Hybrid-Consistency Replicated Database

Vincent Boersch-Supan & Bright Liu

A final project for CS 2620: Distributed Systems.

Harvard College, May 2025.

Our code is available at github.com/vboesu/cs262/.

1 Introduction

Modern distributed applications often have both critical and non-critical data in a replicated setup. Critical data (e.g., user credentials, billing records) require strong consistency to ensure correctness and prevent anomalies such as duplicate usernames or incorrect account balances. Non-critical data (e.g., social media posts, comments, likes) is read and written at much higher rates, but can tolerate eventual consistency without severely impacting the user experience.

Strong consistency typically requires routing every modifying request through a single leader node, serializing transactions and synchronizing replicas before acknowledging the client. While this model guarantees correctness, it introduces substantial latency and a coordination bottleneck under high concurrency. In contrast, eventual consistency allows replicas to process writes independently and reconcile state asynchronously, maximizing availability and throughput but risking temporary divergences.

Our motivation is to balance these two models by allowing developers to mark specific tables or columns as strongly consistent, while treating the rest of the data as eventually consistent. This hybrid approach delivers fast, highly available operations for the bulk of application traffic, without sacrificing correctness for data that truly matters.

The primary goal of our project is to build a prototype of a hybrid-consistency replicated database overlay on an existing SQL engine. The setup we are envisioning is one where many clients interact with many replicated API servers, which in turn interact with the replicated database server, e.g. through a load balancer. To keep our discussion more general, we will refer to the entities interacting with the database as "clients", even if they are potentially servers themselves. Thus, the clients must be unaware, unaffected, and uninvolved in the replication mechanism of the database.

2 Challenges & Design Choices

2.1 Wire Protocol

The first question we asked ourselves was "How will the clients interact with the database?" In an ideal scenario, we build replicated proxy servers which sit on top of existing databases (e.g. PostgreSQL or MySQL) and handle all incoming communication with the client as well as communication between the replicas. This would integrate seamlessly with any existing software that works with such database servers. We explored this option with PostgreSQL but found three major challenges. For one, the PostgreSQL wire protocol uses some kind of SSL encryption which our proxy needs to replicate and relay. Secondly, our proxy would need to re-implement the entire PostgreSQL wire protocol. Most critically, though, the PostgreSQL wire protocol uses `RELATION`s which we can think of as shorthands for server-client communication (e.g. by shorthanding a specific table as a number). These references would need to be communicated among the replicas to ensure that any request can be routed to any replica without any weird errors. Moreover, it is unclear to us whether the IDs used by the relations are unique or deterministic across clients and servers, so there is potential for these references to collide, creating undefined behavior.

We therefore decided to design a simple wire protocol from scratch to support simple `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements (though it would be straightforward to extend this to transactions made up of compositions of these). It is implemented as a simple REST-style HTTP API using Flask. The replicas use sockets for communication with a bare-bones wire protocol minimizing overhead. While this approach also involves a fair amount of parsing, it gives us full control over how we use and combine data.

The HTTP API maps HTTP request methods to basic SQL operations:

$$\texttt{GET} \rightsquigarrow \texttt{SELECT},\ \texttt{POST} \rightsquigarrow \texttt{INSERT},\ \texttt{PATCH} \rightsquigarrow \texttt{UPDATE},\ \texttt{DELETE} \rightsquigarrow \texttt{DELETE}.$$

The client then calls the endpoints `/<schema>` or `/<schema>/<id>` for operations on the table (`SELECT`, `INSERT`) or operations on rows (`SELECT`, `UPDATE`, `DELETE`), respectively. Where additional data is required, such as for `INSERT`s or `UPDATE`s, the client sends a JSON body with key-value pairs.

2.2 Query Log

The next big design choice we had to make was how the write queries would be stored in the log of each replica. Since concurrent queries (especially eventually consistent ones) can lead to conflicts, we need a way for the client to verify that its request had the intended effect.

