UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Improving the performance of a Conflict-Free Replicated Relational Database System

Sondre E. Brynjulfsen

INF-3981 Master's thesis In Computer Science – June 2023

Abstract

SynQLite is a relational database (RDB) built on SQLite. Using conflict-free replicated relations (CRR) to apply conflict-free replicated data types (CRDT) to augment the SQLite database. With SynQLite sites update their local replica without coordination from others sites, and sites converge once they have applied the same sets of updates [1]. These sets of updates are fetched from the CRR and History tables generated by SynQLite during CRR initialization. To keep the tables updated SynQLite uses triggers associated with the insert, update, and delete statements. From a previous project it was found that with the current setup of triggers the performance of SynQLite was almost 94 times slower in the TPC-C benchmark compared to an SQLite database using no triggers. The project concluded that it was the use of triggers that was the cause of the performance issues [2].

In this thesis the focus will be shifted from finding the problems to finding a solution to improve the performance. While the use of triggers is the cause of the problems they cannot be fully removed. SynQLite requires some data to be stored automatically to correctly merge two, or more, sites. To find a potential solution an iterative design process is used. The different solutions will be tested using a custom benchmarking tool and py-tpcc by Andy Pavlo, which implements the TPC-C benchmark in Python [3]. The results from these benchmarks are used to make decisions on what solutions to dismiss or continue work on. During the thesis a different solution to the current implementation was found which takes inspiration of SQLite's Journals. Using the new solution saw an almost 110 times improvement in the TPC-C benchmark, reaching just over 88% of the SQLite's performance.

Acknowledgements

First off, I would like to thank my supervisor Weihai Yu who has guided me through both my projects, the capstone and Master's Thesis. Being able to work alongside him and discuss ideas and solutions has been a wonderful experience, and I am grateful for the opportunity I have received. Yu is an excellent professor and knowledgeable in his field and I feel I have learnt a lot from him the last year.

I would also like to give a huge thanks to my family and friends I have made along the way. The amount of support they have provided me has helped motivate me throughout my education. I will never forget our, maybe too long, breaks playing table tennis.

Finally, I would like to give a special thanks to my brother, Kim. When I first started my education, he helped me settle and acquaint myself with the city. He has without doubt been a person I could rely on through thick and thin. If it were not for him, I am not sure any of this would be possible, again, thank you.

Table of Contents

List of Figures

List of Tables

1 Introduction

Due to an ever-increasing demand of data, it is vital that the current database systems can keep up. In 2020 it was estimated that a total of 64.2 zettabytes of data was created and used globally. Over the next five years, this number is expected to triple, reaching over 180 zettabytes of data. Even though only a few percentages of the new data is being kept it would still amount to 3.6 zettabytes in 2025 [4]. While not one database would be responsible for this amount of data in the near future, it shows a rapid increase in data creation. Having a high performing database system would therefor seem like a necessary requirement for any database system in the future.

Selecting the correct database management system (DBMS) is therefore an important first step. Depending on the use-case there are multiple options. If the need to support big datasets, e.g., Big Data, is required databases such as PostgreSQL or MySQL are good options due to their client-server architecture. Some systems, however, do not always have access to the server, yet still always needs access to the database. For these types of situations, options like SQLite are a perfect candidate. One such system, SynQLite, which this thesis builds upon, uses SQLite for this exact reason.

Even though selecting the correct database engine is important, it is mainly after the selection is over that the real optimization process can begin. Some of these are writing an efficient API for communicating between the application and database, writing efficient queries, e.g., for reading from the database or triggers used for automation, and optimizing the database schema. If any of these are not up-to-par it can create a noticeable delay for the end user.

In a previous project [2], it was discovered SynQLite was suffering from some of these problems. In particular, an unoptimized database schema with trigger queries was the main problem severely decreasing the write throughput. From the results it was shown that compared to a standard out of the box SQLite database SynQLite was on average 93.9 times slower using the default journal mode running the TPC-C benchmark. Running a custom benchmarking tool inserts and updates saw an exponential growth in time spent as the number of rows increased, while delete had more of a linear growth. These numbers should however be taken as is due to reason discussed in section 4.1.

This thesis will extend on what was previously discovered and will attempt to improve the performance of SynQLite as best as possible. The majority will be focused on improving the triggers used for keeping the database up to date. Other areas of importance include improving the database schema and finding other improvements. For the end user the change would not be noticeable except for the performance improvements.

1.1 Goals

When entering the project, a main goal, as well as a few secondary goals, was created to ensure it was clear what was going to be achieved. The success of this thesis will be rated depending on the goals reached. Improving the performance of SynQLite is a broad task and more of an overall goal, rather than a main or secondary goal. Therefor for this thesis the main goal was set to:

Increase the write performance of SynQLite's insert, update, and delete operations to match an out of the box SQLite database as closely as possible.

Since SynQLite does not do anything to the original application database tables the read performance will match of a normal SQLite database. In addition to the main goal, three secondary goals were created.

1. For any existing SynQLite system, the change should be unnoticeable in any way except write performance.
2. The outcome of any insert, update, or delete operation should be the same as the current SynQLite implementation.
3. Existing SynQLite features, e.g., clone, push, and pull, should see as minimal impact on performance as possible.

2 Methodology

When improving the performance of any system, decision is often based on numbers making them purely objective. A pure objective decision system would be preferential although subjectivity will occur. This can come in the form of readability and maintainability of the solution, e.g., writing clean code to make it easier to expand later. Writing clean code does not necessarily equal a worse solution but means decisions are not always purely objective.

To make any decision a four-step iterative process was used.

1. Research and create a hypothesis which would improve the write performance.
2. Implement a quick solution to be tested.
3. Perform benchmarks on the new solution.
4. Compare the new results with previous results from other solutions.

If the new solution saw an improvement over previous solutions it was saved to either be used in future implementations, or to be further improved.

2.1 Time Plan

Choosing a strategy for research, implementation, and experimentation is important to ensure the success of any project. An agile approach was selected for this thesis due to a few factors: (1) While the problem area was already uncovered, information about it was still unclear at the beginning. As more research into the workings of SQLite and SynQLite was done newer and better solutions would be uncovered. Including more, but smaller, work packages would allow for more rapid experimentation and development. (2) Unforeseen events can occur in any project. Not setting a fixed long-term goal would allow for more flexibility. (3) Using an agile approach seemed like a fitting strategy due to bi-weekly meetings acting as sprint reviews.

Alongside the agile approach work packages were planned at the beginning of the thesis as seen in Table 1. Work packages was split into months and would count as goals for the month.

Work Package	Description
January	Exploring alternatives to triggers in SQLite and coming up with potential solutions.
February	Starting with the implementation and testing of different solutions developed in January.
March	Updating the benchmarking program for faster and easier benchmarking. Carrying out an in-depth performance evaluation and analyzing the performance characteristics of the different solutions.
April	Implementing the final solution to SynQLite and performing an in-depth performance evaluation to be used in the report. Start writing on the report.
May	Finalizing the project and the report.
June	Delivery of the master's thesis.

Table 1: Work packages for thesis

3 Technical Background

3.1 SQLite

SQLite is a library which implements a self-contained, serverless, transactional SQL database engine with no extra configuration required. A main feature of SQLite is its no extra server requirements meaning it can run locally, with all data stored in one file. All writes are written directly to disk, and in some instances SQLite operations can be faster than I/O operations. SQLite fully supports ACID even in the event of failures, e.g., a system crash [5]. SQLite is a relational database (RDB) which organizes data into tables, columns, and rows with a full-featured SQL implementation for querying against the data [6].

3.1.1 Queries and triggers

All features offered by SQLite makes it a clear choice for SynQLite, although there are still important features missing. SynQLite being heavily reliant on triggers makes SQLite's lack of a full trigger implementation noticeable. Currently only row-level triggers are supported by SQLite. Row-level triggers will execute the trigger once for each row being inserted, updated, or deleted [7]. For example, given a transaction where 10 000 rows are inserted the trigger is executed 10 000 times. The other alternative is statement-level triggers. Statement-level triggers are triggers which are only executed once per statement. Given the same example, instead of executing the same trigger 10 000 times it would only execute the trigger once [8]. While statement-level triggers offer better performance than row-level triggers it cannot give the same level of detailed control as row-level triggers.

3.1.2 Journal Modes

To achieve atomic commit and rollback capabilities SQLite uses a temporary file, called a rollback journal, to store data during writes. Rollback journals are created at the start of a transaction, and depending on the journal mode the behavior of what happens to the journal at the end of a transaction changes [9].

SQLite supports six different journal modes. Journal modes are set by using PRAGMA statements. A PRAGMA statement is an extension to SQLite allowing for modifying the behavior of operations [10]. By default, the journal mode for all transactions is set to DELETE. Once a transaction completes the temporary file is deleted and needs to be recreated for the next transaction. Unlike DELETE, TRUNCATE and PERSIST keeps the journal between

transactions. TRUNCATE truncates the journal to zero-length, and PERIST overwrites the journal header with zeros. A more common option, alongside DELETE, is write-ahead logging (WAL) [11]. WAL works by inverting the process of a journal. Instead of copying the original database into a separate file, and then writing the changes into the original file. WAL keeps the original file as is, then writes the changes into a temporary file [12].

Although not recommended, SQLite allows for storing the rollback journal in memory or completely turning it off. To store the journal in computer memory the journal mode MEMORY can be used. While giving an increase to performance it comes at the cost of safety and integrity. If the journal mode is set to OFF SQLite cannot longer guarantee atomic commit or rollback capabilities, and any event which stops the transaction will likely corrupt the database [11].

3.2 SynQLite

SynQLite is an extension of SQLite which allows for synchronization between multiple sites. A site is a local database stored on any device. SynQLite uses conflict-free replicated relations (CRRs) to apply conflict-free replicated data types (CRDTs) to RDBs. The usage of CRDTs allows sites to update its local replica without requiring any coordination from other sites. SynQLite guarantees Strong Eventual Consistency (SEC) between sites. SEC is achieved by guaranteeing that the state of all replicas converges once they have applied the same set of updates [1]. Currently SynQLite's access point interface (API) offers several functionalities. The main ones being init, clone, push, pull, and sync. The names are inspired by existing hosting services, e.g., GitHub, for familiarity.

3.2.1 CRR Initialization

A main feature of SynQLite is its ability to work with any SQLite database without requiring any extra addons. Using the 'init' command an SQLite database is converted to SynQLite database. By default, seven (7) tables are created upon initialization.

Table name	Shortened name	General description
crr_meta__clock	Clock table	Contains the time in microsecond and nanosecond of the last update to the database.
crr_meta__site	Site table	Contains metadata of all existing sites, including the local one.
crr_meta__site_state	Site state table	Contains state information of current site.
crr_meta__history	History table	Contains the history of all actions executed by the database. An action being an insert, update, or delete.
crr_meta_islocal	Local table	Contains a boolean of whether updates propagated to the database is local or remote. Used to increase performance by disabling triggers, e.g., when migrating with another database.
crr_meta__resolution	Resolution table	Contains temporary information used while synchronizing the local database with remote databases. Used to preserve integrity constraints.
crr_meta__res_attr	Resolution attribute table	Contains temporary information used while synchronizing the local database with remote databases. As with the resolution table, it is used to preserve integrity constraints.

Table 2: Tables created by default on CRR initialization.

While this thesis does not touch most of these tables it is important to have a general understanding of the reason these tables exist. In addition to these tables two tables per existing table are created. Existing tables being the original tables in the database before the CRR initialization. For this thesis the existing tables will be referred to as original tables (OTs).

Table name	Shortened name	General description
crr__table_name	CRR table	Contains time of last update for each column, their values and causal length.
crr_op__table_name	CRR op table	Used together with the Resolution table for repairing integrity constraints.

Table 3: Tables created for each OT.

Table 3 uses “table_name” as an example name. The example name would be replaced with the real table names. For example, given a database with the table ‘students’ it would generate the tables “crr__students” and “crr_op__students”.

3.2.2 Updating CRR tables

SynQLite relies heavily on triggers to keep all tables up to date. Per OT five triggers are created. Three triggers are created to update the corresponding CRR table (CRR triggers), and two triggers are created to update the history table (History triggers). Figure 1 shows how updates are propagated throughout the database as an any insert, update, or delete operation is executed against an OT. After any transaction CRR triggers are activated to update the crr table, which in turns activates the History triggers to update the history table.

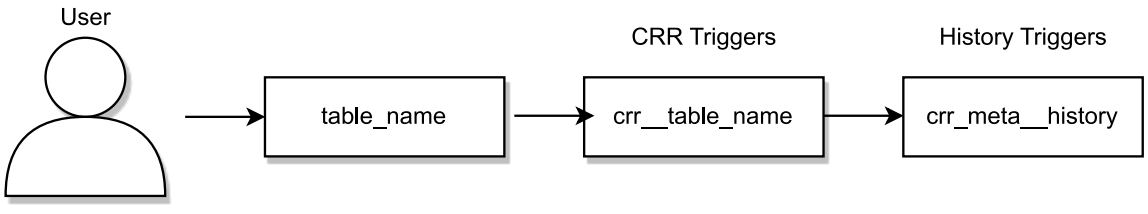

Figure 1: Trigger logic on any insert, update, or delete.

3.2.2.1 CRR Triggers

The insert trigger is made up of three queries. The first query updates the clock table to reflect the time of the latest change. Only the ‘ms’ column in clock table is updated by this query. A separate trigger updates the ‘ns’ column as talked about in section 3.2.3. The next two queries are to make sure the new entry is correctly inserted into the CRR table. First a query is run to check whether a row with the same primary key(s) already exist in the CRR table. If the row is not found the next query will succeed and the row will be successfully inserted. If the row exists, its values will be updated as seen in Figure 2. Checking for existing rows with the same primary key removes redundant rows and ensures the OT and CRR tables are equal in row placement.

