in a distributed system: 1. Hash-based sharding where a consistent hashing function is used to designate a hash key (which corresponds to a certain node) for each shard. 2. Range-based sharding divides data based on ranges of keyspace. 3. Geo-based sharding partitions data according to a user specified column that maps range of shrads to specific regions[7]. In the context of our DSQLite service, we could provide an optional sharding parameter that can be specified by the client to shard based on a certain column. Different shards will be assigned to different Raft server groups so we could improve performance by spread loads across the groups and still tolerate failures.

Another optimization is to introduce a caching layer or fine tune the SQLite cache configurations in order to improve the system's performance. For example, if a client issues a request with some ID and reissue the same request before the first request finishes, we start a new request. If we cache the incoming request ID, this problem can be avoided. From the GUI client, we currently support one SQL query at a time, this could be improved as well.

6. Conclusion

Distributed SQLite service is a fault-tolerant, consistent, replicated database service built on top of Raft and SQLite to support common SQL query operations and partially support nondeterministic functions. The service can be easily deployed in a containerized environment by a python script which later can be generalized to be deployed on network configured on-prem or cloud infrastructures. [8]

7. Acknowledgement

We would like to express our gratitude to the cs244b course staff David and Jim for providing constructive suggestions and feedback for our project and hosting inspiring talks throughout the quarter. It's a fruitful journey and we learned a lot about distributed systems.

REFERENCE

[1] Most Widely Deployed SQL Database Engine. Accessed April 25, 2020. https://www.sqlite.org/mostdeployed.html.

[2] Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pp. 305-319. 2014.

[3] "Distributed Consensus - Raft and JRaft." Distributed consensus - Raft and JRaft · SOFAStack,n.d. https://www.sofastack.tech/en/projects/sofa-jraft/consistency-raft-jraft/.

[4] Lampson, Butler, and David B. Lomet. "A new presumed commit optimization for two phase commit." In VLDB, vol. 93, pp. 630-640. 1993.

[5] O'Toole , Philip. "Rqlite/Rqlite." GitHub, February 27, 2020. https://github.com/rqlite/rqlite/blob/master/DOC/CONSISTENCY.md.

[6] O'Toole, Philip. "Philip O'Toole." Vallified, May 13, 2020. http://rqlite.com/.

[7] Choudhury, Sid. "How Data Sharding Works in a Distributed SQL Database." The Distributed SQL Blog, June 6, 2019. https://blog.yugabyte.com/how-data-sharding-works-in-a-distributed-sql-database/.

[8] Rajan, Arun, Esther(CUIPING) FANG, Ethan(Zhexin) QIU, and Purva Kamat. "cs_244b_2020_team7 / Distributed_Sqlite." GitLab. Accessed April 25, 2020. https://gitlab.com/cs_244b_2020_team7/distributed_sqlite.