ECE1747 Project Report

Distributed Versioning - Scaling Back-End Database through Replication in Asynchronous Rust

Ray Huang (huangr19, 1001781565)

Lichen Liu (liuli15, 1001721498)

Sining Qin (qinsinin, 1001181106)

Abstract

As the number of clients accessing dynamic content web applications increases, handling a massive amount of data requests with a single database instance is incapable of providing satisfactory throughput and latency. Scalable distributed database systems replicating the database back-end is thus required, improving data availability and performance. However, maintaining proper synchronization and consistency across database replications is challenging.

Paper [1] presents the distributed versioning algorithm that achieves conflict-aware query scheduling with explicit versions and reduces the number of lock conflicts, providing both consistency guarantees and scalability. This project implemented a scalable system based on this algorithm. It consists of three main components: a sequencer, a scheduler and multiple database proxies connected to databases.

Using client load generated following an E-commerce benchmark, TCP-W, within reasonable latency, our system demonstrated around 30% improvement in peak throughput at 4-16 database replications, compared with only a single database instance. A higher ratio of Read to Write operations in the client load has shown to have a positive impact on the performance of our system, but two optimizations, supporting transaction-free single-Read and the early version release of database tables, have not.

Throughout this project, we have confirmed that database replications with accesses controlled by distributed versioning can enhance performance. However, the architecture, implementation and benchmarking of such systems are quite complex.

Table of Contents

1. Introduction

Our system is entirely based on the algorithm and designs proposed in the paper: *Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Web Sites* [1].

The idea of distributed versioning comes to attention as a result of this paper in 2003. By definition, the algorithm proposed by the paper achieves one-copy serializability under a transactional database system. A prominent drawback for such a system is the bottleneck on the primary replica that limits the throughput of the entire system. Back then the benchmark of the system was carried out on limited hardware and internet conditions compared to nowadays. It serves as a proof of concept that this algorithm is still relevant in today's network environment to build a scalable database replica system, in the meantime answering the question that if a modern software stack can leverage this design and algorithm to a practical level. Particularly, we decided to use Rust, which is a fairly new system programming language, to implement the entire system. Rust is famous for its unique memory safety feature and performance competence closed to C++. Rust does not have a garbage collector and the memory safety feature prevents issues like double freeing and segmentation fault[2]. This makes parallel and asynchronous programming less error-prone.

Due to the restricted scope of the original paper, most of the implementation details were excluded from the published content. Building an actual implementation allows us to analyze the potential bottleneck of the system and thus gaining insights into the gaps between the theorem and an applicable system. We also hope this prototype can become a stepstone to a more mature database replica system.

The rest of this report is organized as follows. Section 2 describes the distributed versioning algorithm in detail, while Section 3 explains the architecture of our implementation and elaborates on the three main components of the system -- sequencer, scheduler and database proxies. Section 4 elaborates the benchmarking setup and experiments we performed to collect data for performance evaluation, and Section 5 presents the evaluation result and analysis. Lastly, A discussion of the conclusion and future work is provided in Section 6.

2. Algorithm: Distributed Versioning

2.1 Consistency Definition

The system empowered by the distributed versioning algorithm guarantees 1-copy serializability. In other words, all transactions can be seen as executed on a single logical database. This is achieved through a transaction versioning process and regulated Read and Write operations to database replicas.

The system follows a read-one and write-all approach. A Write query will be sent to all replicated and only the first response is returned to the application server. On the other hand, a Read query is only sent to a single replica and the result is immediately returned to the application server.

2.2 Table versioning

The first crucial part of distributed versioning is all tables within the database are tagged with a version number. Each table on every database replica starts its version number at 0 and will be incremented when a transaction accessing the table completes (after a *commit* or *abort*) or if it receives an early release request.

A query that arrives at the replica will also be assigned with a set of version numbers for all tables it accesses. The query will only be executed if it satisfies the version constraint for all the related table versions on the replica. The version constraint is defined differently for different query types, to satisfy the constraint:

- For Read query, all assigned versions are *less* than or *equal* to the table versions on the replica.
- For Write query, all assigned versions are *equal* to the table versions on the replica
- For commit/abort, all assigned versions are *equal* to the table versions on the replica

In the other words, if a query that arrives at the replica holds higher table versions, it is blocked until table versions on the replica get updated due to completion of transactions assigned with lower table versions or early release requests. Therefore the internal transaction locking enforced by the database engine is avoided in most of the cases since table versioning prevents transactions with conflicting table queries to be executed at the same time.

2.3 Transaction Versioning

Transaction is the smallest working unit that performs database queries. In our system, transactions are assigned with a set of table version numbers for the tables it uses, and this set of version numbers defines a global transaction execution order.

Before a transaction arrives at the database, it has to first go through a version assignment process. A global version number is maintained for each table. A transaction has to declare all the tables it uses by its queries, as well as the type of the operation. If a table is both accessed by Read and Write queries in the same transaction, it is deemed to be a Write

operation. For each declared table, depending on the operation types, different assignment methods are used:

- For a Read operation, the last version number assigned to a Write query is used
- For a Write operation, the current global version number is used

After a table version assignment, the global table version is always increased by one. Therefore for a particular table, a Write query will wait on the database replica until all outstanding Read queries finish, while Read queries from multiple transactions can be executed on the same replica concurrently. Figure 1 is an example how a table version is assigned to consecutive transactions and gets incremented.

operation	w	w	r	w	r	r	r	w
version assigned	0	1	2	3	4	4	4	7
version produced	1	2	3	4	5	6	7	8

Figure 1: *An example of consecutive transactions get a version assigned to the same table*

All transactions will be processed sequentially and atomically, specifically a transaction will be assigned for all the tables it uses before the next transaction will be assigned with any table version. If two transactions have a set of conflicting table operations, all the conflicting tables in one transaction are assigned with less or higher version numbers than those in the other transaction.

2.4 Query Versioning

A transaction is made of three phases: Begin, Read/Write, Commit/Abort. During the Begin phase, a transaction will have its version numbers assigned and it will be memorized during the lifetime of this transaction. Followed by the Read/Write phase, it is when database replicas receive Read and Write requests and reply to the results. Read/Write queries use the version numbers acquired in the Begin phase and carry this information to the database replica. Finally, a Commit/Abort query will be executed when all the previous outstanding queries are completed.