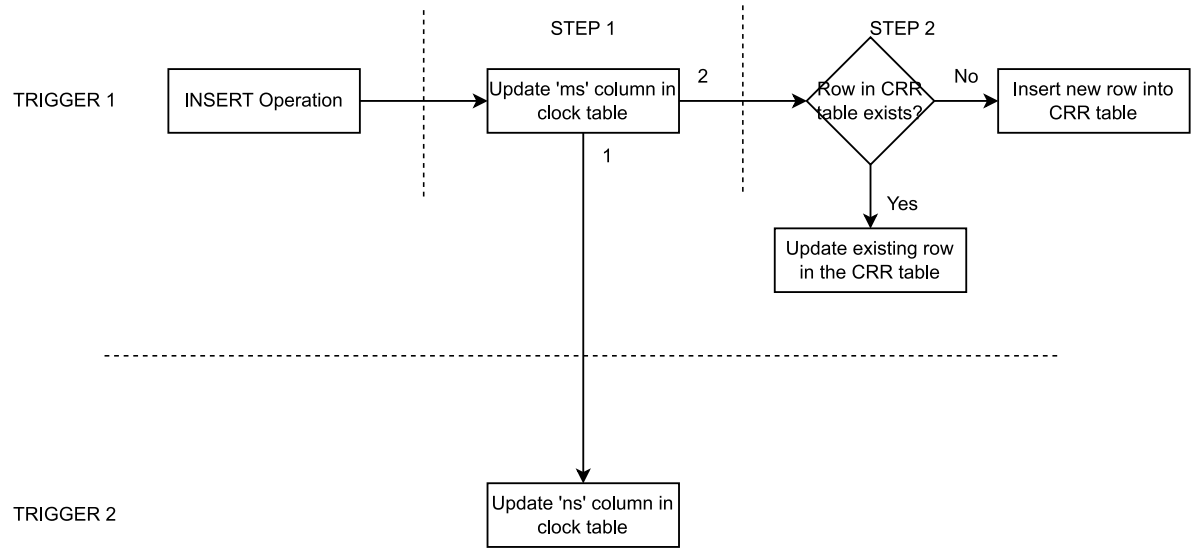

Figure 2: Insert trigger.

As seen in Figure 3 the delete trigger is made up of two queries. The first query is equal to that of the insert trigger. Then the next trigger updates the corresponding rows causal length (CL) value in the CRR table. SynQLite uses CLSet, where each row has an associated CL value, to keep track of the number of updates to the row. The CLSet is used during migration of two SynQLite databases to ensure updates are propagated equally. The causal length set is a natural value used to identify the causality between inserts and deletes [13]. During an insert the CL value is either incremented by one or set to one depending on whether the row exists or not. When a row is deleted the CL value is always incremented by one. The reason for this choice is that insert and delete operations always happens in turns [1]. To tell whether a row exists or

has been deleted is done by checking whether it is even or odd. Even rows do not exist, and odd rows do.

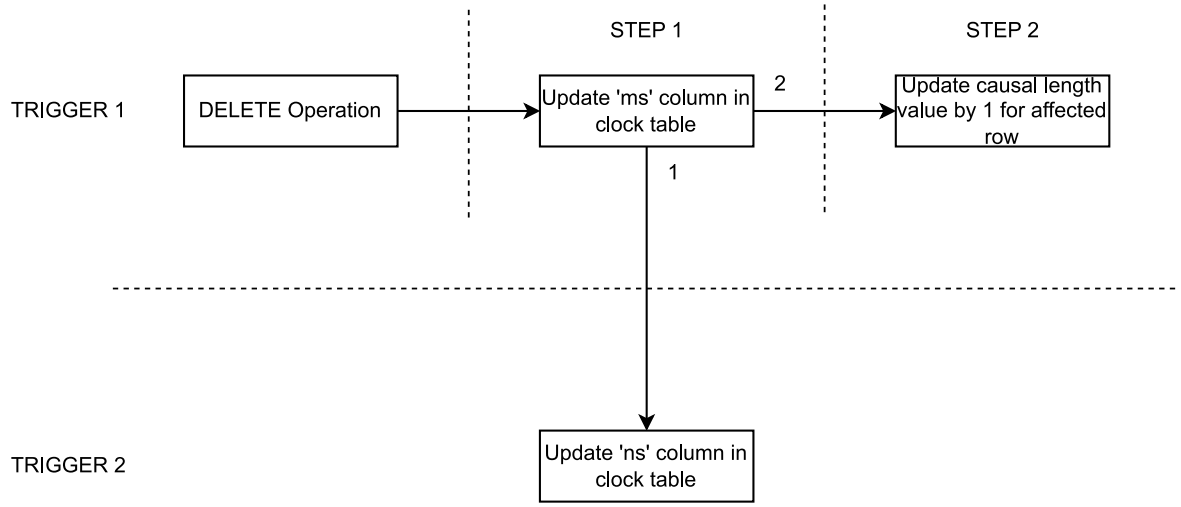

Figure 3: Delete trigger.

The update trigger is made up of two queries as seen in Figure 4. Like the previous two it first updates the ‘ms’ column in the clock table. Then the second query uses the values found in the updated row to update the corresponding row in the CRR table. In addition to updating the column values, the timestamp columns associated with each column in the OT is updated to reflect the time of update. Each column has their own timestamp column meaning these values can be different from each other. Only the columns which was updated has their timestamp values updated. The update trigger does not change the CL value unlike insert and delete.

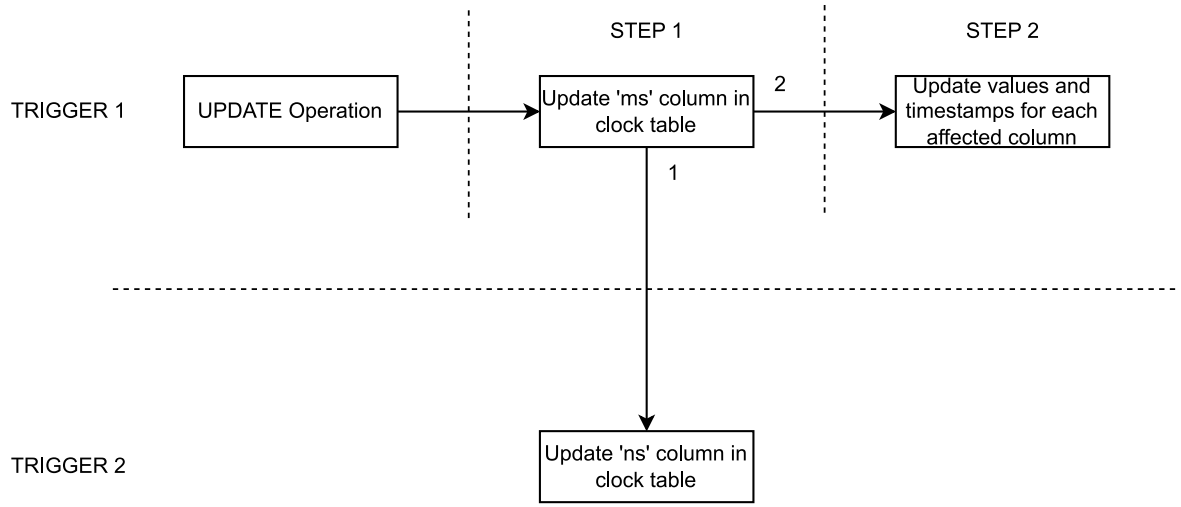

Figure 4: Update trigger.

3.2.3 Updating Clock table

A commonality between all CRR triggers is the updating of the clock table. The clock table is made up of two values, a millisecond and nanosecond column. SynQLite uses the clock table to solve two issues. SQLite only supports timestamp values at a sub millisecond level, and a physical clock is not enough to represent the happen-before relationship between updates [1]. Using the clock table timestamps in nanosecond resolution is implemented. Updating the clock table is a two-step process. First the initial trigger sets the 'ms' column to the current time converted to milliseconds, then another trigger associated with the clock table updates the ns column. The ns value is decided using the 'case' operator. If the millisecond value has not changed in the clock table, e.g., if the updates happen too quickly, the nanosecond column is incremented by a random value. The value used to increment is generated using Equation 1 where *R* is a random value.

$$ns_{new} = ns_{old} + |R \; mod \; (1000000 - ns_{old})|$$

Equation 1: Nanosecond increment value generation.

If the millisecond value has changed since last update the nanosecond value is generated using Equation 2.

$$ns_{new} = |R \; mod \; 10000|$$

Equation 2: Nanosecond value if millisecond does not match.

3.2.4 Updating History table

The History table has two triggers associated with it, one for inserts and one for updates to a CRR table. In essence both triggers perform the same actions as seen in Figure 5. Both triggers update the History table and the Site State table.

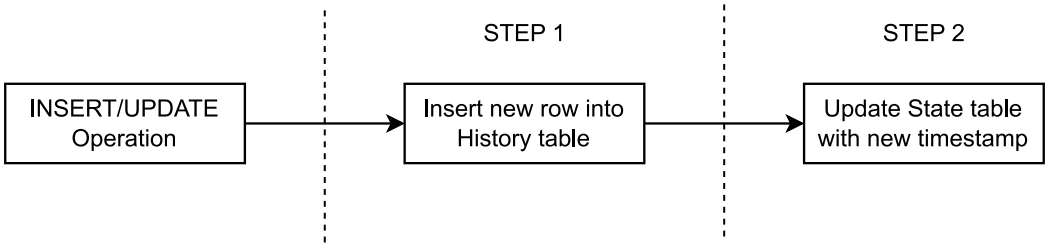

Figure 5: History table triggers.

The insert trigger inserts a new row into the History table, and the update triggers updates the values for the corresponding row. Rows are matched using a primary key made up of three columns: the table name, the CRR row uuid, and the site id. The next query is equal for both triggers. To update the Site State table a query fetches the latest millisecond and nanosecond value from the clock table. These values are combined and replaces the previous value for the 'last_out' column for the current site id.

3.2.5 Counting triggers and table lookups

Given a transaction which includes 100 inserts, 10 updates, and 25 deletes. Equation 3 is used to calculate the total number of triggers, and Equation 4 is used to calculate the total number of table lookups. Where *i* is the number of inserts, *j* is the number of updates, and *k* is the number of deletes.

$$N_t = 3(i + j) + 2 * k$$

Equation 3: Total number of triggers in each transaction.

$$N_l = 7 * i + j(5 + c_t) + k$$

Equation 4: Total number of table lookups in a transaction

Using the example above with a table made up of four columns, a total of 380 triggers and 815 lookups would be required to update the CRR and History tables.

3.3 Benchmarking Tool

As explained in Section 2, all decision is mostly objective as they are decided by numbers. The subjectivity comes from the readability and expandability for future implementations and modifications. All numbers are generated from two benchmarking tools. (1) A custom made benchmarking tool made specifically for this thesis, and (2) a TPC-C benchmark written in Python by Andy Pavlo [3].

3.3.1 A Custom Benchmarking Tool

The benchmarking tool made for this project is a re-write of a benchmarking tool made for a previous project [2]. An entire re-write was done due to the unnecessary complexity and problems with the previous benchmarking tool. The new tool includes most of the previous

features in addition to some new features. Why the tools were rewritten and what is replaced is elaborated further upon section 4.1. This section will briefly explain the current features, and how the benchmarking tool works.

3.3.1.1 Running tests

Currently the benchmarking tool supports a wide variety of arguments to run tailored benchmarks. The general arguments include which benchmarks to run, what journal mode to use during query execution, and the filename of the results file. Arguments used during runtime include the minimum and maximum number of tables, columns, and rows to iterate between, the number of rows to increment per iteration, whether to add the same number of rows per table or give each table a random number of rows, and how many times to repeat the test.

To make it easier to visualize how the benchmark works an example with the given parameters shown Table 4 will be made. First a Database object is made with the given journal mode. This object stores all information needed during runtime such as the path to the database file, database name, values to check if database was correctly initialized before testing, current tables, their column, and their datatype. Then the test will run through all possible combinations of the given parameters. In the given example it would total 405 tests. For each iteration the test remakes the database to ensure each test is as similar as possible. Although it is possible to disable this feature and let the database grow for a more realistic simulation. Row values are different per run to make the result as general as possible. After an iteration is over the database is deleted.

Parameters	Parameter value
Benchmarks	All
Journal Mode	DELETE
[Minimum tables, maximum tables]	[1, 3]
[Minimum columns, maximum tables]	[1, 3]
[Minimum rows, maximum rows]	[0, 500]

Row increment	100
Random divide	True
Iterations	3

Table 4: Example parameters used during testing.

Measuring the performance of each test is done by storing the time used during query execution. This does not include the time spent generating the queries or populating the database if needed.

3.3.2 TPC-C

Transaction Processing Performance Council Benchmark C (TPC-C) is an on-line transaction processing (OLTP) benchmark. TPC-C was made to simulate an example of an industry that sells, distributes, and manages a product or service. It is not meant to be a guideline for a product supplier but rather a potential representation of one. The benchmark is built on the concept of terminal operators which execute transactions against a database. These transactions include queries such as entering or delivering new orders, recording payments, checking the status of current orders, and monitoring the stock level of each warehouse. A warehouse contains all stock which are supplied out to districts. Each warehouse must supply at least ten sales districts, and each sales district must serve three thousand customers. Customers make orders which drain stock from warehouses which must try to maintain stock for its 100 000 items. TPC-C was created as a replacement for TPC-A, but has in turn been replaced by TPC-E although it is still the most widely used OLTP benchmark of the options given by TPC [14].

3.3.2.1 Python TPC-C

Python TPC-C is an open-source project by Ando Pavlo to benchmark different database engines using the TPC-C standard. Originally the project was written for Python 2 but has been modified to run with Python 3. Other modifications have been made to the source code, e.g., for storing of results or running with different journal modes. These modifications were done in a previous project “Benchmarking a Conflict-Free Replicated Relational Database System” which found the performance problems this thesis deals with [15]. It should be noted that py-tpcc was written by Ando Pavlo and has only had slight modifications done by the author. No attempt is made to claim the program as self-written and all credits goes to the creator [3].

4 Design and Implementation

4.1 Improving on the previous benchmarking tool

This section is meant to cover the important changes to the previous benchmarking tool. While sufficient for tracking down the reasons for SynQLite's performance issues it had its share of flaws. Optimization was a mostly neglected when writing the original version. There were two main reasons why this was the case.

