ReSQLite: Replicating SQLite using Raft

Henri Stern, Jervis Muindi
Stanford University

ABSTRACT

We present a simple implementation of a replicated SQLite database system. While SQLite is not optimized for high-concurrency workloads or large data needs, its simplicity still makes it an attractive choice for simple applications and we envision certain use-cases are out-of-scope for SQLite today given its lack of support for replication out of the box. ReSQLite is a simple implementation of a replicated SQLite system built using the Raft consensus algorithm. It is built using a new Raft implementation built from scratch and existing SQLite3 drivers for Golang. Whilst many features remain to be added to ReSQLite, enabling dynamic cluster size changes for instance, it currently supports leader election and log replication thus making it easy to run a replicated SQLite database across multiple machines, enabling system logs or some cached information to survie machine failures. We also present a simple ReSQLite client to interface with the system and finally discuss the limitations of the system.

1 INTRODUCTION

SQLite is a popular relational database management system used in many applications. It comes pre-packaged with many operating systems like OpenBSD, Windows 10, iOS and Android, and web frameworks like Rails and Django. SQLite is easy to use and requires little configuration, and as such it is often used for development work, or to fulfill simple database needs for applications. SQLite databases live on a single file and the entire system is accessible through a simple C/C++ interface making it very portable. Not surprisingly, it is also used by modern browsers like Google Chrome, Opera, Safari, and parts of Firefox (for managing bookmarks, cookies, etc.).

This simplicity however means that SQLite is ill-suited for high-concurrency workloads, distribution or replication. The maintainers of SQLite themselves argue against using SQLite for when the application is separated from the data store (as it is in many client/server applications [10]). There exist widely-used RDBMSes (Relational Database Management Systems) that enable replication or data sharding out of the box, such as MySQL or PostgreSQL. However, without envisioning high-concurrency workloads, there are still many cases when a user would want a lightweight, fault-tolerant database system, keeping application logs on multiple machines for instance in case any single machine fails, or replicating data caches across servers. For that reason, we attempted to build an easy-to-use replicated SQL database system: ReSQLite [6]. We decided to use the Raft consensus algorithm [5] to simply replicate our data across different SQLite instances and build a simple client interface as a proof-of-concept to demonstrate the simplicity of the resulting system. Given Raft's dependence on write logs to operate, it is a natural choice as a consensus algorithm with which to build a replicated SQL service.

In our implementation, we chose to focus on the aspects of replication most critical to our envisioned use cases, namely leader election and log replication and for the purposes of CS 244b decide to implement Raft from scratch, whilst integrating with an existing SQLite library for Golang [11]. In section 2 we will provide an overview of our raft implementation and SQLite integration before discussing some implementation details. We will then discuss some properties of our Replicated SQL service in section 3 before discussing some performance considerations in section 4. In section 5, we will talk about some limitations of our implementation and future work we would like to carry out before concluding. To keep this writeup brief, we choose not to discuss related works, but instead point our readers to the Raft paper [12], SQLite documentation [9], and an already widely-used implementation of replicated SQLite [7].

2 RAFT

In this section, we describe our adaptation of Raft—built from scratch—to ReSQLite.

2.1 Overview

Raft is a consensus algorithm used to manage replicated logs across multiple machines. As such it is a natural choice for implementing a replicated SQL service at the statement level. Central to Raft's consensus algorithm is the notion of a strong leader. For the purposes of this project we chose to focus on the main features of Raft—its leader election and log replication protocols—and forgo its support for cluster membership changes and client and log compaction given the project time constraints as we felt those were less central to supporting our replicated SQLite implementation.

2.2 Implementation

We implement the *AppendEntries* RPC (which enables log replication) and *RequestVote* RPC (which enables leader election) in about 1,500+ lines of Go code. We make use of Google's *grpc-go* library [2] to enable RPCs between our machines, and use protocol buffers [4] for our messages.

Our Raft logs are kept on disk in SQLite databases. This greatly simplifies our implementation and enables us to guarantee fault tolerance in case of machine failure. A machine will reboot and can rebuild its state from the log, interacting with other machines in the cluster to ensure its state is up to date.