A transaction consists of a series of SQL queries, and each query accesses all or a subset of tables declared during the Begin phase. So a Read or Write query can carry all table versions declared in the Begin phase or only a subset of them. The assigned table versions to Read/Write queries will be checked against the table version on replicas to determine if a query can be executed yet.

Commit and abort queries also need to be assigned with table versions. A commit or abort query carries all table versions assigned to the transaction. A commit/abort query marks the completion of a transaction which results in a version release to each used table. Due to that, a commit or abort request is only able to cause version release if it is not blocked by any table.

2.5 Early Release

Table and transaction versioning enables further parallelism between transactions. We adopt a technique named the early release as described by the original paper. In our implementation, a table version can be released (increment) during a Read/Write request, provided that the Read/Write query contains the last access to the table within a transaction. The rationale behind this strategy is intuitive, the next transaction can start reading or writing a conflicting table given that the previous transaction has finished using it.

Early release is only available to Write query. To enable this approach, on a Write request, a list of tables that can be early released by the Write request is included in Write request as an extra field. Upon completion of a Write request carrying an early release request, the replica table version will be released immediately, allowing potentially blocked queries to start executing.

2.6 Single Read

If a transaction only contains a single Read query, it is a special case that no version assignment is needed. Thus no Begin/Commit/Abort query is required for a single Read transaction. A single Read transaction directly bypasses the version assignment process and arrives at the database replica, and will be executed once all outstanding Write queries on the replica are completed.

3. Implementation

3.1 Architecture Overview

As mentioned earlier, the system functions on three modules: Scheduler, Sequencer and Database proxies. The scheduler is the frontend of the system that directly communicates with the application server. There are four types of query requests that an application server can send to the scheduler: Begin, Read, Write, Commit/Abort.

The scheduler forwards a Begin Transaction request to the sequencer for version assignment and directly replies to the application server. There is no Begin Transaction request transmitted to the database proxy. If the scheduler receives a Write query or an End Transaction request, the scheduler forwards the Write query request, along with the assigned version numbers to all database replicas. For a Read query request, it is only forwarded to a single replica. Figure 2 illustrates the data flow of the entire system.

Currently, we only support a system setup with only a single scheduler. The scheduler also expects client input requests to be encoded in a customized JSON-based API that contains the SQL request and annotations. Due to the scope of the project, the system also does not support fault tolerance and error recovery.

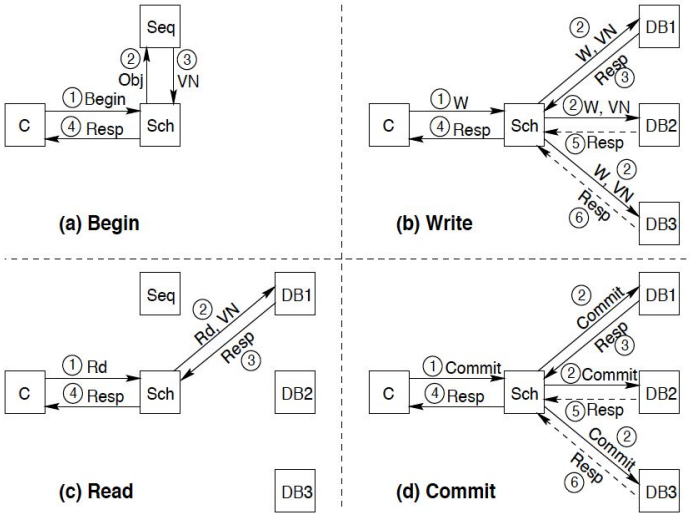

Figure 2: *An overview of the system data flow*

3.2 Asynchronous

The entire system is built using the asynchronous architecture implemented in Rust. The asynchronous architecture is mainly for hiding away all the network latency and allows for maximal concurrency since this entire system deals heavily with the network. To be specific, when a task is blocked by a network operation, it will be suspended to the background from the worker thread executing it, so that the worker thread will not be wasting computational resources and other tasks can be executed. Once the operation is completed, the task will be

unsuspended to continue its execution. Rust uses Future traits to represent an uncompleted task and its internal state. The language provides trait and syntax support for asynchronous programming but not the executors or runtime, which drives Future to completion by polling. Tokio [3] is the most popular asynchronous runtime crate in the Rust community for providing such facilities. Tokio also provides many various synchronization primitives that work under asynchronous context, such as mutex, rwlock, semaphores, notify, conditional variables and channels. It is especially worth noting the multi-producer-single-consumer (mpsc) channel supports backpressure by limiting the capacity of the channel, this prevents the receiver from overloading.

3.3 Network Communication

Within the system between modules and also for communication with clients, all communications are done via messages using a TCP connection. Messages are implemented as a Rust enum type, which supports storing data members inside enum variants. Messages are first serialized into JSON format and are then joined into the TCP byte streams, with a header that denotes the length of the message as the frame delimiter. When receiving messages, the entire process is reversed. Rust has a powerful serialization/deserialization crate that not only requires minimal work on the developer side but also allows to change to any arbitrary encoding format, JSON was chosen for now to simplify the debugging and is also very friendly as an external API (with clients). Although we have not found the verbose Json encoding posing network communication bottleneck, we are actively looking into switching to a binary encoding format for communications within the system. The entire encoding and decoding process is done automatically within the Tokio framework, our code just simply needs to specify the serialization and deserialization format and the byte stream frame delimiter.

3.4 Sequencer

The sequencer is responsible for assigning version numbers to every SQL transaction when the scheduler receives a SQL begin transaction request from the client. This request is then forwarded to the sequencer. The exact algorithm on how versions are assigned to tables that are annotated as Write and Read is described in section 2.3. Figure 3 provides an overview of the sequencer system.