1. There was not a set design for the benchmarking tool except for an idea of what it should do. Therefor during the entire project, the tool saw major revisions. Spending the time to optimize each version would have wasted too much time, and it was seen as a better choice to keep the unoptimized version while testing. At the end there was not enough time to optimize the final version.
2. At the end of the project the entire benchmarking tool was cumulation of different ideas and designs. This made it difficult to work with and understand. To optimize the tool would require a complete rewrite.

4.1.1 Removal and replacement for previous features

With the new version came removal of previous features, improvements of old features, and introduction of new features. Features such as multi-client benchmarking, benchmarking of time to clone, push, and pull, and minor features such as purging previous databases on startup were removed. Multi-client benchmarking was removed due to never being a fully implemented. In addition to SynQLite being a local-first database meaning it will mostly be accessed by one client a time, and the solo-client performance was not great in comparison to a regular SQLite database. The ability to the benchmark time to clone, push, and pull was replaced with a new benchmark. The new benchmark is meant to test the final solution, as discussed in section 4.5. Another reason was due to the conclusion reached in the previous project of benchmarking SynQLite. The project concluded that the slow-down was caused by triggers [2], and therefore it was more important to build the benchmarking tool to measure the aspects impacted by triggers.

4.1.2 Better debugging tools

A missing component of the previous benchmarking tool was the ability to properly see what was going on. To improve upon this issue two new features were added with the new version.

(1) The ability to display all triggers in the database. At times it can be hard to keep track of all triggers in SynQLite, and especially when they are being constantly changed to test new ideas and implementations. With the use of SQLite's Schema table, a function queries the table for all triggers and the tables they are connected to. All triggers are sorted using their connected table. A loop then prints the table name, and all triggers associated with the table and their names as. Figure 6 shows an example of the output for one table in SynQLite. This feature was used to ensure the correct triggers were in place before running benchmarks.

```
table name : crr_meta__clock


trigger name : crr_meta__clock_update_trigger
CREATE TRIGGER crr_meta__clock_update_trigger
    AFTER UPDATE ON crr_meta__clock
    BEGIN
      UPDATE crr_meta__clock SET ns = CASE WHEN OLD.ms = NEW.ms
                THEN OLD.ns + ABS(RANDOM() % (1000000 - OLD.ns))
                ELSE ABS(RANDOM() % 100000)
                END;
    END
```

Figure 6: Example of displaying triggers.

(2) Enabling and disabling triggers. Disabling triggers allows for testing an unmodified SQLite database using the same benchmarking tool. Which version to use during benchmarking is decided with a 'version' argument. Using the same benchmark makes results more comparable to each other. The results are also stored in the same format as other results which in turn allows them to be easily plotted against other results. Toggling triggers on and off was made possible with the new test environment created for this thesis. How the test environment works is discussed in section 4.1.3.

4.1.3 Creating a custom test environment

Previously the only way to test new solutions was to change the source code of SynQLite. If a promising solution was found there were three options. (1) Store the code in the same function and comment out or use if statements to switch between them manually. (2) Create a separate

function to store the solution in. (3) Create a separate file to store solutions in. Both options 2 and 3 require manual selection as with option 1. All options made testing slow and tedious, as well as error prone.

To combat these issues a custom test environment was created. The use of triggers is known to be the root of SynQLite's performance issues. Therefor the table and trigger creation part of SynQLite's CRR initialization was implemented as its own program. Creating a custom test environment allowed for faster and easier testing, more flexibility and customizability, and made for an easier work environment due to the file being less cluttered with unnecessary functions.

4.1.4 Improvement on existing features

A flaw with the previous benchmarking tool was how rows were divided over tables. Row dividing is used by the insert, update, and delete benchmarks. Insert uses it to ensure no more than the specified number of rows are divided over all tables. Update and delete use it to ensure they do not update or delete more rows than what exist in a table. This did not work as expected and had the potential of giving inaccurate results. Usually this was not a problem due to the benchmark overpopulating the tables. The new benchmarking tool keeps track of the number of rows in each table allowing for more accurate and faster testing.

To improve execution time the new version takes more advantage of SQLite queries. Instead of querying for all existing rows, and then using a Python library to select random rows to update or delete, all rows to be modified are selected with same query.

4.2 Updating the plotter for more comprehensive graphs

As with the benchmarking tool, the plotter has seen some changes as well. For some of the same reasons as mentioned in section 4.1, the plotter was rewritten. A noticeable improvement to the plotter is the restructuration. A proper structure made it easier to use, work with, and expand. As a result, more runtime arguments were added to make each run tailored to what was needed.

4.2.1 Arguments

Arguments include: (1) The names of result files. Only the files given will be used to generate graphs and tables. Using this argument select comparisons between two or more results can be created without hardcoding it into the program. (2) The ability to turn of plot and table creation. It is not always necessary to generate plots or tables. Having the ability to toggle either off both speeds up the process and reduces clutter. (3) Adding prefixes to file names. Prefixes are currently only applied to tables as plots are guaranteed unique names. Tables have fixed names such as *txns*, *baselinediff*, and *relativediff*. Adding a prefix allows for differentiating tables from each other and prevents tables from being overwritten.

4.2.2 Plot and Table creation

Unlike the previous version plots are not generated as data is being read, but instead after all data has been processed. After processing, three different plot- and table categories are generated. For plots, these categories include: (1) Individual plots per result set. A result set is one set of times for a benchmark, e.g., an insert performance benchmark. (2) A comparison of all individual results for the same benchmark. These plots show the performance difference between different journal modes. (3) Database performance comparisons. Being able to plot the performance difference between an unmodified SQLite database, the original SynQLite, and the potential solutions, was crucial for seeing the full picture during testing. Only the journal modes DELETE and WAL are plotted, although all modes are supported. These were selectively chosen due to them being the most likely choice for transactions. DELETE is the default journal mode, and WAL does not need to be set for each transaction unlike the others. Examples of the plots generated can been seen in Figure 7.

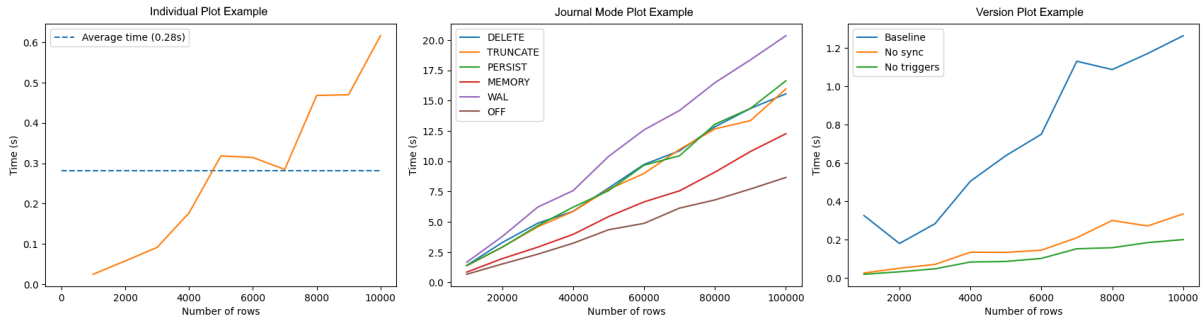

Figure 7: Examples of generated plots.

Tables, or comparison tables, are tables comparing the transactions per second (TPS) for individual benchmarks. There are two types of comparison tables. (1) Tables comparing the TPS between different journal modes on the same benchmark, and (2) tables comparing the TPS between different database versions. Unlike the first category, and the version graph plotting, only DELETE is used to compare different versions. An example of a comparison table can be seen in Table 5.

Journal Mode	Insert	Update	Delete
DELETE	14325.316	6348.944	10822.741
WAL	12714.937	4925.964	8985.512

Table 5: Example table generated from results.

4.3 Exploring alternatives to triggers

Due to the conclusions draw in section 5.2. One idea was to remove the use of triggers altogether. Removing triggers would put SynQLite's performance on-par with what SQLite can achieve. Unfortunately, this idea was quickly rejected due to SynQLite being dependent on triggers. One trigger for each operation is required for two main reasons: (1) SynQLite needs a CL value associated with each row to correctly merge two databases. (2) Removing triggers would require the OTs to be modified to include a CL and datetime column. This would both require a trigger to update the CL and datetime values and goes against SynQLite's idea of leaving tables as is.

4.4 Experimenting with different Clock designs

One area that could easily be improved upon was the Clock. As explained earlier the clock is made up of two parts, as seen in Figure 8. The first trigger updates millisecond column, and a second trigger updates the nanosecond column. By using two separate triggers the same table is accessed twice for each insert, update, or delete just to update the column values. The clock was optimized in two steps: (1) Combining the two triggers into one. (2) Changing the behavior of the clock. In the second step three designs were proposed, implemented, and tested.

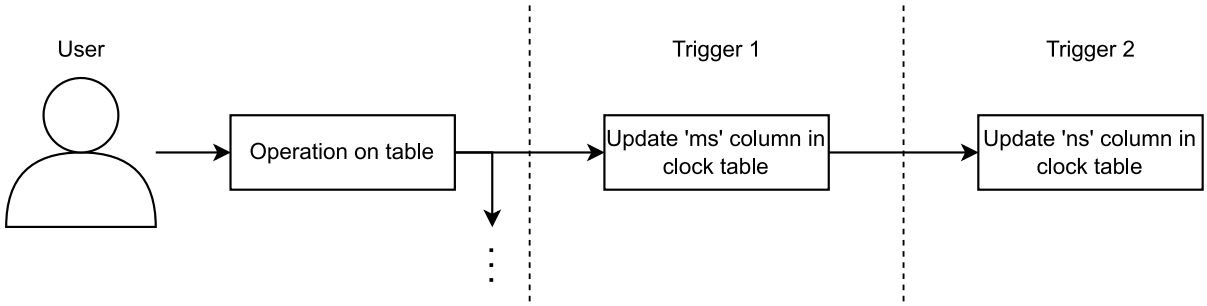

Figure 8: Original logic for updating Clock table.

4.4.1 Combining Clock triggers

The clock trigger is fired after any update to the Clock table. Any insert, update, or delete will update the millisecond column of the Clock table. Therefor it makes no difference when the nanosecond column is updated, as it is guaranteed to be updated. The secondary trigger can therefore be removed, and the logic added to clock update part of the insert, update, and delete triggers. Looking at Table 6 shows removing a trigger gives a noticeable increase to performance. With a ~1.35 times TPS increase for inserts, ~1.13 times increase for updates, and ~1.45 times increase for deletes. All while keeping the original logic. A downside with this approach is the need to access the Clock table to read the previous millisecond value. The next section covers different designs to see if there is more room for improvement.

	Insert	Update	Delete
Baseline Clock	16375.538	8027.833	12515.235
Combined Clock	22065.224	9086.849	14330.926

Table 6: Baseline clock TPS versus Combined clock TPS.

4.4.2 Changing Clock behavior

Except for combining the two triggers no other implementation saw the same improvement. Three designs were tried, both alone and in combination with each other. (1) Reducing the randomness. The nanosecond column can have a maximum value of 1 000 000. If the new millisecond value is equal to the previous millisecond value a new value is generated and used

to increment the previous nanosecond value. This value will always be less than the difference between the current nanosecond value and the max value. The idea was to see the impact generating a random value has instead of incrementing the nanosecond column by a fixed value. After all, the nanosecond column just needs to increase. In this case the fixed value was set to 1 but can be any value that does not increase the nanosecond column above its max value.

	Insert	Update	Delete
Combined Clock	22065.224	9086.849	14330.926
Fixed Value	22115.755	8467.229	14730.624

Table 7: Baseline clock TPS vs Reduced randomness TPS.

Table 7 shows no improvement to the TPS for the three operations. Update even has worse performance with the change. There are two possible reasons for this. (1) Only some of the randomness is removed. If the millisecond values do not match a random value is still generated. There might be an improvement if this was also set to a fixed value, but from these results there is no guarantee. (2) Fluctuations happens during benchmarking. Steps were taken to ensure a minimum amount of background tasks were running, but the same performance cannot be guaranteed throughout the entire run. This is discussed further in section 6.1.

(2) Removing the CASE statement. In theory, checking if two values are equal, one of which must be fetched from disk, should have an impact of performance. Therefor two different designs were implemented which does not use a CASE statement. The first design uses the previous idea of incremented the nanosecond value with one, and the second uses the same equation as is used to generate the millisecond value.

	Insert	Update	Delete
Combined Clock	22065.224	9086.849	14330.926
Baseline new clock plus one	25283.597	7346.56	14252.643

Baseline clock real	25997.325	8297.872	11349.581

Table 8: TPS difference for Clock triggers with no CASE statement.

As with the previous results, the same pattern emerges. Insert sees an improvement to TPS, while update and delete has lower TPS. From these results it can be concluded that the only worthwhile improvement is to combine the two triggers.

4.4.3 An important observation

During the experimentation with different designs for the clock triggers an observation was made which laid the foundation for the final solution: Insert, update, and delete does not need to constantly update the Clock table. This does not mean the Clock table is not necessary as it is required by other functionalities, e.g., merge, and cannot be completely discarded. What is necessary is that during such an operation the last update is recorded in the Clock table. Finding a solution that preserves this behavior but removes the constant updating of the Clock table is a positive step forward in improving SynQLite's performance. As it will greatly reduce the number of disk writes per operation. A potential solution to this problem is discussed in section 4.5.

While the proposed solution does not update the Clock table for each insert, update, or delete the data gathered from the previous experiments were not wasted. Keeping track of when a row, or a specific column, was lasted updated is still crucial for correctly applying updates. Using what is learned allows for creating effective solutions to generate these time values.

4.5 An approach based on SQLite's Journals

4.5.1 A Brief Introduction

SynQLite's performance issues stems from the triggers used during an insert, update, or delete. The main method to improve the performance is to make each trigger as light as possible. By being light involves doing the minimal amount of work necessary without over-complicating other processes or breaking them. A solution found during this thesis manages to avoid both these two problems while keeping triggers lightweight. The inspiration for the design comes from SQLite's Journals. More specifically, the journal mode: TRUNCATE.