3 REPLICATED SQL SERVICE

In this section, we describe the SQL service used in creating ReSQLite.

3.1 Integration with Raft

Rather than build a standalone Raft implementation and separate ReSQLite server to interface with it, we decided to integrate our SQL service directly into Raft given the simplicity of doing so in reusing some of our existing Raft code to make this integration. It added another 500 lines of code or so. Given more time, we would opt to split our Raft implementation from our SQL service, to separate concerns and have a cleaner Raft interface, but this method enabled us to get a working implementation faster.

The service uses SQL statement-based replication by implementing a new RPC of our own making: *ClientCommand*, described in Figure 1. Simply said, each replica is an independently running SQLite database. *ClientCommand* expects a request containing the SQL query sent by the client. The query will be in one of two fields: *command* which denotes a query that mutates the underlying SQLite instance or *query* which simply fetches some data. If the *ClientCommandRequest* is a command, the Raft leader will issue an *AppendEntry* RPC call to append the command to all follower logs. Upon receiving a confirmation from a majority of followers, the leader will execute the command locally (updating its own state machine, in this case its SQLite instance) and return a confirmation to the client. If the *ClientCommandRequest* is a query, the leader will simply run the query to its local state machine and return the result to the client.

Mutating commands are only applied if a majority of the cluster has received them. We also optimize for quick replies in the case of non-mutating queries where the leader will simply run the query in a non-replicated fashion (since it knows the query and should return the same response on all machines). If a client sends a command to a non-leader, its command is rejected and the client is notified of who the current raft cluster leader is.

In order to ensure safety, we must disable all non-deterministic SQL functions, namely *RANDOM()* and *NOW*. We also disable support for SQL transactions as that would require maintaining extra state at each machine in order to track transaction state (committed, rolled back, etc) and ensure machine states are not modified prior to transaction commitment on Raft replicas. Supporting transactions would require modifying the way Raft log replication works to support this special case, or adding an abstraction layer between our ReSql service and Raft. Whilst we prefer the latter method, we decided to disable this use case in ReSqlite for now. Right now, all of this work is done as part of ReSQL client. This stems from our decision not to separate our Raft implementation from our SQL service. In the future, we envision moving these checks to the standalone ReSQLite server to make lightweight ReSQL

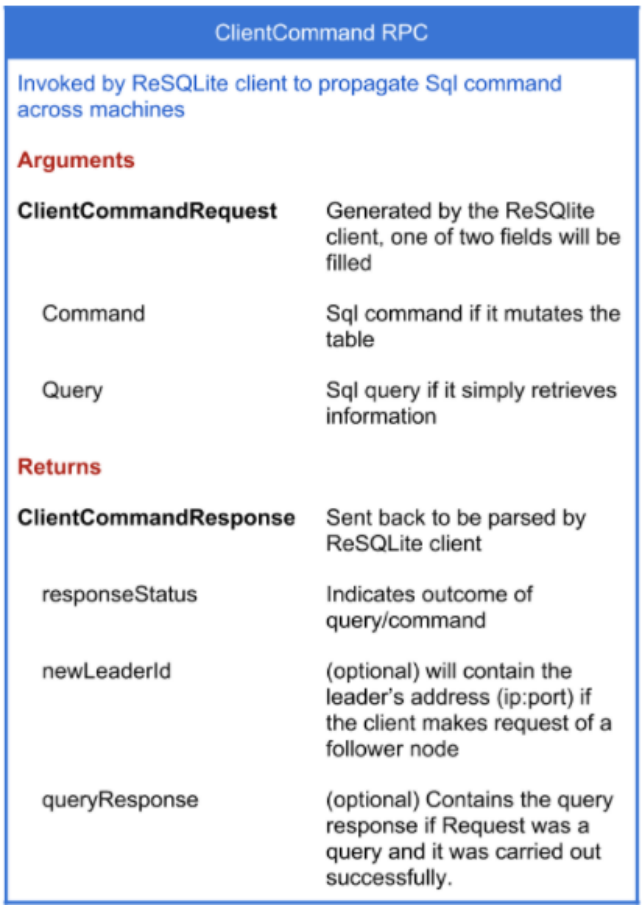


Figure 1: description of the *ClientCommand* RPC.

clients easier to implement.