Internally, the sequencer consists of two parts, an asynchronous TCP handler and a central state. The handler is responsible for all TCP network communications with the scheduler, where each incoming connection from the scheduler is spawned as a separate task that can be run concurrently. For each SQL begin transaction request received from each connection, the central state will be responsible to assign version numbers to all tables involved in the transaction via annotation. The state is a globally shared variable protected by a mutex. As the system increases the number of schedulers, it is expected to have lock contentions around the state variable. However, the version assignment operation is fast and nonblocking, performance is not expected to be downgraded by much. Once the version numbers are assigned for all tables for a given transaction, the transaction version is sent back to the scheduler by the handler through the same TCP socket where the request was received. The

communication is done in a lock-step fashion, the sequencer processes a single request at a time for every connection, and will not process the next one until the current one is replied.

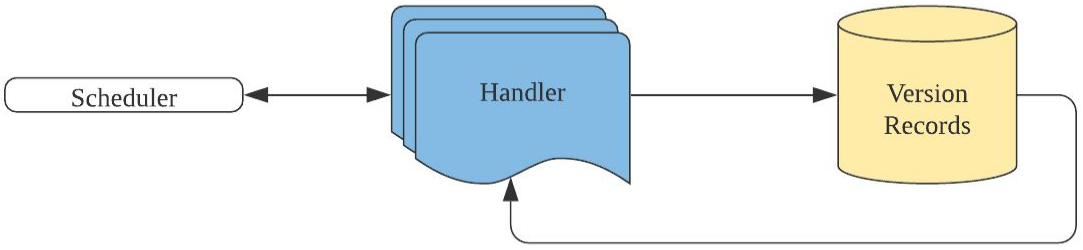

Figure 3: *An overview of the sequencer*

3.5 Scheduler

The scheduler is the controller and driver for the entire system. Externally, it receives SQL requests from clients, processes them and replies with corresponding responses; Internally, it coordinates and sends requests to the sequencer and database proxies. It has a TCP handler processing incoming client requests, a dispatcher that processes SQL request and forwards them to the correct database proxies, a list of transceiver modules each responding for communications with a single database proxy, a logging system that can record all client requests, an administrator handler for administration operations on the system. Figure 4 provides an overview of the scheduler system.

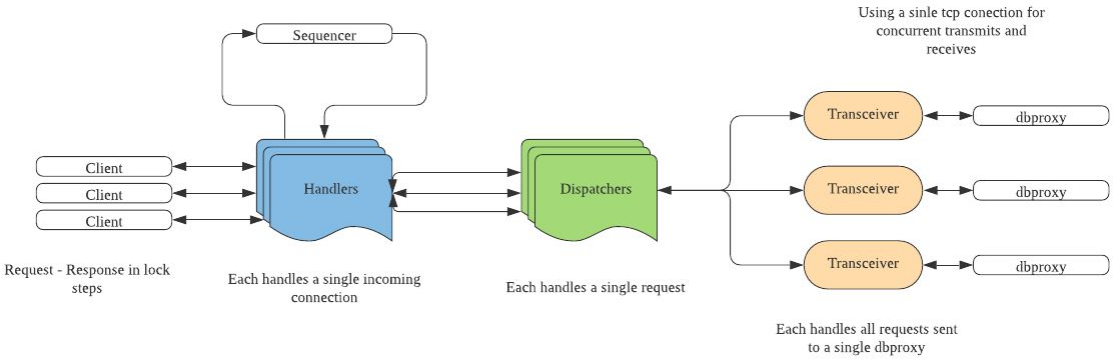

Figure 4: *An overview of the scheduler*

3.5.1 Client API

Ideally, the entire system should function as a black box to the clients as a database management system (DBMS) and DBMS as a client. However, to minimize the extra effort on dealing with DBMS packet protocols, the client now needs to send SQL requests using our API rather than SQL protocol packets, then the system behaves like a proxy that forwards requests and responses between the client and the DBMS. Our API is a JSON text-based

wrapper on the original SQL request, with additional annotations to mark for tables being used in a transaction and a query. Figure 5 shows how the Begin Transaction request looks like, it includes a "tableops" annotation to declare all tables being accessed in the transaction. Figure 6 shows a query request with tables accessed annotated in "tableops" and early release tables in "ertables". It is worth noting we only support early release annotation with Write query requests, so the early release requests in Figure 6 will be ignored.

```
{
    "op":"begin_tx",
    "tableops":"read table0 write table1 read table2"
}"#;
```

Figure 5: *BeginTx Request with Annotation*

```
{
    "op":"query",
    "query":"select * from t;",
    "tableops":"read t",
    "ertables":"t0 t1 t2"
}"#;
```

Figure 6: *Query Request*

3.5.2 Client Handler

The client handler behaves very similarly to that of the sequencer, each incoming TCP connection is spawned as an individual task that can be run concurrently. Every connection also communicates in a lock-step fashion, the next incoming request is only expected and handled after the previous request was properly replied to. When a Begin Transaction request is received from the client, the request is forwarded to the sequencer to acquire the version numbers for the entire transaction, and the version numbers of the transaction are stored within a per-connection state so that following SQL queries from the same client are automatically tied to the current transaction. Communication with the sequencer is also in a locked-step fashion so that a TCP connection pool is used to allow multiple concurrent connections with the sequencer. SQL requests other than the Begin Transaction are forwarded to the dispatcher via a bounded multi producer single consumer (mpsc) channel with backpressure. This message also includes a singleshot channel sender for the dispatcher to reply. The client handler will reply to the client with a response to all types of requests.

3.5.3 Dispatcher

The dispatcher is responsible for forwarding the SQL requests other than Begin Transaction to the correct database proxies. The dispatcher uses an asynchronous design pattern known as an actor [3], which is a standalone task and uses channels for communication with other parts of the system. The dispatcher spawns a new task for each incoming request from the client handler via the mpsc channel. Within each request, the dispatcher keeps an internal state to keep track of all the local versions on each database proxy, so it knows which database proxies to send to.

For Read queries that are within a transaction, the scheduler needs to wait until at least one database proxies have the table versions that allow the Read to happen, following the versioning rules described in section 2.2, if multiple database proxies have the right version numbers, the one with least number of requests is chosen. For Read queries that are not within a transaction (aka, a single Read request), the scheduler does not need to wait on any table versions since there are no transaction versions assigned to this query, indeed, it tries to find a database proxy that holds the most updated versions of the tables involved in the query.