4.5.2 Designing the Journal

To understand how the concept of a Journal was created it is important to understand the job of the current triggers. On any operation, e.g., an insert, the trigger will update the CRR table and History table. Then if another database makes a pull request, or the current database makes a push request, it has the necessary information of the current state and its history. Without these it is not possible to apply the same set of updates to all databases. This would break the consistency property. Except, this information is only needed when such a request is made. An unspecified amount of time can pass before the database is synchronized, e.g., the device does not have the ability to make or receive request for a prolonged amount of time. During this time millions of transactions can be made, overwriting, deleting, or inserting rows. Having triggers updating the CRR and History tables for each operation is a needlessly costly operation.

The Journal would therefor act as a temporary ledger keeping the minimum amount of information necessary to update the CRR and History tables once any request is made. While having an idea of what the Journal would do. Knowing how to design it and use it properly took several iterations. During the design process two different journals, as seen in Figure 9, were proposed: (1) a local journal. These journals would each be assigned to an OT, just like how CRR tables are assigned to each OT. (2) A global journal. Instead of keeping track of multiple, smaller, journals a single journal keeps track of all that has happened since the last database synchronization.

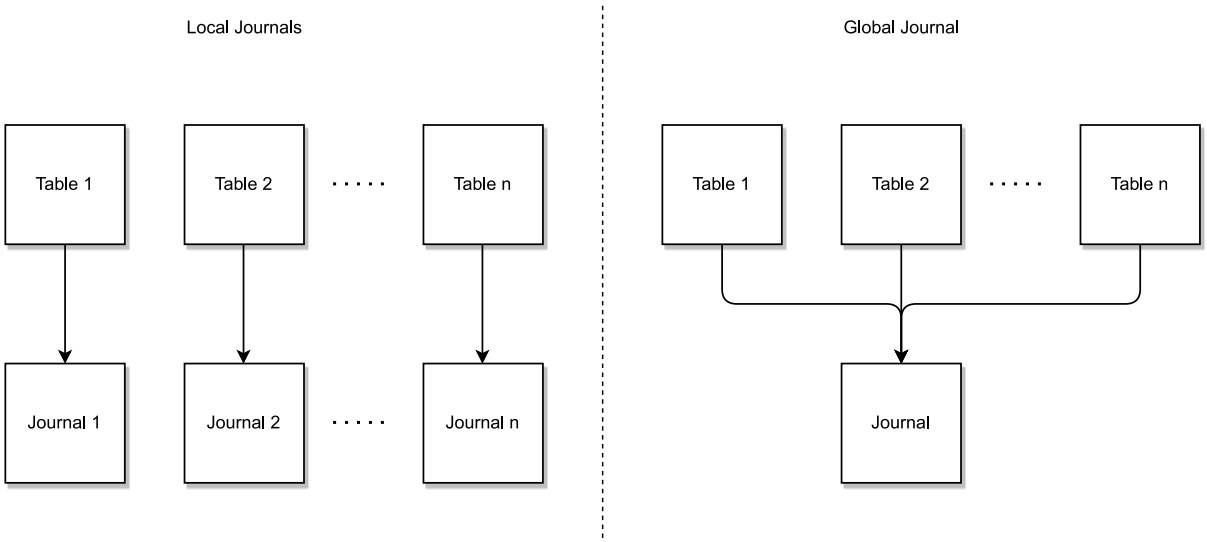

Figure 9: Local versus Global journals

Neither design is perfect, and both have advantages and disadvantages. Local journals have better durability over global journals. While SQLite fully supports ACID these can be turned off using different journal modes. If a transaction were to be corrupted the entire journal would not be lost. Another advantage to local journals is their size. It is not necessary to store the table name for each row, as it is only one table which can update the journal. While not a massive difference with enough rows it can be noticeable on low-space systems. The disadvantage with this design is its increased complexity. Keeping track of multiple journals is not easy as it cannot be stored in memory. If devices shut down the memory clears the program must query to get the journal names. Having one journal with a specified name removes this issue. The advantage of durability is also not an issue. Most systems would not use journal modes which disables the ACID property. A single journal also means less disk reads due to less queries being required to get all the information. This does, however, require more memory to store all the data. Querying multiple tables takes less memory, but today's systems are almost guaranteed to have adequate memory.

After weighing the pros and cons of both designs, local journals were ultimately decided as the only possible solution. Setting aside the other properties such as using local journals makes querying a lot simpler and faster, as querying a big, unsorted, table many times is more expensive. The previously discussed advantages and disadvantages ignores one important aspect of SynQLite. SynQLite requires a timestamp column for each column in the OT. A global journal would not be possible as each table is, most likely, different from each other.

The potential solutions for making a global journal work are too complicated and will most likely be slower than using local journals.

Deciding on the type of journal was just the first step. The next step was to find out what information should be stored. A requirement for the Journal is that it needs to have the information required to successfully update the CRR and History tables. Only the CRR table requires the CL and row values from the OT. The rows to be inserted into the History table are generated from static information, e.g., site id and table names, and values known from the CRR table. Therefor the required information to store is the row id, CL value, and time of last update for the specific column. It is also possible to store the values of the row.

With this comes the design decision: Store the maximum amount of information, or the minimum amount of information. Both options have their advantages and disadvantages. Storing the row values in the journal tables requires less table lookups during synchronization but creates a lot of redundancy. Keeping the same information stored in three tables, the OT, Journal, and CRR table is wasteful. Especially since the number of columns per OT is different for each table. Storing the minimum amount of information was therefore the best option.

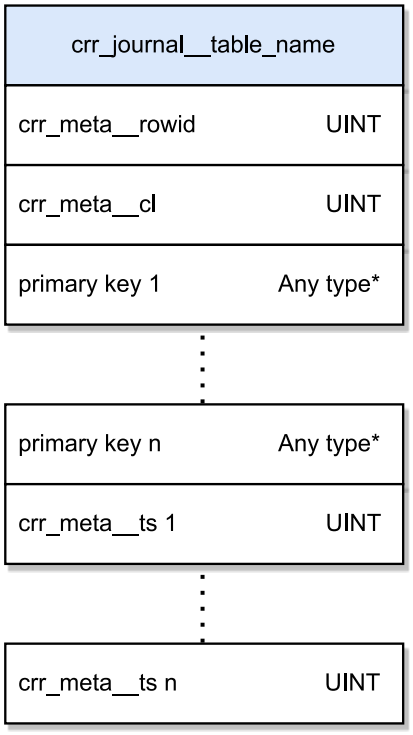

Figure 10: Journal Table Schema

Figure 10 shows the table schema for the Journal. The row id column keeps track of the row id of the row that was inserted, updated, or deleted. Without knowing the row id of the row which was inserted, updated, or deleted it is impossible to know the values of each row. For insert or delete the CL value for the associated row is either set to a specific value or updated. While in the original SynQLite implementation the CL value was only updated by an insert or delete, it is possible for an update operation to set the CL value as seen in Figure 11. For insert, if the row already exists in the Journal, it means that it has previously been deleted from the OT. For update and delete, having the row already exists means it has either previously been inserted or updated. Update can set the CL value in the case the row has yet to be updated after the last journal synchronization.

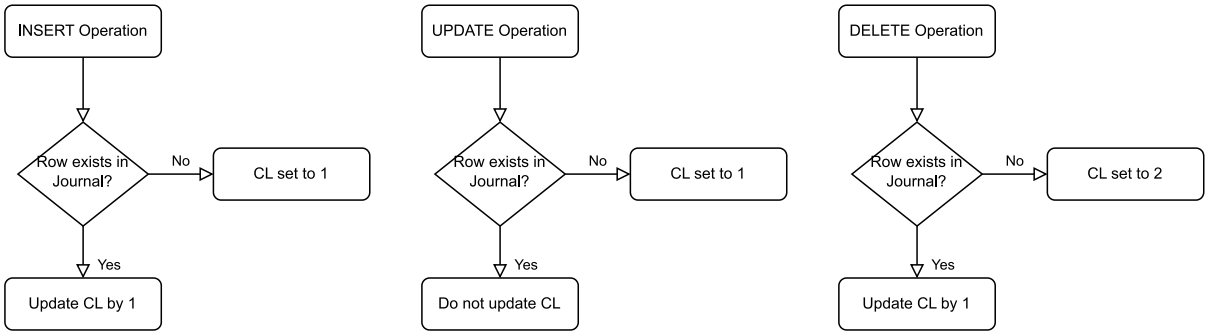

Figure 11: Deciding on the correct CL value for the Journal.

As described later in this section, row ids cannot be used as a connector between the Journal and CRR- / History tables. To connect these rows the same solution as SynQLite uses was implemented. SynQLite connects rows using their primary keys as they are guaranteed to be unique. While SynQLite continuously updates the CRR table containing these values the new solution does not. And as the same row in the Journal can be updated multiple times before a synchronization is executed, it is necessary to store the original primary key values to find the correct row in the CRR table. These values are therefore only set when the row is first inserted into the Journal. Any consecutive update to the row will not update these columns.

As shown in Figure 10 the Journal supports any number of timestamp columns up to SQLite's maximum of 32 676 [16]. To prevent any features from breaking after implementing the new solution both the CRR and History tables are kept as is. Different designs were tested for both tables but ultimately the original designs were deemed the best in the end. By using the original designs avoids the need of re-writing how SynQLite performs migrations. A design with the

CRR table introducing a row id column to connect the OT, Journal, and CRR table was tried but found flawed. The problem with using the row id is that they are only locally unique, and not globally unique and would cause issues when attempting to merge two sites.

4.5.2.1 New Triggers

With all information now being stored in journals, all three triggers had to be updated. The new triggers are significantly less complex compared to the previous triggers, and mostly equal. A row has two states, unchanged and changed. Therefor all triggers need to try to update the row, and if it does not exist insert the row into the journal. The logic of updating CL values is still the same as with the original SynQLite implementation. An insert or update inserts a row with CL value of one. If the row already exists insert updates the value by one, and update does not change it. If a delete inserts a row, it will set the CL value to two to indicate the row has been deleted, else increment the CL value by one if it already exists. How the CL value is correctly set after synchronization is described in section 4.5.3.1.

4.5.3 Journal Synchronization

In the current version of SynQLite on any request all data is pre-processed and can be sent right away. The new implementation requires a component placed in the middle of the request receival and data sending. Figure 12 shows a simplified overview of the change to the request pipeline. For this thesis this component will be referred to as the Journal Synchronization Unit (JSU).

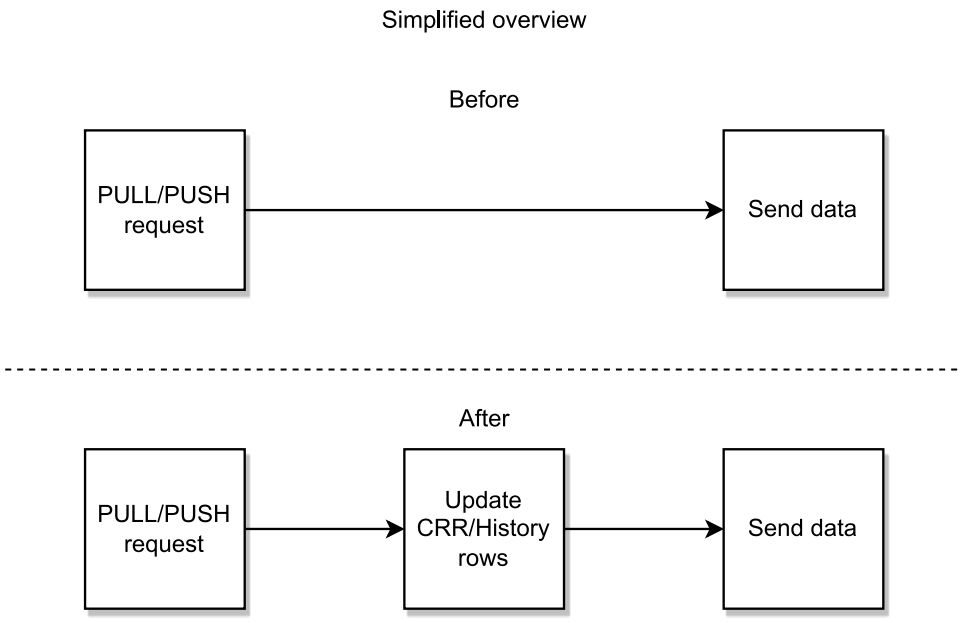

Figure 12: SynQLite pipeline on requests

Before any row generation begins the JSU first queries for the site id as it is needed for the last step. The site id will not change, and it is therefore only necessary to do this once. Due to how the journal is designed the JSU must iterate over each table and perform three main queries. The first query fetches all rows that exist in the OT, the Journal, and the CRR table. These are rows that have been previously synchronized and only needs to be updated. The query works by first performing an INNER JOIN between the Journal and the CRR table using the original primary key values. An INNER JOIN is a join which only selects the rows both tables have in common. A second INNER JOIN is then performed against the OT using the row id column.

The second query fetches all rows that only exist in the OT and the Journal. These rows are guaranteed to be local row and only requires to be inserted into the OT. The query first performs an INNER JOIN between the OT and the Journal using the row id column. To remove any rows that exist in the CRR table a WHERE statement is used. It selects all rows where the primary key values cannot be found in the CRR table.

The last query fetches all rows that only exist in the Journal and the CRR table. First the query performs an INNER JOIN between the Journal and CRR table using the primary key values. To remove any rows which still exist in the OT it only selects rows in the Journal which has a CL value that is divisible by two. These are rows that have been deleted locally but still need to be updated due to them potentially existing on other sites. Any row that only exists in the

Journal is guaranteed to be local rows and has been deleted. These can be discarded to save space.