We illustrate this using a simple example. Suppose we store account data including a balance for a particular user in our database, marked as strongly consistent. A client may fetch the user's account balance, check if it is enough for a specified withdrawal, remove the amount from its balance, and send the updated value back to the database for storage.

If the client just sent a SQL request like

```
UPDATE accounts SET balance = 20 WHERE user_id = 123
```

and it was successfully implemented in all replicas, it wouldn't know whether in between its fetching of the user's balance and the update, another client did the same such that we double-spent the user's money. This is very bad!

One potential solution for this which we also discussed in class is *leasing*. A simpler option is simply for the client to tell us which old value it is expecting, and for the database to check if this old value is still the current value. That is, the database server checks if

```
UPDATE accounts SET balance = 20 WHERE user_id = 123 AND balance = 40
```

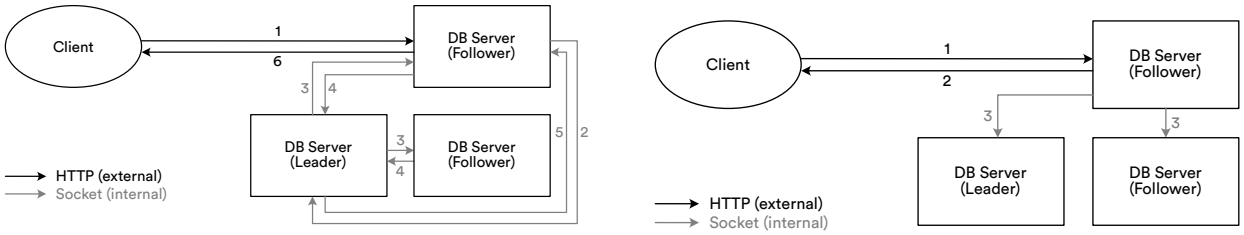
affects exactly one row, in which case we are all good. If, however, the balance has changed, the operation is rejected, and the client has to try again with the new data. This is the approach we settled on, even if it potentially requires more back and forth between the database and the client. To store this in a simple log, we therefore treat a write query as either a single `INSERT` operation, multiple `UPDATE` operations, or a single `DELETE` operation. We store old and new values where relevant, the data to be inserted for the `INSERT` operations, and the unique ID of the object to be modified for `UPDATE`s and `DELETE`s.

For strongly consistent queries, these are stored alongside a unique transaction ID, a physical timestamp and a logical timestamp giving the strong queries a total ordering, while for eventually consistent queries, we only store the transaction ID and the physical timestamp, which can later be used for conflict resolution.[1]

2.3 Consensus

Because strongly consistent queries are easiest to implement and keep track of if they have a total ordering, we decided on a leader-follower model with a single leader responsible for the implementation and propagation of all strongly consistent queries. Any client can talk to any replica, but if it sends a strongly consistent write query to a follower (as determined by the columns affected by the operation), the follower

[1] Here, it is crucial, that the ID and timestamps are only set once by the first replica which implements the query, and then propagated to the other replicas; otherwise there is no ordering possible based on time.



(a) Strongly consistent query

(b) Eventually consistent query

Figure 1: Schematic representation of requests involved in strongly consistent queries (left) and eventually consistent queries (right)

forwards the request to the leader and returns the response from the leader. For an eventually consistent write query, any replica can implement it, return the response to the client, and then broadcast the query to the other replicas. Figure 1 gives a schematic representation of these requests.

Since we decided that each replica would broadcast eventually consistent queries directly to all other replicas, we make all replicas keep a socket open to all other replicas, and send frequent handshakes among each other when no other communication is occurring. When it comes to a leader election, we want to elect the replica with the latest entry in the strong log, and we use a unique integer ID as a tie-breaker in favor of the replica with the higher ID.