For all Write queries and End Transaction queries, the scheduler will send the request to all database proxies.

After waiting for and determining the database proxies to send the request to, the dispatcher forwards the request to the corresponding transceivers, also with a singleshot channel sender for the transceiver to reply. Only after all requests are sent, the dispatcher will wait concurrently for responses from the database proxies. This is primarily to preserve the relative ordering of requests within the same transaction. It is worth noting only the first responses from any database proxies will be sent back to the client handler via the singleshot channel sender provided. For SQL requests that need to release table versions (such as SQL End Transaction request, or an early release request), the internal state will need to properly release the table versions for each database proxy when its responses are received.

3.5.4 Transceiver

The scheduler spawns multiple transceivers, each is responsible for communication with a single database proxy. It is also using the actor design pattern. Lock-step communication with database proxy is not suitable, since database proxy can potentially block on waiting for table versions and database responses. Therefore, the transceiver communicates with the database proxy asymmetrically. The TCP socket is split into a reader half and a writer half and can operate independently and concurrently.

The transceiver has two event loops, the first one listens on an mpsc channel for requests sent from the dispatcher, while the second one listens on the TCP reader socket. After a request is received from the dispatcher, the request together with the singleshot channel sender for replying is stored into a client-specific outstanding queue. Then the message is sent to the database proxy via the TCP writer socket. While sending via TCP, the transceiver preserves the ordering to be the same as the ordering receiving them from the dispatcher. Once the database proxy replies, the message is popped from the client-specific queue using a request identifier. This lookup is necessary since it is possible to have requests from multiple transactions for a client in the outstanding queue. Finally, the response is forwarded to the dispatcher.

3.5.5 Administrator Features

To facilitate system shutdown, performance tracking, and other features such as blocking all new incoming Begin Transaction requests, the scheduler also binds to an administrator port in TCP. The administrator handler has access to all the shared state variables, so that performance logging files can be dumped. The logging system will record all incoming client requests, responses to the request and corresponding timestamps. For debugging, the logging can be turned to the mode to preserve full records, so that all requests can be replayed; by default, only tags representing types and results of requests are kept along with the timestamp. A system shutdown mechanism is also carefully designed to allow all components to shut down in correct order without hanging. The approach is simple, that no new incoming connections are accepted and as soon as existing connections are dropped, the client handler will drop all its handles to the dispatcher which holds handles to the transceivers. These components will then shut down in the reversed order. When TCP sockets to the sequencer

and the database proxies are dropped, those modules will properly shut down as well in a similar fashion.

3.6 Database proxy

The database proxy is responsible for transmitting queries and responses between the scheduler and the database server. It maintains a separate version number for each of the tables in the database. Each database replica has an individual database proxy, thus each database proxy can have different local versions from each other. All the incoming queries to database servers first arrive at the proxy and have their associated table versions checked against the local replica table versions. If the query contains table versions that do not satisfy the aforementioned constraint in section 2.2, the query is pushed into a queue. Otherwise, the query is forwarded to the database server. Figure 7 provides an overview of database proxy system

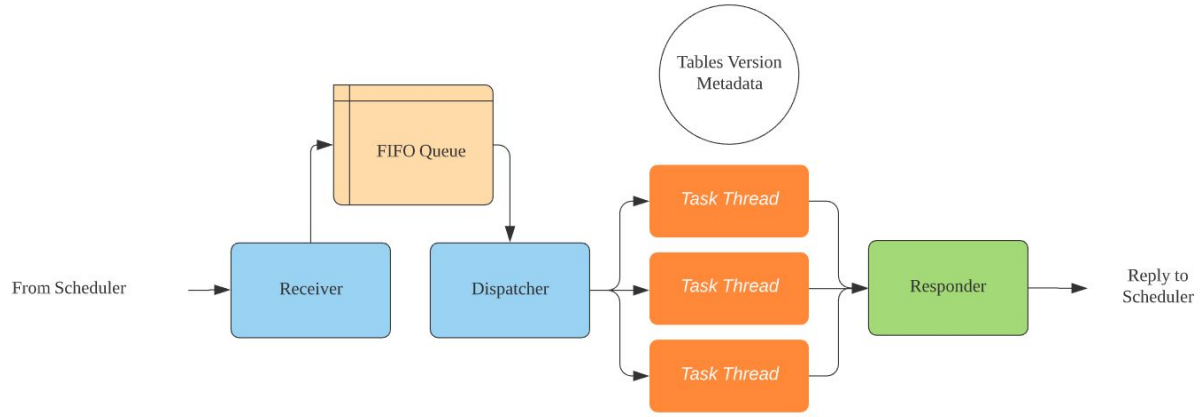

Figure 7: *An overview of the database proxy*

The entire database proxy majorly consists of three functional modules: receiver, dispatcher and responder. Each of them runs on a separate thread concurrently. Each database proxy maintains a consistent TCP socket connection with the scheduler throughout its lifetime.

3.6.1 FIFO Task Queue

As mentioned earlier, queries that are not yet version ready are pushed to a queue. Queries from all transactions are stored in this queue and there could be multiple queries from the same transaction lined up in the queue at the same time. All the queries in the queue are in the order of arrival at the database proxy. Therefore an early query is always in front of the later queries within the same transaction.

3.6.2 Receiver

The receiver is a separate functional module that runs on an async thread, known as the actor design pattern, which is actively listening to the receiving channel of the socket established with the scheduler. All the incoming requests from the scheduler firstly reach the receiver module. Once the receiver receives a SQL query from the scheduler, it produces a task and pushes it to the FIFO task queue directly, and notify the dispatcher about the new task.

3.6.3 Dispatcher

The dispatcher is an actor that runs on a separate thread listening to a notification to pop out the version ready tasks from the queue. The dispatcher gets notified to pop any potential version ready task in the queue in two cases:

- A new task is pushed to the queue
- A version release to any table has happened

For the first case, a new task from the scheduler can already satisfy the version constraint, thus checking is necessary. In the second case, a version release possibly makes some tasks in the queue version ready, and they should be popped out. To ensure the correct order of execution within a transaction, it only pops out the earliest arrived query of each transaction, given the query is version ready.