The reason for the queries to be split up is to simplify the queries made for inserting/updating the CRR- and History table. With the way the data is fetched and sorted only two queries are needed. Both queries are REPLACE queries which during testing was found to be the fastest alternative over INSERT and UPDATE queries. After all queries and values have been generated two steps remain. The first step is to update the Site table's *last_out* column to the current time, and then wipe the Journal. Wiping the Journal between each run is based on SynQLite's TRUNCATE journal mode. Although there are no consequences for not wiping the Journal between each synchronization, as the same rows are generated. Wiping the Journal ensures only the rows changed since last synchronization will be updated in the CRR and History tables increasing the performance.

4.5.3.1 Correctly updating the CL value

The last challenge is to figure out how to properly update the CL value for existing rows in the CRR table. If a row already exists in the CRR table, it will have a CL value assigned to it. Rules for how to correctly combine the two CL values can be created by making a truth table.

	Journal CL Value	CRR CL Value	Expected CL Value
Example 1	1	1	1
Example 2	2	3	4
Example 3	1	2	3

Table 9: Truth table for updating CL values.

Table 9 shows the three different possibilities and their expected outcomes. The first example shows a row that has been updated. This is the case as the CRR table already includes the row with a CL of one, meaning it has previously been inserted. As the CL value in the Journal is set to one means the row has just been updated. If it had been re-inserted the CL value would have a minimum value of three in the Journal. Two from the delete and one from the insert.

When these two CL values are combined the expected value is there for one. In the second example the row has been deleted. When combing the two CL values the expected value is then four. The last example shows a row that has been inserted after it was previously deleted. Combining the two CL values should therefore yield a CL value of three. Using these three rules Equation 5 is created to determine the correct Cl value.

$$CL = \begin{cases} CL_{Journal} + CL_{CRR}, & CL_{CRR} \,\%\, 2 \;= 0 \\ CL_{Journal} + CL_{CRR} - 1, & CL_{CRR} \,\%\, 2 \;\neq 0 \end{cases}$$

Equation 5: Calculating the correct CL value.

5 Experimentation and Results

5.1 System Hardware Specification and Setup

To get a more accurate result during testing two different setups were used, a desktop and a laptop. The desktop hardware specifications are listed in Table 10, and Table 11 lists the hardware specifications for the laptop. **All numbers presented comes the setup listed in Table 11.**

Component	Name
Operating System (OS)	Windows 10 Pro 64-bit
Central Processing Unit (CPU)	AMD Ryzen 9 3900X @ 3.80GHz; 12 cores; 24 threads
Graphics Processing Unit (GPU)	NVIDIA GeForce RTX 3080 Desktop GPU
Random-Access Memory (RAM)	16GiB (2x8GiB) DIMM DDR4 3200Mhz
Storage	1 TB M.2 NVMe SSD; 3500/3300MB/s read/write

Table 10: Desktop Hardware Specifications

Component	Name
Operating System (OS)	Windows 11 Home 64-bit
Central Processing Unit (CPU)	AMD Ryzen 5 5600H @ 3.30GHz; 6 cores; 12 threads
Graphics Processing Unit (GPU)	NVIDIA GeForce RTX 3060 Mobile GPU
Random-Access Memory (RAM)	16GiB (2x8GiB) DIMM DDR4 3200Mhz
Storage	512 GB M.2 NVMe SSD; 3500/2900MB/s read/write

Table 11: Laptop Hardware Specifications

5.2 Measuring the cost SynQLite's triggers

Before any solution could be implemented it was important to know the cost of different triggers. Measuring the cost gives a clearer picture of the areas that is in the most need of improvements. SynQLite's triggers can be categorized into three classes: (1) The Clock trigger, (2) CRR triggers, and (3) History triggers. Each category was carefully measured, and the results are a combination of multiple runs. All benchmarks were performed on both systems for a more accurate result which was used during decision making. Before the cost could be measured a baseline was created. The baseline is a benchmark of an unmodified version of SynQLite. To measure the cost of a trigger, the trigger was removed, and a new benchmark was run.

The Clock trigger is a two-part process, as mentioned in section 3.2.3. To measure the true cost both parts are required. Measuring only one part does not give the full picture needed. To disable the Clock, trigger the millisecond update in the insert, update, and delete triggers were removed. The same process was applied to the CRR- and History triggers. The problem with the History trigger is that it is dependent of the CRR trigger to activate. The relationship between both triggers was shown in Figure 1. Therefor it is only possible to measure the cost of the CRR trigger and the full CRR/History trigger pair. Extra modifications could have been done to make the History trigger to activate of any operation but in the end decided against due to the same reasons mentioned above with the Clock trigger.

	Insert	Update	Delete
Baseline	14828.371	3898.895	8895.323
Only CRR/History trigger	15686.034	6812.77	9017.808
Only CRR trigger	76746.829	13667.629	17643.143
Only Clock trigger	104758.159	30594.504	34661.633
SQLite	238635.674	41097.095	45296.932

Table 12: TPS with different triggers deactivated.

Table 12 shows the TPS for runs with different triggers deactivated. Baseline is the result of the run using an unmodified version of SynQLite. The next row is the results of disabling the Clock trigger and only keeping the full CRR-/History triggers. As mentioned above it is not

possible, or reasonable, to only test the History trigger. The third row shows the results from a run only including the CRR trigger. Disabling the CRR trigger leaves only the Clock table left. The second to last row shows the result from that run. To get an idea of optimal performance a benchmark with all triggers deactivated was run. Comparing the baseline to a run with all triggers deactivated it is clear that all triggers combine severely limits the achievable TPS.

Table 13 shows the relative performance of each run to the baseline. Disabling all triggers increases the insert TPS over 16 times. Update and delete sees less of benefit with an ~10.5 and ~5 times increase when all triggers are disabled.

	Insert	Update	Delete
Baseline	1.0	1.0	1.0
Only CRR/History trigger	1.058	1.747	1.014
Only CRR trigger	5.176	3.506	1.983
Only Clock trigger	7.065	7.847	3.897
No triggers	16.093	10.541	5.092

Table 13: Relative TPS to baseline with different triggers deactivated.

Deactivating the Clock trigger gave an average of ~1.27 times increase to TPS across all three tests. It should be noted that, although overall above average, during some runs the average TPS of update and delete dipped below the baseline. From these numbers it can be concluded that the impact of the Clock trigger, although a sub-optimal implementation as seen in section 4.4, is negligible compared to the CRR-/History triggers.

A more noticeable boost to performance can be seen with the History trigger deactivated. This gave an average increase to TPS of ~3.55 times. The biggest performance increase happens when only the Clock trigger remains. On average the total TPS is increased ~6.27 times. With all tests insert, having the highest TPS overall, sees the largest benefit of deactivating triggers. Even though only the Clock trigger, a relatively low impact trigger, remains the difference in performance compared to using no triggers is massive. Insert sees a TPS increase of ~135 000, while update and delete each see an increase of ~10 500.

These results show that the current setup of triggers is inefficient, and there is a lot of room for improvement. Even combining triggers will give some improvement as seen in section 4.4. Some conclusions to take away from these are:

1. Triggers, even low-impact ones, will have an impact on performance. They should therefore be as lightweight as possible, and a trigger should not activate another trigger.
2. If a single trigger can perform the same job as two existing triggers, the two triggers should be merged into one.
3. The use of triggers should be avoided whenever possible. Especially for transactions which affect thousands of rows. Reasons for this was mentioned in section 3.1.1.

5.3 Comparing the new approach to the current

5.3.1 Custom Benchmark

Four benchmarks were run with the common parameters being listed in Table 14. This made sure all benchmarks were as equal as possible except for some parts as discussed in section 6.2. The first benchmark measured the performance of SynQLite to get a lower bound. To benchmark SynQLite the argument *version* was set to *old*. To get an upper bound of maximum achievable performance a second benchmark was ran using the argument *disable_triggers*. The benchmark runs as normal except the SynQLite associated tables do not get updated. Measuring the performance of the final solution was done by setting the argument *version* to *new*. To test the impact of the JSU the last benchmark uses it in the worst case possible. Instead of waiting for a clone, push, or pull request it continuously synchronizes after each transaction. It should be noted that the benchmarks using the new version do not have synchronization enabled as indicated by the *no sync* label in each graph.

Argument	Value
Iterations	3
Journal modes	DELETE, WAL
Minimum tables	5

Maximum tables	5
Minimum columns	5
Maximum columns	5
Minimum rows	10000
Maximum rows	50000
Row increment	10000

Table 14: Parameters used during benchmarking.

5.3.1.1 Insert Performance

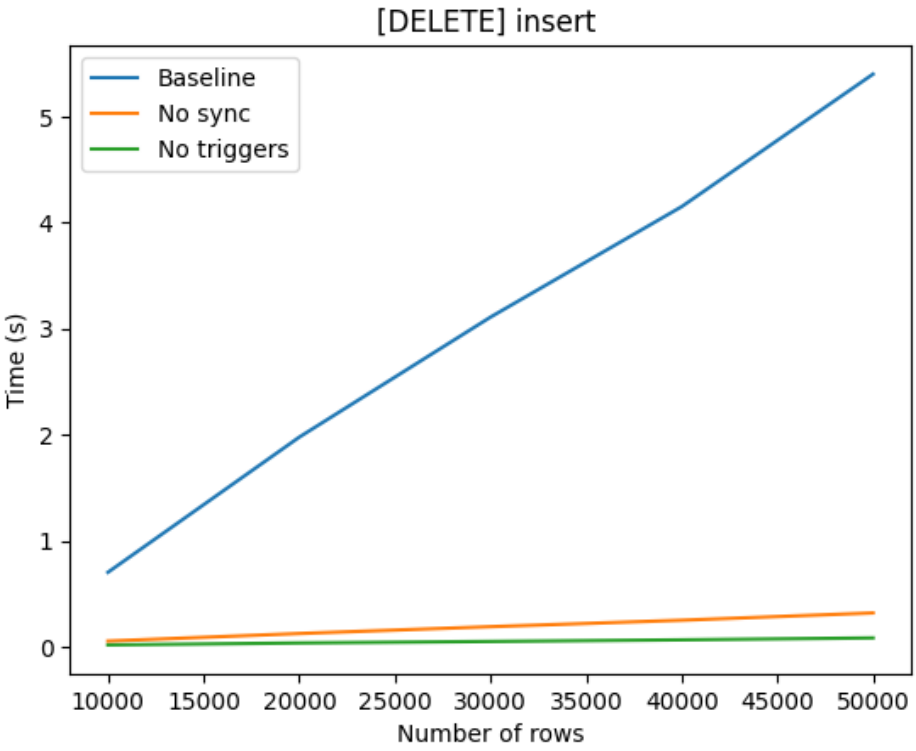

Figure 13: Insert performance comparison using journal mode DELETE.

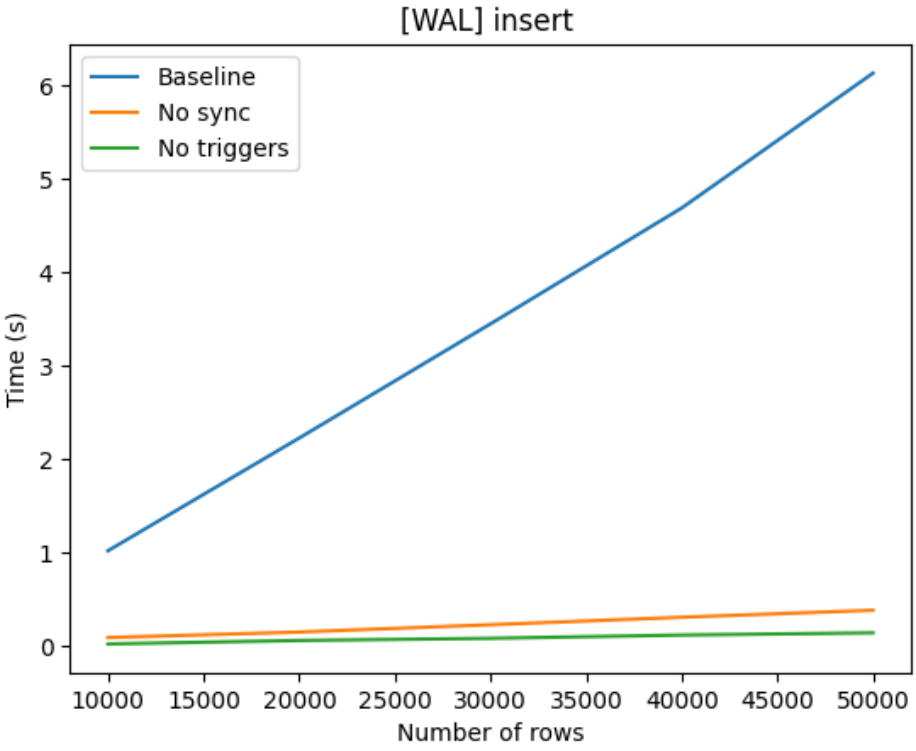

Figure 14: Insert performance comparison using journal mode WAL.

Figure 13 and Figure 14 shows that the new solution is a massive improvement over the existing method. The new method scales almost as well as SQLite while the previous method sees a drastic time increase as the number of rows increase. On average the previous method increases by ~0.469 seconds per 10 000 rows, and the new method ~0.0292 seconds. Still, the new method falls behind SQLite on average as seen in Table 15. The average TPS over all runs sees the new method being ~3.553 times slower using the DELETE journal mode, and ~15.97 times faster than the previous method. An improvement over the ~56.743 times difference between SynQLite and SQLite.

Journal Mode	Baseline	No sync	No triggers
DELETE	9777.2	156145.518	554793.31
WAL	8556.376	127091.246	347778.904

Table 15: Insert TPS for DELETE and WAL.