Similar to Raft, we use randomized election timeouts to initiate elections. The initiating replica then broadcasts its current logical clock and its ID and receives votes from all of the other replicas, who initiate their own elections if their clock + ID combination is higher. To become leader, a replica needs to receive the votes from all other available replicas. This way, no replica that does not have the highest logical clock can ever become leader.[2]

When the leader receives a strongly consistent query, either directly from a client or forwarded from a replica, it first attempts to implement it itself, and if successful, it distributes the query with its logical and physical timestamp to all available replicas. Because the part of the database which is strongly consistent is a state machine based on the strong log, all other replicas must be able to implement this query. The leader waits for their completion (with some timeout mechanism), updates its logical clock, and returns the result to the client.[3] If it doesn't get a positive response from all followers, it returns a temporary failure to the client, since this usually means that one of the replicas crashed mid-query, and so the client should simply try again. This mechanism ensures that after a successful write, *all* replicas have the same strongly consistent data in their database (not just their log), such that any replica can immediately return reads in

[2]For an even stronger consistency guarantee, we can additionally require that the replica receives the votes from at least a majority of all *possible* replicas, much like Raft. This harms availability (if there are less than half of the possible replicas online) but ensures tolerance to certain kinds of partitions and guarantees consistency.

[3]Here, again, we may require that a majority of replicas be online for consistency.

a consistent fashion.

2.4 Failure

We primarily consider fail-stop node crashes of followers or the leader, which our replication mechanism can handle. The consensus mechanism described above gives $f + 1$ tolerance, while the Raft-like extension requiring a majority of all possible replicas to be online at any given time adds partition tolerance at the expense of $2f + 1$ tolerance.

2.5 Primary Keys

For eventually consistent data, auto-increment integer primary keys are subject to collisions. We were able to think of two possible ways to resolve this. If we know the number of replicas, each replica can simply assign multiples of the number of replicas as primary keys, with a unique offset. Alternatively, we can simply use UUIDs and believe in their (global) uniqueness. We chose to go with the second, because we believe in the existence of GUIDs, and this means that we can scale our system up and down more easily.

2.6 Miscellaneous

- **Adding/restarting replicas.** To allow for the addition of new replicas, the leader transmits its logical clock with each query and each handshake. If it is ahead of the replica's own logical clock, it switches into "learner" mode (which means that for the purposes of consensus problems, it is effectively offline), and continuously asks the leader for the strong queries it is missing until it is caught up.

- **Load balancing and finding.** For demonstration purposes, we use a docker swarm with a round-robin load balancer, and docker-internal DNS to find the other available replicas. Docker's load balancer introduces a single point of failure, such that in a production setting, we would of course use a replicated load balancer instead. As an alternative to DNS, one could also use some combination of configuration files.

- **Race conditions.** Oh my. We want to handle concurrent requests on the external API side but need to bottleneck for strongly consistent queries, wait for elections to finish, etc. We are continuously checking the health of our connections, sending and processing eventually consistent queries, and working on multiple requests simultaneously (whenever possible), so there is a lot of concurrency

that needs to be taken care of. We use relatively coarse-grained locking but switching to finer-grained locking has the potential to massively boost performance (see more below)

3 Results

To test whether our setup meets the goals outlined in the introduction, we create two sample tables, `users` and `posts`. Each `user` object has a unique ID, a unique username and a name. We treat the username as strongly consistent, such that we can easily ensure global uniqueness. Each `post` object has a unique ID, a reference to a `user`'s ID, and some content. Because we treat object IDs as immutable, we will not worry about their consistency. All data about posts is considered eventually consistent.

Now, we run different types of queries and compare their performance across multiple numbers of replicas of our database server. We run each query 1,000 times, first on a single thread, and later spread out over 100 threads, to simulate multiple clients accessing the database cluster at the same time.