For each popped task, if it is the first query in a transaction, the dispatcher spawns an async thread to handle asynchronous executions with the database server. The task thread owns a consistent connection with the database until the transaction finishes and the DB connection is recycled into a pool.

A channel is opened between this task thread and the dispatcher, from then on all queries belonging to the same transaction are sent through this channel. On the receiving side of the channel on the task thread, the thread sequentially sends received queries to the database server. Once the DB server responds with a result, the task thread sends this SQL result along with other information directly to the responder and waits for the next query from the dispatcher. The task thread drops after completing a commit or abort query, and the channel is automatically closed.

3.6.4 Responder

Responder, another actor in the system, owns the sending channel of the TCP socket with the scheduler. Responder receives responses from all task threads via a single receiver and multiple sender channel. If the response is related to a commit/abort request, or it contains an early release request, the responder will release the versions on the corresponding tables and notify the dispatcher. Responder then replies the SQL result back to the scheduler through the socket channel.

4. Benchmarking

4.1 TPC-W Benchmark

TPC-W is a benchmark introduced by the Transaction Performance Council (TPC) targeting the E-commerce environment [4]. It specifies an E-commerce workload that simulates the activities of an online bookstore, where customers can browse and order books from the website and administrators can update book information.

Specifically, there are 14 webpages, each considered a database transaction consisting of some number of database queries. These queries each access a subset of a total of database tables, including address, author, cc_xacts (credit card information), country, customer, item, orders, order_line (items corresponding to orders), shopping_cart, and shopping_cart_line (items corresponding to shopping carts). Table 1 below gives a summary of what database tables a webpage accesses with which permission. As demonstrated, 4 out of 14 web pages contain Writes, while the others are Read-only.

A random value is generated at each webpage to determine which webpage is the next to access, by checking against a transition probability matrix. Three probability matrices are available, corresponding to different ratios of Read and Write transactions:

- Mix 1: Browsing mix (95% Read-only);
- Mix 2: Shopping mix (80% Read-only); and
- Mix 3: Order mix (50% Read-only).

Webpage	Read	Write
Admin request	item, author	N/A
Admin response	author, orders, order_line	item
Best sellers	item, order_line, author	N/A
Buy request	shopping_cart_line, country, item	customer, address
Buy confirmation	customer, country	shopping_cart_line, order_line, item, address, cc_xacts, orders
Customer registration	customer	N/A
Home	customer, item	N/A
New products	item, author	N/A
Order inquiry	N/A	N/A
Order display	customer, order_line,	N/A

	country, item, address, cc_xacts, orders	
Product detail	item, author	N/A
Search request	item	N/A
Search result	item, author	N/A
Shopping cart	item	shopping_cart_line, shopping_cart

Table 1: *Relationship between web pages and database accesses*

4.2 Load Generator Implementation

We have implemented a simpler version of load generator for TPC-W-like traffic in Python, adapting from a GitHub repository of a Java implementation [5]. We supported the same web logic and probabilistically determined transitions between web pages, preserving the way how sequences of database transactions and queries are auto-generated. However, front-end support has been omitted, which means webpages cannot be accessed and displayed through browsers anymore.

Each client can be configured with the following parameters: a client ID that uniquely identifies a client, which mix out of the three it will follow to generate the next webpage to access, the maximum time this connection will be, the time interval between two web page accesses. When a client program is started, within the maximum connection time, it will keep accessing new webpages, as generated according to the probability matrix associated with the chosen mix, after the specified access interval. Each webpage access translates to a transaction consisting of some number of database queries. At the beginning of a transaction, a BEGIN operation containing information about tables that will be accessed and their access permission (i.e. Read or Write; if both, record as Write) is sent, while at the end of a successful transaction, a COMMIT operation is sent. In the case of a transaction of only a single Read operation, optimization is to send the Read query only, omitting both the BEGIN and COMMIT operations. Within a multi-query transaction, the result of an older query will be received and processed before a younger query can proceed. An older query's result can determine if later queries should be executed and which query to execute next. Here, there is another optimization opportunity, which is for the older query to identify tables that younger queries will not access and "early release" them by incorporating them into the query. Annotation of tables accessed in each transaction, single-Read transactions, and tables that can be early released are all performed manually.

There are two main reasons to have our benchmark implementation. The first is to bypass handling communication with the SQL server at the TCP protocol level. The original Java implementation communicates to the database server directly by making use of libraries to encode SQL queries from clients into TCP packets as specified by the SQL database server's protocol. Since our system acts as a middle layer between clients and the database server, if we were to use the Java implementation, we would have to be able to decode queries from those TCP packets for processing before re-encoding them to communicate to the database

server, which has proven to be difficult due to special TCP format required by the handshake protocol. The second is that the Java implementation is designed to be a simulator running on a single machine. However, we would like to test our system in a distributed way, with many clients running across multiple machines to generate loads to our system, without overloading a single one.

4.3 Experiment Setup

We deployed our system across multiple UG machines, each with a 4-core Intel Core i7 4790 multi-processor at 3.6GHz and 16GB RAM.

On the software side, the main system (i.e. sequencer, scheduler and database proxies) is written with Rust (version: rustc 1.48.0), using the tokio (version: 0.3.6) library to achieve event-driven, non-blocking asynchronous I/O. The database server adopted is

PostgreSQL (version: 13). The load generator and auto-deployment scripts are written in Python 3.7. Deployment uses the Paramiko library to handle SSH connections.

4.4 Methodology

The two metrics we identified for performance are throughput, measured by completed client requests per second (RPS), and latency, which is the duration between a successful request is received and replied, measured in seconds. We also enabled measurements of those metrics per run per request type (BEGIN, Read, Write, COMMIT).

We would like to see how changing the following parameters will affect the two performance metrics:

- Number of clients;
- Number of database proxies;
- The ratio of Read to Write operations, i.e. varying the probability matrix, to correspond to browsing mix, shopping mix and ordering mix, respectively;
- With and without the early release optimization;
- With and without the single Read optimization.