5.3.1.2 Update Performance

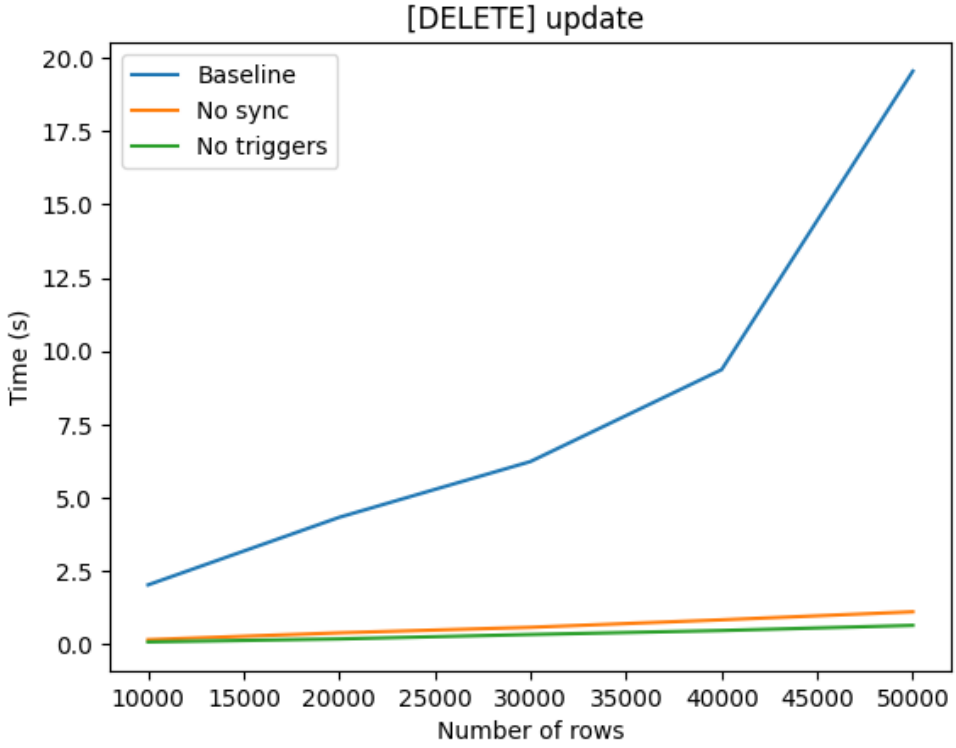

Figure 15: Update performance comparison using journal mode DELETE.

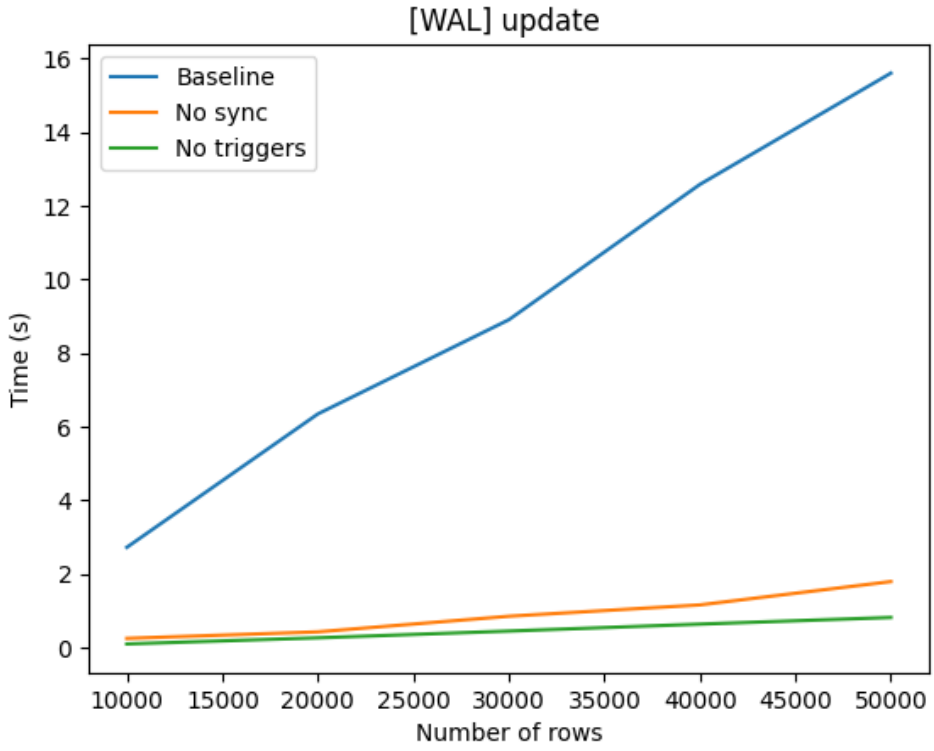

Figure 16: Update performance comparison using journal mode WAL.

As with insert, update scales with the same degree as insert. Matching the TPS closely in the beginning and starting to drift away around 35 000 rows. Using journal mode DELETE sees the best scaling for all methods for the current setup used during benchmarking. With an average increase of ~1.752 seconds for the previous method, ~0.154 seconds for the new method, and ~0.0564 seconds for SQLite per 10 000 rows. Looking at Table 16 shows a TPS difference of ~24.444 times between SynQLite and SQLite. With the new method this difference is lowered to ~1.8, an improvement of ~13.575 times over SynQLite.

Journal Mode	Baseline	No sync	No triggers
DELETE	3615.517	49081.882	88381.306
WAL	3249.911	34134.442	65812.151

Table 16: Update TPS for DELETE and WAL.

5.3.1.3 Delete Performance

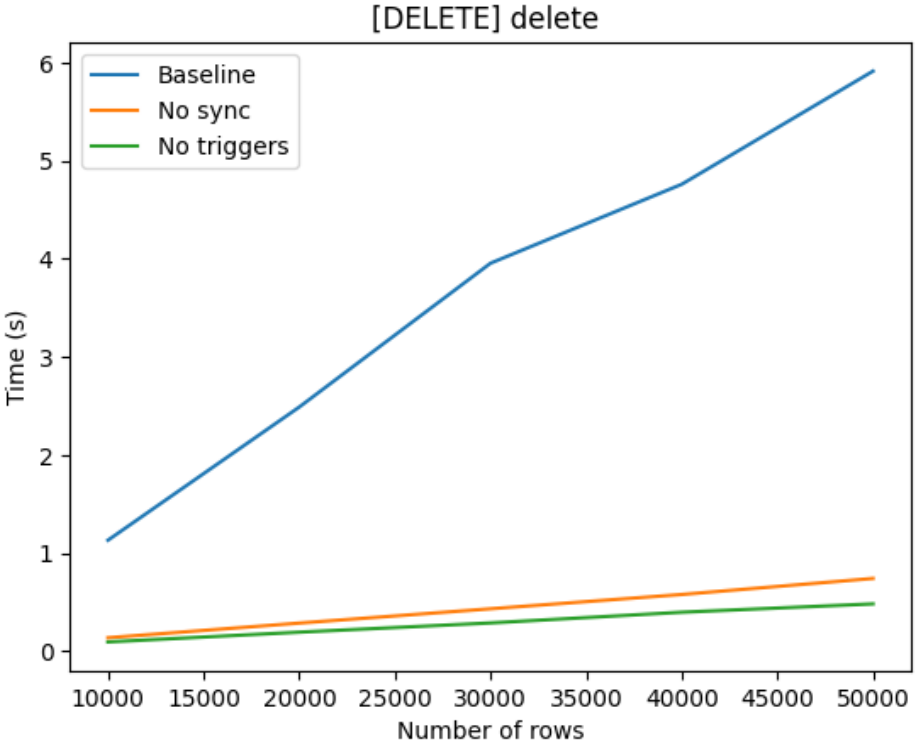

Figure 17: Delete performance comparison using journal mode DELETE.

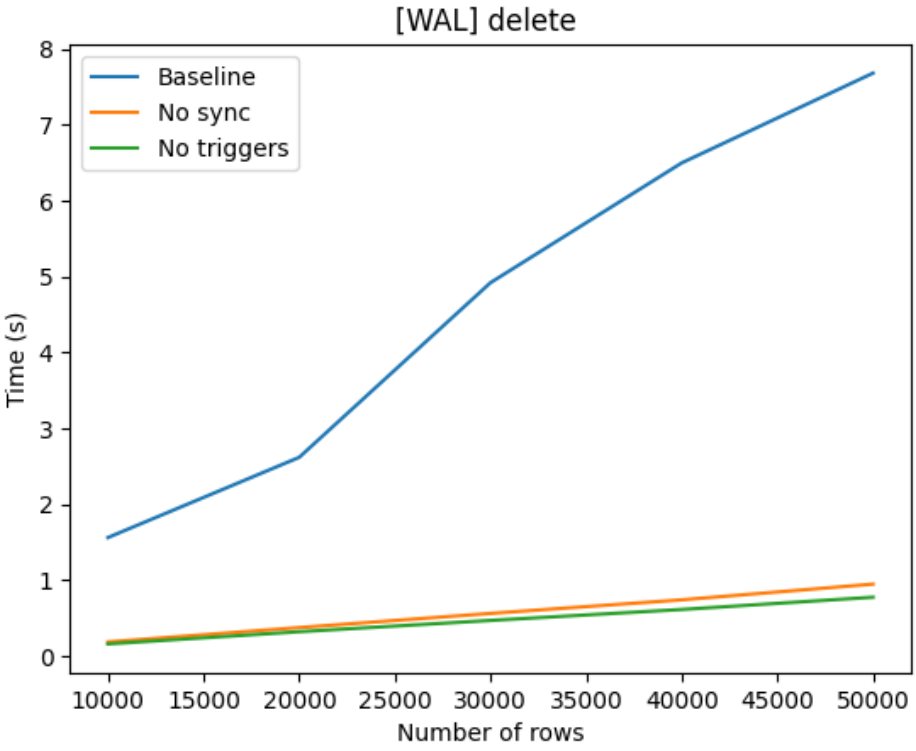

Figure 18: Delete performance comparison using journal mode WAL.

The new version of delete sees the biggest relative improvement to performance compared to insert and update. From looking at Figure 17 and Figure 18 it seems the scaling is better than with update, although worse than with insert. With an average increase of ~0.47 seconds for SynQLite, ~0.076 seconds using the new method, and ~0.038 seconds with SQLite per 10 000 rows. Looking at Table 17 it shows that the scaling is better than for both insert and update. Using the new method insert was ~3.553 times slower and update ~1.8 times slower. The new delete is about ~1.501 times slower than SQLite. Giving it an improvement of ~8.384 times over SynQLite which was ~12.589 times slower than SQLite.

Journal Mode	Baseline	No sync	No triggers
DELETE	8217.511	68902.149	103457.016
WAL	6444.405	52750.076	64572.802

Table 17: Delete TPS for all DELETE and WAL.

5.3.1.4 Synchronization Performance

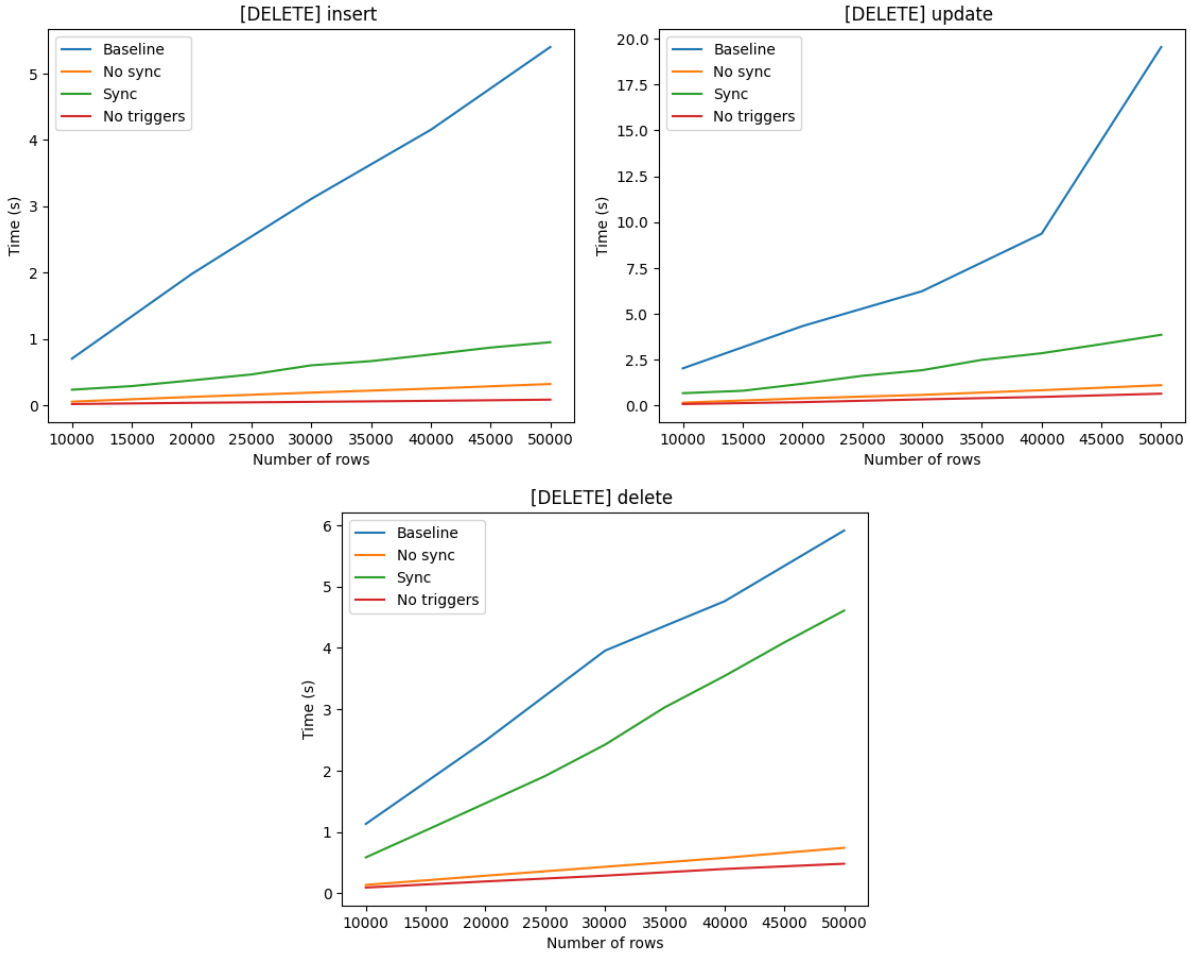

Figure 19: Performance difference for insert, update, and delete with sync enabled.