After creating some sample data, we run the following queries:

- `get_users`: run a `SELECT` query on the table `users`, returning all rows. This includes strongly consistent data (the usernames) but only as reads, so it can be served by any replica.
- `get_users`: run a `SELECT` query on the table `users`, returning all rows. This does not include any strongly consistent data, but it transmits more data overall.
- `create_user`: run a `INSERT` query on the table `users` with a randomly generated user ID, username, and name. This is a strongly consistent write query.
- `create_post`: run a `INSERT` query on the table `posts` with a randomly generated post ID, one of the previously generated user IDs, and content. This is an eventually consistent write query.
- `create_user_and_posts`: create a user and ten posts by that user. These execute as separate strongly consistent and eventually consistent write queries, challenging the cluster to process both simultaneously.
- `update_user_username`: run a `UPDATE` query on the table `users` for one of the previously generated users with a new, randomly generated username. This is a strongly consistent write query.
- `update_user_name`: run a `UPDATE` query on the table `users` for one of the previously generated users with a new, randomly generated name. This is an eventually consistent write query, since only the `name` column is affected, which is not marked as strongly consistent.

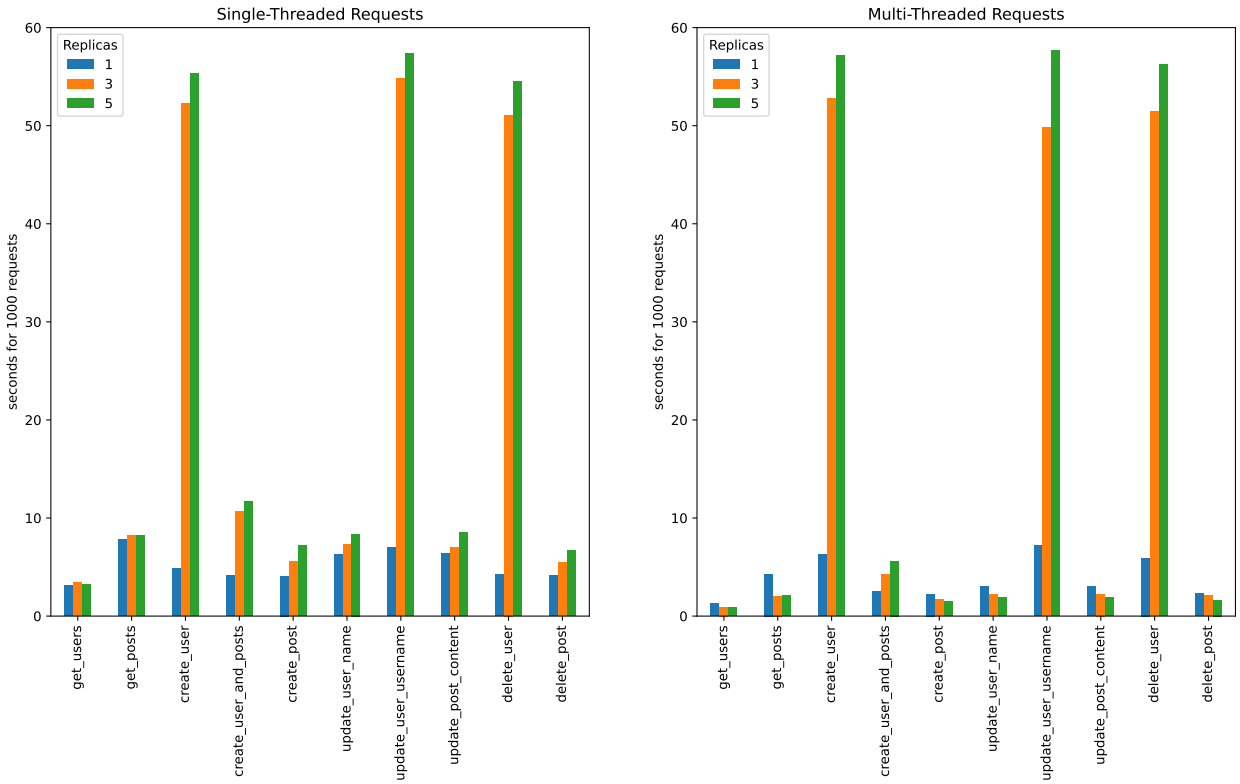


Figure 2: Experimental results of running different types of queries with different numbers of clients and replicas.