As a result, we identified the following five experiment categories, each sweeping across 10 different client numbers (1, 5, 10, 20, 40, 80, 100, 200, 400, 800) and 7 database numbers (1, 2, 4, 8, 16, 32, 64):

- Category 1: Browsing mix, with both optimizations;
- Category 2: Shopping mix, with both optimizations;
- Category 3: Ordering mix, with both optimizations;

- Category 4: Browsing mix, without the single Read optimization;
- Category 5: Ording mix, without the early release optimization.

Browsing mix is chosen to evaluate the effect of the single Read optimization as it contains the highest rate of Read operations, thus the highest amount of single Read operations. The ordering mix is chosen to evaluate the effect of the early release optimization as it contains the highest rate of Write operations, which will see more speedups than Read operations.

For each 3-minute experiment run, we deployed the sequencer, scheduler and each of the database proxies in release mode, to different machines, respectively. All clients ran distributedly across a randomly selected subset of all available machines. We have monitored the CPU and network utilization for runs requiring the most resources, and ensured that machines were not overloaded.

Since database servers are expensive to set up, both in terms of population time and storage space, we generalized a per-request-type latency model tuned with real latency measurements at low database server count (5) to simulate performance measurements for systems with up to 64 database servers. All data presented in Section 5 come from systems with varying numbers of database proxies connecting to this simulator based on real measurements, rather than real database servers.

5. Results

In this section, we compare the peak throughput across the five experiment categories as mentioned in Section 4.4. More precisely, we consider the peak throughput as the highest throughput across all data points whose corresponding latency at the same client count stays within an acceptable threshold. Furthermore, we consider the latency threshold to be around 0.2s.

5.1 Performance with both optimizations

5.1.1 Browsing mix

As shown in Figure 8, the overall peak throughput occurs at 4 databases with 800 clients. Compared to the peak throughput at 1 database with 800 clients, there is a 29% improvement.

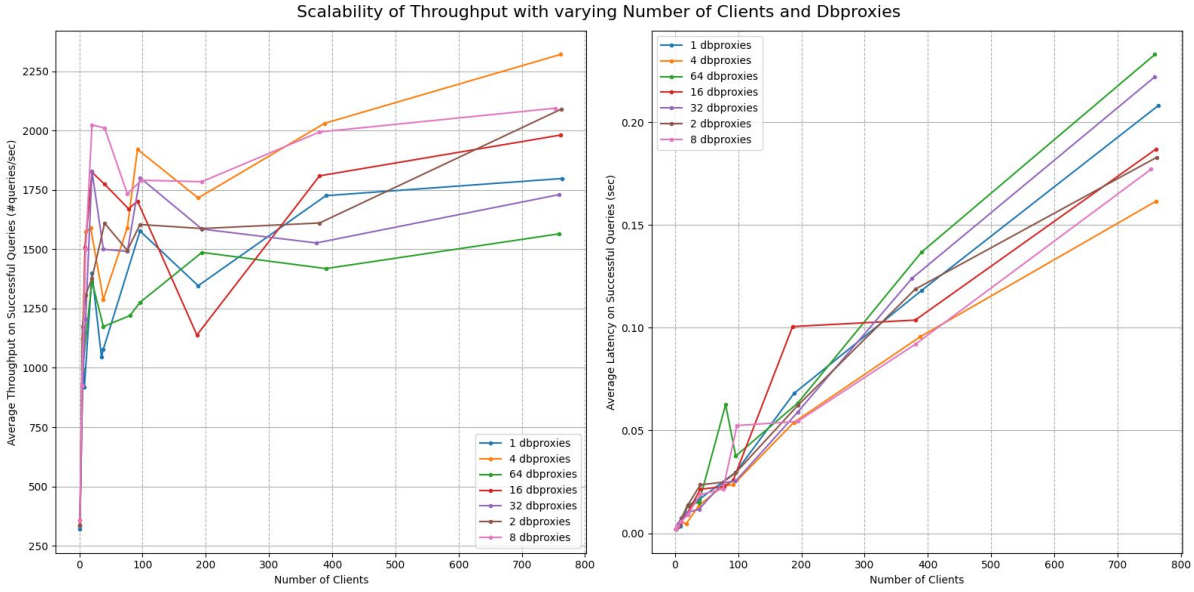

Figure 8: *Performance of experiments in Category 1 - Browsing mix, with both optimizations*

5.1.2 Shopping mix

From Figure 9, it can be identified that the peak throughput is located at 16 databases with 100 clients. It demonstrates a 34% increase compared to 1 database with 100 clients.

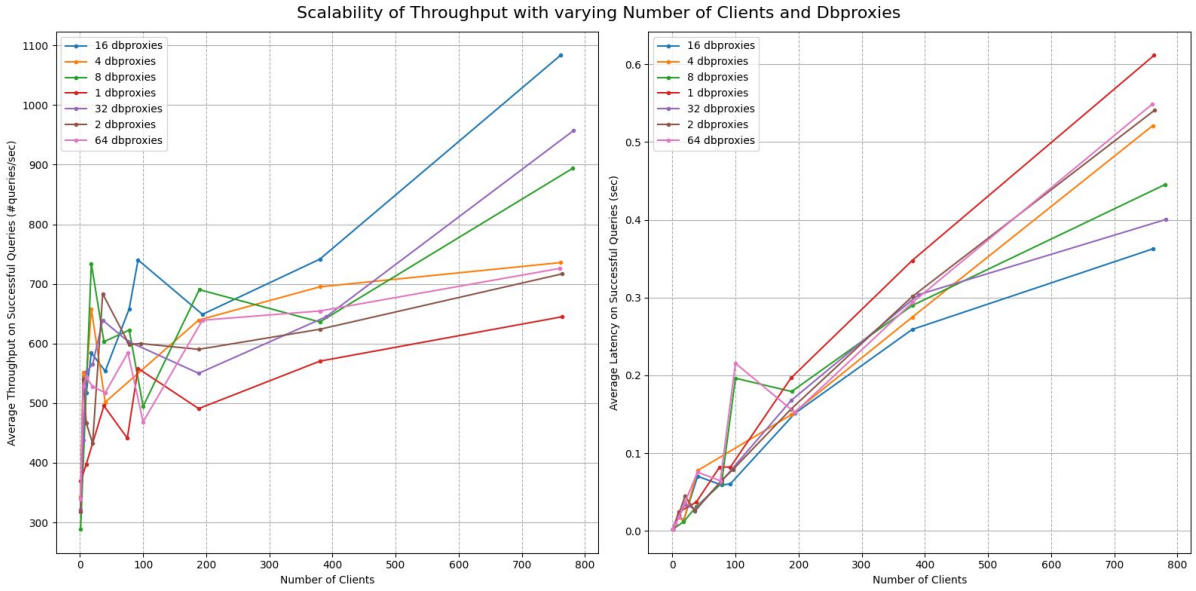

Figure 9: *Performance of experiments in Category 2 - Shopping mix, with both optimizations*

5.1.3 Ordering mix

As shown in Figure 10, 8 databases with 20 clients demonstrates peak throughput, which is 22% higher than 1 database at 1 client.