Running each benchmark again with synchronization enabled shows the impact it has on performance. Ideally, synchronization should only be executed per SynQLite request and not per transaction as discussed in section 5.3.1.5. Looking at the figure above, and table below, it shows all three operations takes a noticeable hit to their TPS. Insert sees a ~3.04 decrease to performance, and update and delete respectively sees a ~3.404 and ~5.8 decrease. While noticeable in comparison to the original performance, using active synchronization still sees an average improvement of ~3.606 times compared to SynQLite.

With inserts originally high TPS speed it is no surprise with the fall of about 100 000 TPS as the slightest increase in time will drastically change the results. Comparing the times of inserting 100 000 rows, using no synchronization used ~0.384 seconds and with synchronization it was increased to ~0.951 seconds. That is a difference of ~0.567 seconds

between using no synchronization and active synchronization. Update is the least affected by the three operations with a loss of ~34 600 TPS and delete is the most affected with almost a six times drop in TPS, losing almost ~57 000 its original TPS of about 69 115.

	Insert	Update	Delete
SynQLite	9777.2	3615.517	8217.511
Synchronization	51607.011	14422.126	11900.94
No synchronization	157020.364	49093.384	69115.717
SQLite	554793.31	88381.306	103457.016

Table 18: TPS comparison with and without synchronization enabled with journal mode DELETE.

5.3.1.5 Continuous Synchronization

In the example seen in Figure 19 for each transaction the database is wiped. Meaning no rows are left remaining between each iteration. For all benchmarks this is an optimal situation as there are less rows to filter through. In a realistic scenario the database will not be empty between transactions. Therefore, it is important to benchmark the performance of synchronization as the database grows, or already contains rows. This will give an idea of how well the implementation scales as the database grows. Figure 20 shows the time it takes to synchronize the Journal at different transaction volumes. By transaction volumes it is meant the number of rows affected between each synchronization. Looking at Figure 19 it shows the time for a transaction to commit for three different transaction volumes: per 100 transactions, per 1000 transactions, and per 5000 transactions. Looking at per 100 transaction it sees an exponential scaling to the time it takes for the transaction to complete. When increasing the volume to per 1000 or per 5000 rows affected the performance sees much better results for all three benchmarks. These results show the trade-off between prioritizing throughput or responsiveness. Synchronizing more often increases the total overhead, which results in less throughput. However, if the number of rows affected is large enough between each synchronization, or if database sees downtime, the performance impact should not be noticeable.

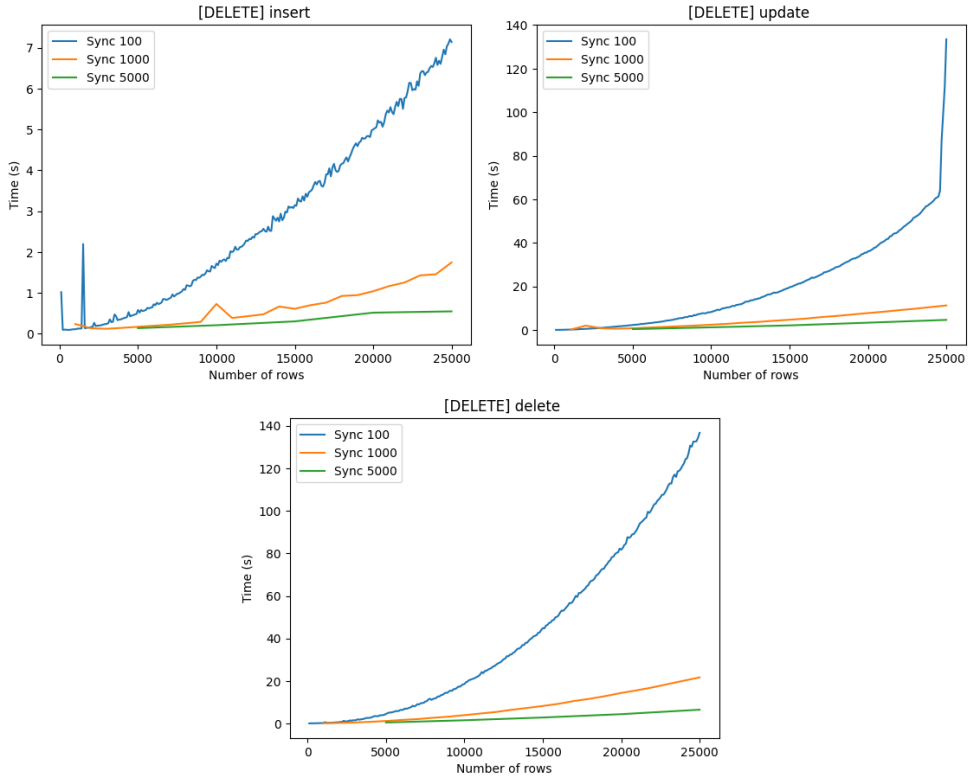

Figure 20: Comparison between synchronizing at different transaction volumes

Just comparing the solution against itself does not give a clear picture of how it compares to the previous method or a barebone SQLite database. Looking at Figure 21 and Figure 22 shows how the different versions scales against each other. As the database is not wiped per transaction each query has more rows to filter through. SynQLite having the most queries per operation shows a significant increase in time to complete a transaction. SynQLite was only benchmarked up to 5 000 rows as the time needed to benchmark to 25 000 rows would have been too long and would not have shown anything that is not already know. As with the new method, SynQLite also sees a speedup when increasing the transaction volume but not to the same degree as the new method. Using no synchronization is the preferred alternative as it gives the best performance.

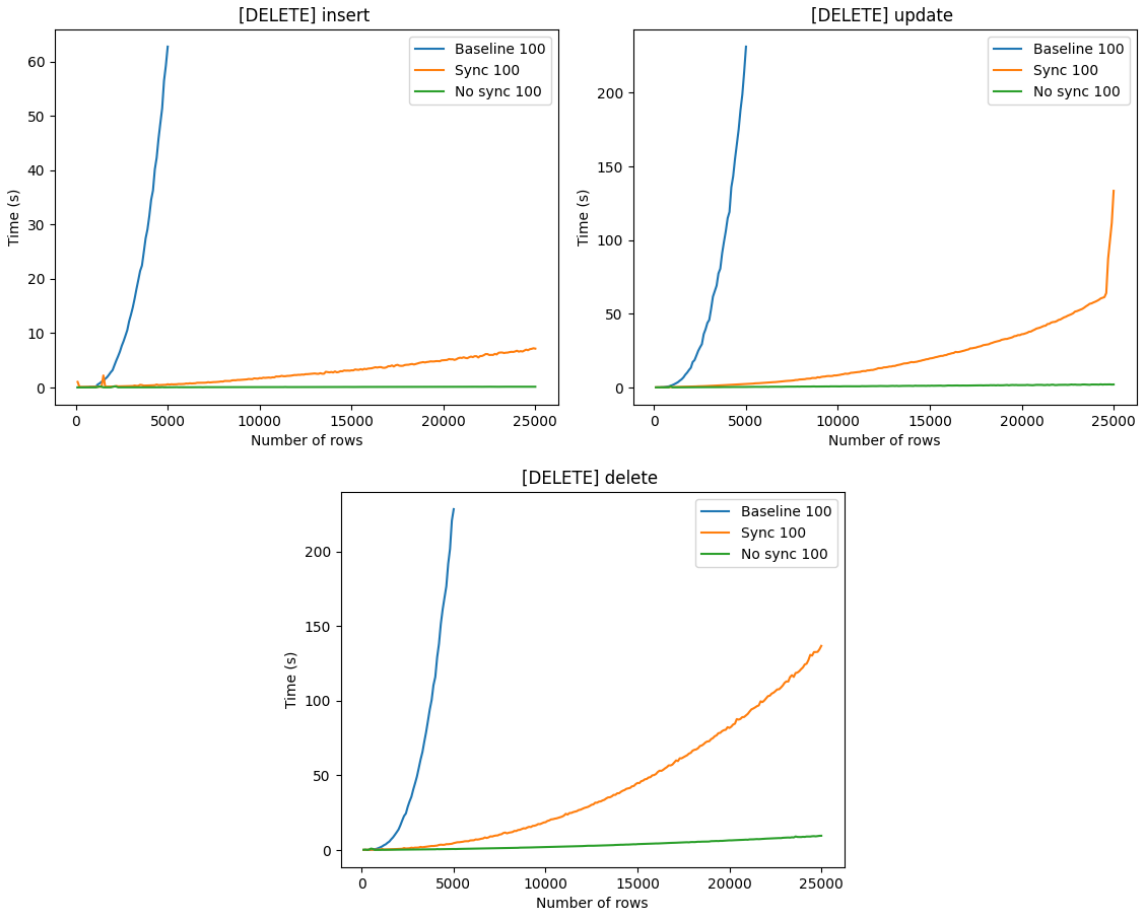

Figure 21: Performance comparison per 100 transactions

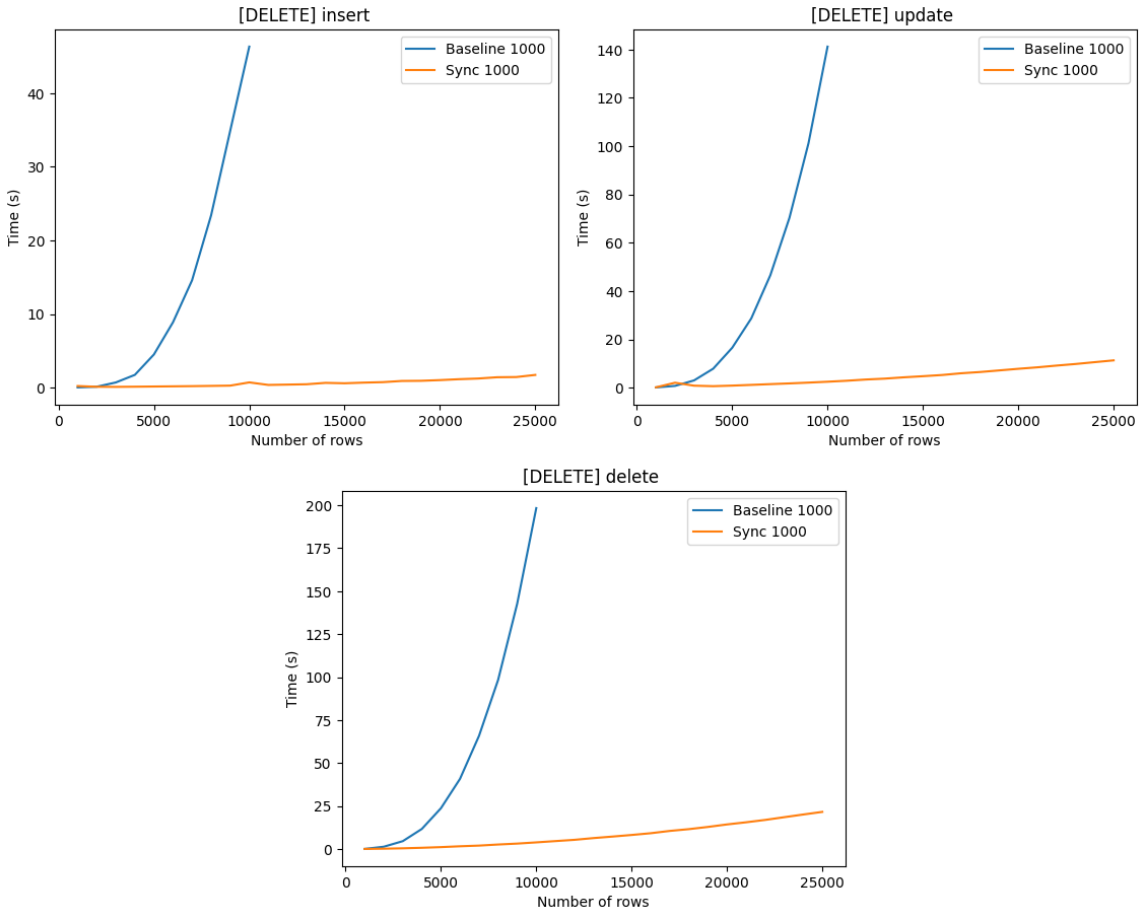

Figure 22: Performance comparison per 1000 transactions

5.3.2 TPC-C

This section covers the results from the py-tpcc python program. Instead of individually benchmarking each operation TPC-C runs a more realistic simulation. Each benchmark has their own advantages and disadvantages as discussed in section 6.3. While TPC-C is mainly meant to benchmark the TPS of a database system, two extra parameters were added: (1) Time Complexity. Representing the time it takes to load all data into the database before TPC-C benchmark can execute. (2) Space Complexity. Representing the space required to store all data loaded into the database. It should be noted that both for time- and space complexity the values can vary between each run as the other background tasks can be performed by the computer, or the data generated takes more, or less, space. Although they should not vary widely from the displayed values.

5.3.2.1 Time Complexity

From the results displayed in Table 19 it shows a massive improvement to the load time. Compared to SynQLite, the new solution is almost 300 times faster, only making it ~1.38 times slower than SQLite. For a more visual representation between SynQLite and the new version is 15 hours, 46 minutes, and 40 seconds. Instead of waiting over half a day for data to be loaded, with the new version it is only less than a minute of extra load time.

Version	Time (s)	Time expanded
SQLite	137	2m 17s
SynQLite (Original)	56990	15h 49m 50s
SynQLite (Improved)	190	3m 10s

Table 19: Time spent loading items into database.

Version	Relative difference (x)
SQLite	1
SynQLite (Original)	522.84
SynQLite (Improved)	1.38

Table 20: Relative time difference for time spent loading items into database.

5.3.2.2 Space Complexity

Comparing the total size of each version it is clear that the new method is both more time- and space efficient. With SynQLite almost being 5 times larger in size, the new version is only ~1.77 times larger than SQLite. SynQLite is meant to be a local-first database meaning the database is stored on the device. Having the database file take up a lot of space might not always be possible due to storage limitations. It is therefore important to not only make a database time efficient, but also space efficient. Unfortunately, once the database synchronizes the journal the size will increase. Due to both the original and new solution requiring a primary

key to be specified per table it was not possible to synchronize the database. A previous implementation, which was similar to the current, saw the file size increase to ~3.2 times of the original SQLite file. The difference being the inclusion of primary keys in the Journal. The current solution will result in a larger file after synchronization but should still be less than SynQLite. The decrease in size can be explained by the fact that the new version is more space efficient with locally deleted items. If an item was never synchronized and is deleted, it does not end up in the CRR- and History tables.