- `update_post_content`: run a `UPDATE` query on the table `posts` for one of the previously generated posts with a new, randomly generated content. This is an eventually consistent write query.
- `delete_user`: run a `DELETE` query on the table `users` with one of the previously generated user IDs. This is a strongly consistent write query, since at least one of the columns of the `users` table is strongly consistent.
- `delete_post`: run a `DELETE` query on the table `posts` with one of the previously generated post IDs. This is an eventually consistent write query.

We report the observed request speeds in Figure 2. Since `create_user_and_posts` runs eleven queries per iteration, we provide its average value for a thousand queries. We include a cluster with only one replica to simulate what it would be like to interact with only a single database without the replication overhead, although this of course does not provide any failure tolerance.

Some general trends that we observe is that for single-threaded requests, i.e. a single client sending consecutive requests to the cluster, the response time increases in the number of replicas. This makes sense as there is no gain from the load balancing for a single client, and indeed the communication overhead between the replicas actually slows them down. We also clearly see that strongly consistent write queries

(i.e. `create_user`, `update_user_username` and `delete_user`) and the hybrid-consistency query `create_user_and_posts` are much slower than the other queries when there is more than one replica involved.

It is only for eventually consistent write queries and select queries where we see a clear benefit from increasing the number of replicas when multiple clients send concurrent requests, as the load balancer distributes them evenly across the replicas.

However, the overhead for strongly consistent queries is quite significant: the `create_user` query takes almost ten times as long in a cluster with three replicas as compared to a single database.

4 Extensions

We can think of several possible extensions to our work.

- **Persistence.** The nature of docker containers means that the database does not persist across controlled shutdowns and restarts. However, we can easily add persistent storage to the docker containers by storing the database files on mounted volumes.
- **Strong query performance.** During strongly consistent queries, we currently lock the databases of all replicas entirely to ensure that no queries can interfere with a strongly consistent one. To improve performance, we may instead lock only those parts of the database which are affected by the strongly consistent query. An alternative would be to consider sharding the database and locking individual shards.
- **Eventual consistency with new replicas.** While we can ensure that new replicas can be brought up to date on the strong log, the lack of ordering of eventually consistent queries means that we need to build a separate mechanism to ensure at-least-once delivery of each query (to *actually* achieve consistency *eventually*). One possible way to do this would be to keep track of each received eventually consistent query by its sender, create Merkle trees out of this, and then use a kind of distributed binary tree search to figure out which of the queries a given replica is missing from that particular replica.
- **Support more complex transactions.** This is straightforward for `SELECT` queries (e.g. filters, limits, offsets, ordering, etc.), and multiple `INSERT`, `UPDATE` and `DELETE` operations can also easily be chained together with our current framework. However, joins, subqueries, etc. are a little hard to envision in our current framework. Also, `default` and `onupdate` attributes are a bit of a pain to do right across replicas but can be done.

- **Integrate more directly with a database.** As we mentioned earlier, there are some challenges with directly using an existing database's wire protocol but if they can be resolved, this would make the adoption of this kind of database much more attractive, since developers could just plug it into their existing system.

5 Conclusion

Our final projects demonstrates that it is possible to design replicated databases that support hybrid-consistency data replication for specific tables and even specific columns, ensuring that the database is highly tolerant to failure, and delivering some performance benefits for frequent reads or eventually consistent writes. However, if the overhead from our implementation of strongly consistent data cannot be sufficiently reduced as compared to a system which is optimized for strongly consistent data (and these queries are frequent enough to cause bottlenecks), it might be worth considering storing strongly consistent data elsewhere and joining it onto the other tables where necessary.