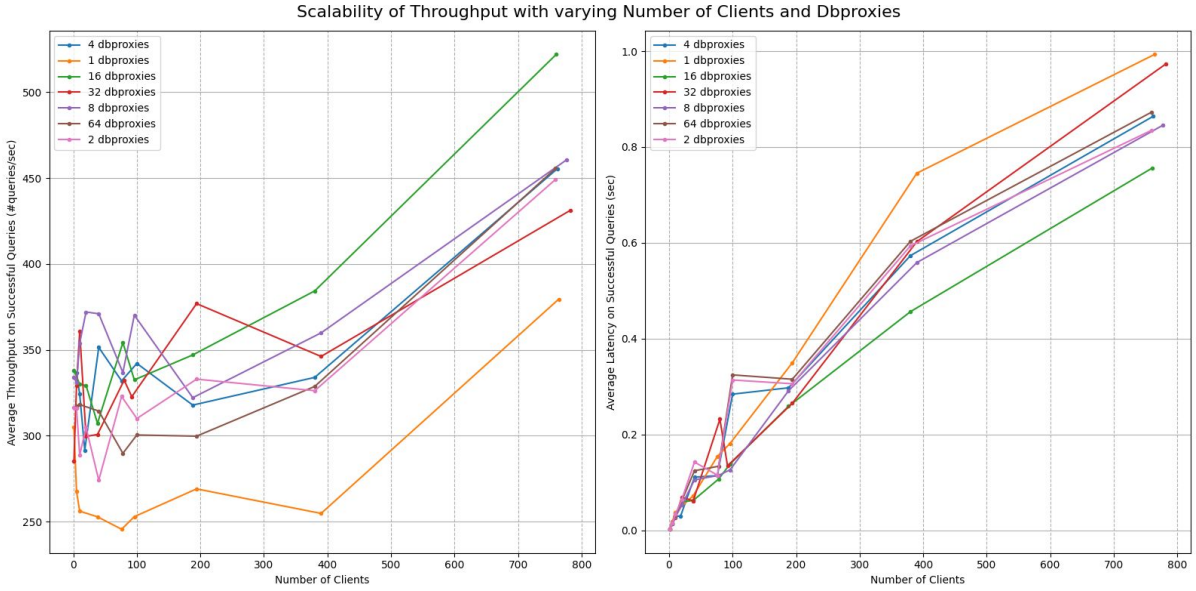

Figure 10: Performance of experiments in Category 3 - *Ordering mix, with both optimizations*

5.1.4 Trend as the number of database replicas increases

Based on Figure 8-10 above, it is noticed that although increasing database count can improve peak throughput, the highest throughput across all database numbers appears at

moderate database counts rather than the highest database counts. We attribute this to the fact that we allocate each asynchronous I/O event to a thread from a worker pool. As database count grows beyond a certain number but the number of cores at the scheduler stays the same, communication with too many database proxies incurs too much overhead, diminishing the return of being able to serve client requests from alternative machines.

5.1.5 Trend across Mixes

Across browsing, shopping and ordering mixes, the ratio of Read to Write operations decreases. At the same client count, the latency increases and the peak throughput drops. This trend is as expected, as multiple Reads can execute simultaneously, but only one Write is allowed at a time, leading to longer execution time per Write.

5.2 Comparison between with and without single Read (browsing mix)

Comparing Figure 8 and 11, at low client count (below 100), runs with single-read optimization demonstrated higher throughput overall, while at higher client count, disabling this optimization yields better performance. As a result, we believe having this optimization or not does not affect the performance of our system much, considering the amount of noises existing in our deployment environment, consisting of machines shared by many students at University of Toronto.

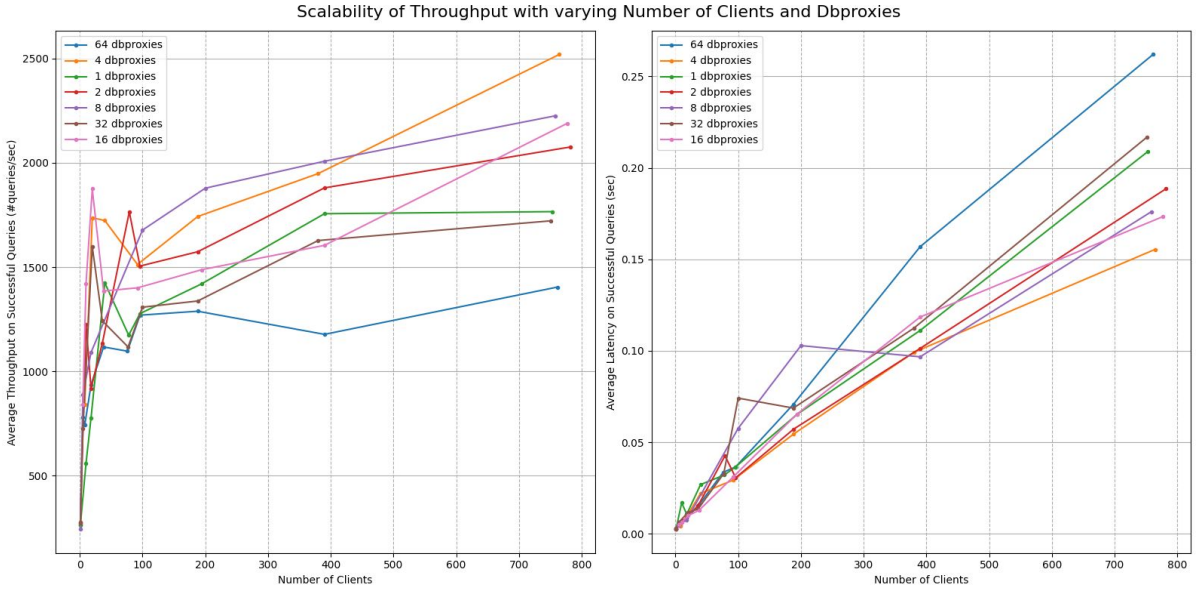

Figure 11: *Performance of experiments in Category 4 - Browsing mix, without the single Read optimization*