ReSQLite does not currently do any special handling of non-deterministic user-defined functions, future work should disable support for user-defined functions or prevent non-deterministic ones from being created/used.

3.2 ReSQL Client

Our ReSQL Client is implemented in about 300+ lines of Go code. It is modeled after the sqlite3 command line shell and accepts much of the same syntax, save for the use of its *dot-commands* [8]. We chose to disable dot-commands given that they are not supported by the sqlite3 Golang library we chose to use but envision adding support for them in the future, if only by translating dot-commands to regular SQLite commands (there are relatively few of them).

The ReSQLite client provides a REPL environment to a user so the user can issue SQL commands to ReSQLite. It starts up by connecting to one of the Raft servers. Whenever a command is issued, the client does the following:

(1) It checks whether the command is valid, rejecting transactions or other non-deterministic commands as mentioned above.
(2) It checks whether the command starts by *SELECT* or whether it mutates the database and creates a *ClientCommandRequest* accordingly.
(3) It issues the *ClientCommandRequest* and displays the result or error appropriately.

Any commands issued to follower servers by the ReSQLite Client will be rejected by the Raft servers, with a reply containing the leader address. A user can manually specify which server the client should connect with using the *–server* flag. Otherwise, the ReSQLite Client should automatically attempt to reconnect with the leader in that case and reissue the command, transparently.

If multiple commands are issued to the client at once (as can be done with sqlite3's CLI by running multiple concurrent commands before ending a line with ;), ReSQLite will run them one at a time, halting if any fails and returning the error to the user.

Finally, the ReSQLite client has a *batch* flag that takes a file location and will silently run the SQL commands from the file. Batch mode can be run with an interactive flag to launch the REPL after the file's SQL commands have been run. We used this extensively to test our implementation and do benchmarking (see section below).

4 PERFORMANCE

In this section, we run ReSQLite on sample datasets and compare its performance to sqlite3.

4.1 Running ReSQLite

The ReSQLite source code can be found on Github, along with instructions to install it [6]. Once installed, run ReSQLite by starting up a cluster and launching the ReSQLite Client as described in figure 2. The github repository contains a helper script to start a local raft cluster for testing.

Figure 2: sample commands to run ReSQLite from the source root.

4.2 Benchmarks and Performance

We've included our benchmark datasets in the data subdirectory to make our tests easy to replicate. For the purposes of this paper, we chose to run four benchmarks: read-only, write-only, read-heavy and write-heavy as described in figure 3. The results on a one-node cluster, three-node cluster and with the sqlite3 library are also shown in figure 3.

The tests were all done locally on a Macbook Pro 15 with 16GB RAM and a 2.8 Ghz Intel Core i7 CPU running macOs Sierra 10.12.6. The version of sqlite3 used for testing was SQLite version 3.16.0 2016-11-04 19:09:39.

Looking at the results we see that ReSQLite is slowest in the write-only test. This can be explained because we need to make synchronous disk writes in order to safely replicate the data. The measured overhead for a single-node cluster is about 2.8x whereas the overhead for a three-node cluster is 5.6x. We remark that these higher values could be improved with some performance optimizations in our implementation. For example, one optimization likely to have a noticeable impact would be to batch entries for the *AppendEntries* RPCs. This would enable the client to send fewer commands to the raft cluster.

The results for read-only test case are a bit different. In this instance, we see that regular sqlite3 took 41.4 seconds to complete the test, whereas the single-node raft cluster and three-node raft cluster took 14.1 and 28.5 seconds, respectively. Taken at face value, this looks like an improvement of at least 31% in read performance. We interpret this to mean that for read-only queries, the performance of ReSQLite is comparable to that of regular SQLite. However, we do not believe these results reflect system performance but rather a quirk in our testing methodology. The read-only test for the SQLite wrote results to standard output whereas the ReSQLite client only checked the RPC response code and did not write out responses. Thus, the poorer SQLite performance likely results from *stdout* I/O costs.

For the write-heavy workload test, which includes a mix of read and writes commands, we find that the measured overhead for ReSQLite in a single-node raft cluster configuration to be about 1.8x and for a three-node raft cluster configuration to be about 3.7x. Just as before, the majority of the slower performance for ReSQLite comes from lack of any optimizations for batch SQL insertions to import data. The write-heavy workload includes some read queries which more closely matches real-world usage patterns. It is worth noting that in such a setting the overhead to running our replicated SQLite system is about 3.7x, without any optimizations. This is high, but provides a pathway to optimizing ReSQLite to acceptable performance.

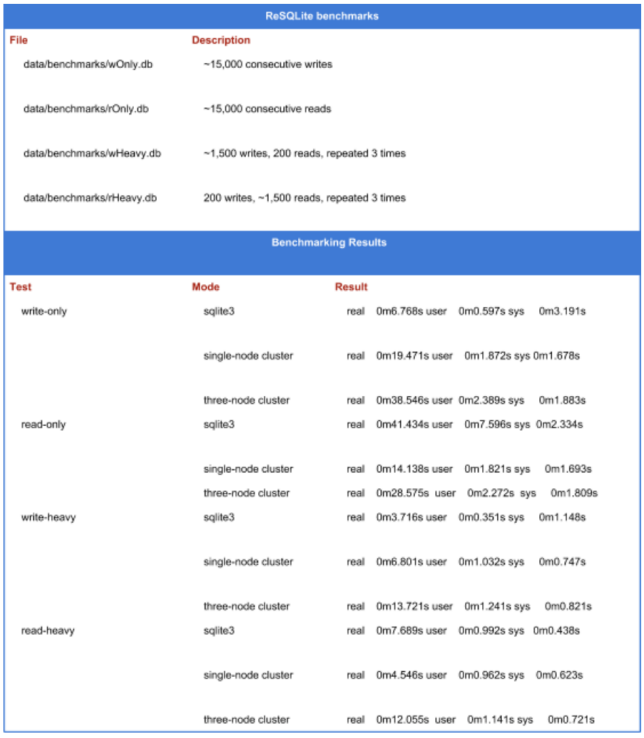
Finally, the last test we conducted was doing a read-heavy workload benchmark test. This test includes a mix of read and write commands (with 7-8 read queries for every write query). Compared to the standard sqlite implementation, we find that a single node ReSQLite cluster to be about 40% faster. This is likely, once again, due to testing methodology and the fact that sqlite3 writes responses to *stdout*. The result for the three-node cluster is about 1.5x slower than regular sqlite3. This is to be expected because ReSQLite has to replicate data synchronously for safety.

In summary, we believe ReSQLite may be an appropriate solution for read-heavy workloads. Read queries are performed directly by the raft leader and as such perform similarly to unreplicated SQLite. Replicating data at multiple nodes necessarily incurs some overhead. As mentioned, we believe the measured penalty—currently 5.6x for three-node cluster—can be drastically improved with a few write optimizations.

5 FUTURE WORK

While basic functionality is here for ReSQLite, there remains work to be done to turn it into a production-ready system.

Architecture. As mentioned earlier, we would like to split our SQL service away from our Raft implementation. Doing so would make the ReSQLite library more modular and facilitate future development of its pieces (Raft, ReSQL server, ReSQL client) independently. This



ReSQLite benchmarks

File	Description
data/benchmarks/wOnly.db	~15,000 consecutive writes
data/benchmarks/rOnly.db	~15,000 consecutive reads
data/benchmarks/wHeavy.db	~1,500 writes, 200 reads, repeated 3 times
data/benchmarks/rHeavy.db	200 writes, ~1,500 reads, repeated 3 times