5.3 Comparison between with and without early release (ordering mix)

Comparing Figure 9 and 12, supporting the early release optimization seems to have resulted in worse performance. We believe this is due to the small amount of Writes with early-released tables, causing the added complexity in optimization implementation to provide no obvious return.

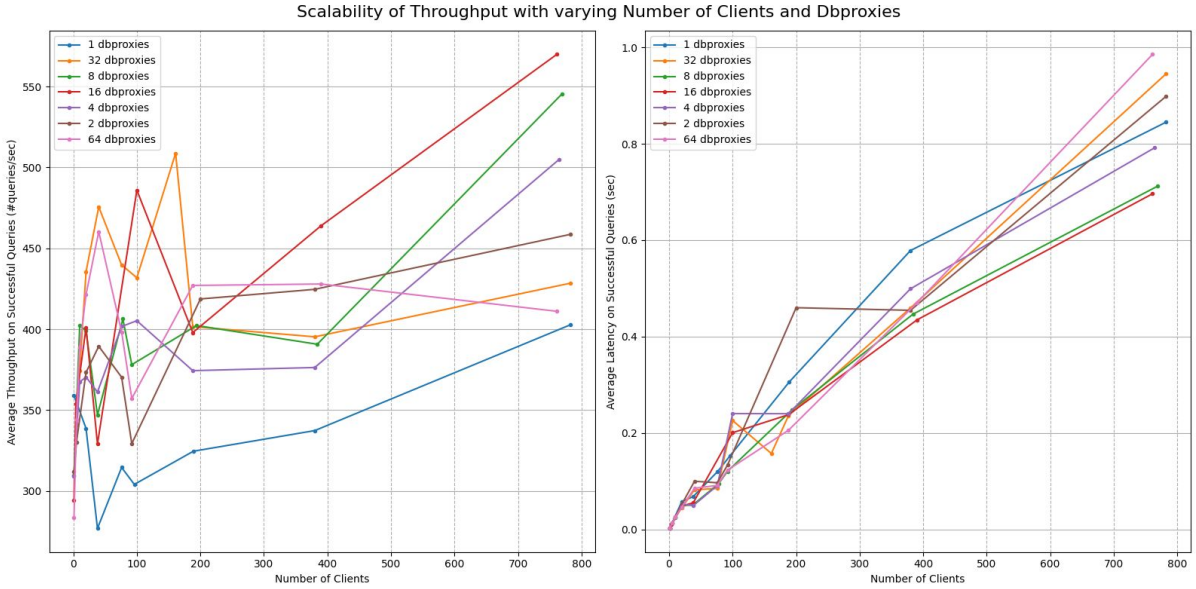

Figure 12: *Performance of experiments in Category 5 - Ording mix, without the early release optimization*

6. Conclusion and Future Work

The algorithm and module-level behaviour were referenced by paper [1]. Our contribution is mainly implementing the distributed versioning algorithm described by paper using the programming language Rust and with asynchronous programming. We implemented features presented in the paper such as early version release and single Read optimization.

Using client load generated following an E-commerce benchmark, TCP-W, within reasonable latency, our system demonstrated around 30% improvement in peak throughput at 4-16 database replications, compared with only a single database instance. A higher ratio of Read to Write operations in the client load has shown to have a positive impact on the performance of our system. However, two optimizations, supporting transaction-free single-Read and the early version release of database tables, do not result in obvious return on performance, due to the small amount of requests affected by them and noises existing in the benchmarking deployment environment.

A potential bottleneck of our system is currently all communication is passing through a single scheduler, in reality, it puts a performance restriction on the total number of database proxies and clients the system can handle. This explains why the system suffers a significant throughput drop when the number of database proxies increases to a certain extent. Due to timing constraints, there are still many features left unimplemented:

1. Adding regression tests to ensure functional correctness of the system. There are many potential functional bugs uncaught during this prototype development.

2. Adding native support to use SQL's network packet protocol as the client API, so that the entire system can behave like a true blackbox to the client. By doing so, it also allows minimal modification to the client code.

3. Currently Read and Write SQL queries require manually annotation for its operation type, which not only is inconvenient but also very error-prone, a great next step is to be able to parse the SQL syntax and automatically create the annotation for tables involved.

4. To further improve the scalability of the system, supporting multiple and distributed schedulers are essential, this can greatly improve the bandwidth to clients.

5. Adding fault-tolerance capability and being able to recover from errors.

6. To productize the system, more administrator features are essential, for example, more loggings, better remote admin interface.

Reference

[1] C. Amza , A. L. Cox and W. Zwaenepo, “Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Web Sites”, in *Middleware 2003: Proceedings of ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 16-20, 2003.

[2] “Rust,” *Rust Programming Language*. [Online]. Available: https://www.rust-lang.org/. [Accessed: 19-Dec-2020].

[3] “Glossary,” *Tokio*. [Online]. Available: https://tokio.rs/tokio/glossary. [Accessed: 17-Dec-2020].

[4] Transaction Performance Council. *TPC-W: Benchmarking An Ecommerce Solution.* (2005) . [Online]. Available: http://www.tpc.org/tpcw/

[5] S. Abeysinghe and S. Helmini. *A Java implementation of the TPC-W benchmark.* (2019). [Online]. Available: https://github.com/supunab/TPC-W-Benchmark