Benchmarking Results

Test	Mode	Result			
write-only	sqlite3	real	0m6.768s user	0m0.597s sys	0m3.191s
	single-node cluster	real	0m19.471s user	0m1.872s sys	0m1.678s
	three-node cluster	real	0m38.546s user	0m2.389s sys	0m1.883s
read-only	sqlite3	real	0m41.434s user	0m7.596s sys	0m2.334s
	single-node cluster	real	0m14.138s user	0m1.821s sys	0m1.693s
	three-node cluster	real	0m28.575s user	0m2.272s sys	0m1.809s
write-heavy	sqlite3	real	0m3.716s user	0m0.351s sys	0m1.148s
	single-node cluster	real	0m6.801s user	0m1.032s sys	0m0.747s
	three-node cluster	real	0m13.721s user	0m1.241s sys	0m0.821s
read-heavy	sqlite3	real	0m7.689s user	0m0.992s sys	0m0.438s
	single-node cluster	real	0m4.546s user	0m0.962s sys	0m0.623s
	three-node cluster	real	0m12.055s user	0m1.141s sys	0m0.721s

Figure 3: ReSQLite benchmarks and results.

change would involve moving all non-deterministic checks, as well as the determination of whether a given user SQL command is a command or a query from the ReSQLite client to the ReSQLite server.

Safety. Future work is required to safeguard ReSQLite from user-defined non-deterministic functions. Likewise, we would like to add transaction support through the ReSQLite server (as mentioned above in the architecture changes). Further we would like to write more integration tests for ReSQLite to ensure it's functioning correctly under various edge cases and other workloads that are tricky to test. We envision leveraging some of the tooling in Jepsen [3] made available by Aphyr [1] to validate the safety characteristics of our distributed database system.

Performance. Support for transactions would go a long way toward improving ReSQLite performance. Likewise, we believe we could do further progress by using an in-memory SQLite database by default, rather than the file based one currently used, thereby removing many disk I/O operations. Moving ReSQLite's database to in-memory is a relatively straight forward change. It would also not put data at risk since Raft's logs are already durably stored on disk and represent a source of truth for all SQL transactions. The only thing missing from our implementation in order to make this switch is support for replaying transactions from the log on server restart (after a machine failure). As mentioned in the *Benchmarking* section, batching writes would also likely help improve performance.

Additional Features. We would like to implement further features from the Raft paper like support for cluster changes and log compression through snapshotting.

6 CONCLUSION

ReSQLite is a replicated implementation of the SQLite database built using the Raft consensus algorithm. While it lacks certain features, it uses the main features of Raft—leader election and log replication—to build a simple and efficient replicated SQLite system. We believe it constitutes a good starting point for this system that combines the simplicity and ease-of-use of SQLite with the fault-tolerance that Raft replication guarantees and could be used for replicated application databases and easily extended to allow SQLite data sharding. The replication functionality might prove useful in IoT (Internet of Things) device clusters to ensure safety of local data (e.g. the data of temperature sensors in a home network).

7 ACKNOWLEDGEMENTS

We would like to express our thanks to Raft paper author Diego Ongaro for talking with us and giving us early feedback/guidance on our project proposal. He proved a major inspiration in having us write our own Raft implementation in order to better grasp the inner-workings of the Raft protocol. We would also like to thank David Mazières, Hiroshi Mendoza and Seo Jin Park for an amazing quarter. We thoroughly enjoyed taking CS244B.

REFERENCES

[1] Aphyr: About us. https://aphyr.com/about.

[2] grpc-go. https://github.com/grpc/grpc-go.

[3] Jepsen: A framework for distributed systems verification with fault injection. https://github.com/jepsen-io/jepsen.

[4] Protocol buffers Go library. https://github.com/golang/protobuf.

[5] Raft: Website. https://raft.github.io/.

[6] ReSQLite source code. https://github.com/jervisfm/resqlite.

[7] RQlite: The lightweight, relational database. https://github.com/rqlite/rqlite.

[8] SQLite CLI documentation. https://sqlite.org/cli.html.

[9] SQLite documentation. https://sqlite.org/cli.html.

[10] SQLite: When to use. https://sqlite.org/whentouse.html.

[11] SQLite3 Go library. https://github.com/mattn/go-sqlite3.

[12] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.