Version	Space (bytes)
SQLite	372 842 496
SynQLite (Original)	1 818 980 352
SynQLite (Improved)	662 822 912

Table 21: Size difference of fully loaded database.

Version	Relative difference (x)
SQLite	1
SynQLite (Original)	4.886
SynQLite (Improved)	1.778

Table 22: Relative size difference of fully loaded database.

5.3.2.3 Transactions per Second

Running the TPC-C benchmark proved difficult with SynQLite due to some transaction queries not having the opportunity to execute. The default execution time of 60 seconds was not enough to execute all possible queries. Some runs therefor lacked results. Fixing the issue was done by increasing the execution time ten times, up to 600 seconds. Increasing the execution time, while taking longer to benchmark, also gives a more accurate result due to the law of large numbers

(LLN). LLN states that the if the same experiment is ran a large number of times the average value should be close to the expected value [17].

When summarizing the time spent per transaction query category the total time will rarely equal exactly 600 seconds. The reason for this is how py-tpcc checks if enough time has passed. Between each check a new transaction is executed. If the execution happens is finished right before the 600 second mark, the time to generate a new time value can be enough to reach the mark. On the other side, if the execution of a new transaction happens right before the 600 it can surpass the time limit. Both scenarios are not seen as problems as from the results it has an average deviation of $\pm 0.23\%$.

As discussed in a previous project, both the transaction query categories *ORDER_STATUS* and *STOCK_LEVEL* only include read queries and is therefore not affected by the triggers [2]. It is for this reason that SynQLite sees an extremely high TPS for these two categories. It can therefor look like SynQLite has a much higher read performance than SQLite. Comparing the number of transactions executed disproves this theory as SQLite manages the executed almost 3000 of these queries and SynQLite only 33. Giving SQLite a more realistic TPS due to LLN.

	Executed	Time (µs)	Rate (txn/s)
DELIVERY	2193	34241860.62812805	64.04
NEW_ORDER	23708	330426885.6048584	71.75
ORDER_STATUS	2062	809164.0472412109	2548.31
PAYMENT	22699	228183174.37171936	99.48
STOCK_LEVEL	2091	5195189.952850342	402.49
TOTAL	52753	598856274.6047974	88.09

Table 23: TPC-C Results for unmodified SQLite database.

	Executed	Time (µs)	Rate (txn/s)
DELIVERY	13	130698473.93035889	0.12
NEW_ORDER	183	463705282.2113037	0.39
ORDER_STATUS	17	2382.993698120117	7133.88
PAYMENT	199	7261548.042297363	27.40
STOCK_LEVEL	16	11210.203170776367	1427.27
TOTAL	428	601678897.3808289	0.71

Table 24: TPC-C Results for standard SynQLite database.

	Executed	Time (µs)	Rate (txn/s)
DELIVERY	1888	36788944.244384766	51.32
NEW_ORDER	20955	331954324.72229004	63.13
ORDER_STATUS	1836	767622.4708557129	2391.80
PAYMENT	20078	224778720.14045715	89.32
STOCK_LEVEL	1836	4672804.832458496	392.91
TOTAL	46593	598962416.4104462	77.79

Table 25: TPC-C Results for improved SynQLite database.

Running py-tpcc using the new version shows the results being more matched with SQLite. Although it falls behind in every category it is a clear improvement over SynQLite. As with SQLite, the new version can easily get a fairly accurate result with the default 60 seconds execution time. During the execution synchronization was turned off, and all data was written into the journals. If continuous synchronization was enabled a dip in performance would be expected.

5.3.2.4 Total- and Relative TPS

Looking at individual numbers is helpful for finding areas of weaknesses, but it is the total TPS which gives an indication of the overall performance. Comparing the total TPS of all three versions shows that the new version executes transactions ~109.563 times faster than SynQLite. With SynQLite having 0.8% of the total TPS SQLite achieves, the new version can reach up to 88% of the total TPS. The last 12% equals about 10 TPS, and while it might not seem like a lot, looking at Table 23 and Table 25 that is a difference of over 6 000 transactions in a span of 600 seconds.

Version	Rate (txn/s)
SQLite	88.09
SynQLite	0.71
SynQLite (Improved)	77.79

Table 26: Total TPS difference

Version	Relative difference (x)
SQLite	1
SynQLite	0.008
SynQLite (Improved)	0.883

Table 27: Relative total TPS difference

6 Discussion

6.1 Fluctuations during benchmarking

There are several steps that can be taken to reduce the differences between each run, e.g., closing background tasks, pausing updates, restarting the computer to close programs and clear memory, or not using the computer while the benchmark is ongoing. Even with all these steps each run will not be equal, and the same benchmark will give different results. These fluctuations mostly occur from the CPU being busy, e.g., allocating time to other background tasks. To combat these issues each benchmark should be ran multiple times. Ideally the number of iterations should be high as possible, as the more time the test is repeated a more realistic average can be calculated. When running the final benchmarks this number of iterations was set to three. Three was selected as it resulted in a good average while keeping the runtime at an acceptable level. The problem with increasing the number of iterations, even by one, is the number of runs increases drastically. In Table 14 the parameters test two journal modes, with one table iteration, one column iteration, and five row iterations, each three times. Summarizing all runs together equals 30 runs for one benchmark. Increasing the number of iterations by one equal 10 more runs as each increase in iterations is multiplicative. It should also be noted that the laptop used to benchmark the different solutions has difference performance modes. Looking at Table 12 and Table 18 as two examples show a clear difference in TPS for SQLite even though they were run on the same machine. The reason for this comes from the fact that the first batch of benchmarking the laptop was ran for multiple days and was set to a power efficient performance mode. The benchmarks in 5 were using a high-performance mode. The different benchmarks in the same tables or sections were all ran with the same mode and are still comparable.

6.2 Differences in benchmarks

One of the most important aspects of benchmarking is making sure each run is as similar as possible. Even if the systems that is being tested are vastly different. As if the results are not from the same benchmark, they cannot be fairly compared against each other. However, it is not always possible to run equal benchmarks as complications can, and will, arise. These can

range from wanted or unwanted randomization, differences with systems that require slight modifications, or reasons discussed in section 6.1.

During benchmarking these complications was present. The only control given the to the user is the parameters listed is what will run, and general parameters, e.g., number of tables, column, and rows. What is not possible to control is the data, or type of data, being generated as it is random. It is possible to set a seed such that the same tables and data are generated for each run. The benefit of randomization is that testing many different combinations of tables and columns will give a more realistic representation of the performance. What cannot be controlled, even with seeds, is the rows that are deleted or updated as it uses SQLite's built in random function in queries. These cannot be set with a seed and therefor will be the only variation between each run. In the previous version of the benchmark these rows were selected used Pythons random library. This method was both inefficient and complicated and therefor replaced with the new method.

With four different versions being tested all arguments could not be equal. Testing the original SynQLite required the version to be set to *old*. This disables some arguments, e.g., disabling of triggers and synchronization. This means to test an implementation with all triggers disabled the version had to be set to *new*. Using the new version with triggers disabled is the same as running the old version with triggers removed from the source code. It just makes benchmarking easier. Testing the new version, with and without synchronization, was done by using the argument *do sync*. The argument is only available with the new version.

6.3 Specific vs General benchmarking

When testing different solutions, it is both necessary to benchmark specific aspects, but also the solution as a whole. If one type of benchmark is used, but the other is left out, the decision making can be compromised and the final solution sub-optimal. Running specific benchmarking, e.g., testing insert, update, and delete performance individually, made it possible to find which one was slowing down the performance. Using the information received by these results the process of improving easier. Allowing for quickly testing different solutions, or slight changes, without waiting for an entire run to finish. The downside of this type of benchmarking is that it does not give a realistic representation of what the usage looks

like. In a realistic scenario both reads and writes will be run, and writes are not restricted to one type of operation but a mix of all. Using a benchmarking tool like py-tpcc allowed for testing the entire solution and see how the performance fares in comparison. These are the reasons why the custom benchmarking tool was built the way it is. There was no reason to build a tool which performs a similar simulation as it already existed. Using py-tpcc by Andy Pavlo allowed for spending more time developing a custom benchmarking tool and experimenting with different solutions.

6.4 Is the CRR- and History tables necessary?

After implementing the Journal, a table which contains the minimum amount of information required to generate the CRR rows, a question arose. Is there a need for the CRR- and History tables? If the rows can be generated on the fly, there should not be a reason for them to exists. This theory is reinforced by the data in both Table 19 and Table 21. Keeping all data in the journal and only generating on a request will take some time but if it is a background task it will mostly not be noticeable. Also, the size difference between only keeping the data in the journal in comparison to the CRR- and History table is almost three times the difference.

With these facts in mind, it is strange that the answer depends on the scenario. If a system is built with plenty of storage an extra ~1.2GiB will not make any difference. Most storage devices used these days support up to multiple Terra bytes. On the other side a system with limited storage capabilities might afford to wait the extra time needed by the JSU. Therefor the answer to the question is sometimes, but the option to use either method could be an interesting topic for the future.

6.5 Why only run benchmarks using DELETE and WAL

All data represented in section 5 was either from using the journal modes DELETE or WAL. It was originally planned to display the results for all different journal modes as each have their use case. Originally all journal modes were benchmarked before the final benchmarks were ran and the decision to only test DELETE and WAL was made. There are two reasons for this choice. (1) While depending on the scenario the different journal modes might see better

performance than the others both DELETE and WAL are most likely going to be the most used ones. As mentioned in section 3.1.2, DELETE is the default journal mode, and as with WAL, does not need to be set each time a connection to the database is established. (2) The results from benchmarking all journal modes are predictable as seen from previous testing. On the systems used for benchmarking TRUNCATE and PERSIST saw either the same performance, or a slightly higher, than DELETE. Using MEMORY or OFF saw the highest performance but neither will likely be used due to the risk associated with them. Leaving DELETE and WAL as the two contenders.

6.6 Should continuous synchronization be used?

In this thesis the concept of continuous synchronization was mentioned a few times. With the current implementation there are two main methods of running synchronization. (1) Perform synchronization regularly. How often can either be tied to a certain amount of time since last synchronization or after a set number of transactions has passed. (2) Synchronize the journal after each transaction. Both options have benefits and drawbacks. Using continuous synchronization will still be a faster alternative than the current method as updating the journal is a lot faster than the CRR- and History tables. In addition, running the JSU after a transaction creates one big transaction instead of activating multiple triggers to update the tables. Having continuous synchronization activated also means the tables are always updated, and results in no wait time on a request. The downside is the impact to performance as seen in Table 18. With an average decrease of almost 75% in TPS across all three operations for high performing systems it might not be an option. As advised in section 5.3.1.4, it is instead recommended to run synchronization periodically. Like the answer in section 6.4 it depends on the scenario, but it is highly recommended to use periodic updates over continuous synchronization.

6.7 Lessons learned

In conjunction with working on this thesis, as well as the previous project, a few lessons can be taken away. Firstly, there technically is no wrong usage of triggers, and they can be made as lightweight or complex as necessary. However, complex triggers, especially the ones that includes multiple layers of triggers, will severely impact the performance of the database.

When making triggers other alternatives should always be considered which would either result triggers no longer being necessary, or less complex triggers. Ideally at most one trigger per operation.

With this in mind, all stages of performance improvements, problem finding, experimentation, debugging, and testing, is made a lot easier with custom tools. Creating custom programs allows for tailoring to the project's needs. Instead of working on the entire system creating a testing environment which does the same as the part in question makes it easier to work with. With less code to worry about it is easier to get a better understanding of what functions do, and how that part works. While not always possible, as it depends on the complexity of the system, creating a custom benchmarking tool integrated with the test environment allows for convenient, fast, and specific benchmarking.

Two other lessons that go hand in hand is optimization based on measurements and quick prototyping. During the project a lot of different designs were implemented and tested. If each idea was fully implemented there would not have been enough time for all to be tried and tested. Instead, a compromise was made where the bare minimum of each solution was implemented. Either a small part of the solution or the bare minimum of the entire solution, if necessary, was implemented to test whether the idea was viable. If the solution saw any promise, it would be expanded upon to test further. As the goal of this thesis was to improve the performance of SynQLite decision on whether an idea was viable was based on measurements.

The last lesson to take away is that even though the presented solution is a massive improvement over the current method it is far from perfect. There are multiple ideas that could be implemented, some which are mentioned in section 8, to improve on the solution, or entirely different solutions could exist which offers better performance. The solution given is meant to be a proof of concept, and a good first step to making SynQLite better.

7 Conclusion

Trying to achieve optimal performance for any systems should always be a goal to strive for. Having to wait a few seconds for simple transactions, let alone days, can cause frustration among users, and potentially be a quit factor. After analyzing SynQLite in a previous project, it was found the ineffective use of triggers was the reason for its performance issues. Comparing the performance against an SQLite database with no triggers using the TPC-C benchmark SynQLite was found to be ~85 times slower in total TPS. A worse result was found when comparing the time needed load the database with the data needed to run the TPC-C benchmark. While SQLite used two minutes and seventeen seconds, SynQLite took almost 523 times longer using fifteen hours, forty-nine minutes, and fifty seconds [2]. The goal of this thesis was therefore to increase the write performance of SynQLite to match the results from SQLite as closely as possible.

Getting to the current solution required many attempts and a lot of experimentation. Many ideas were thought of at the beginning, some were scrapped before they were tried as they were either to similar or not possible to implement due to restrictions. The first idea that was tried was to see if the current solution could be salvaged. To get an idea of the situation each trigger was carefully measured to check their impact on the performance. After all measurements were taken it was deemed that there was little that could be done. Small changes were tested but resulted in little to no improvement. Using what was learned while experimenting with the current solution other solutions came up. The second idea was to combine, or remove, tables to reduce the number of triggers and table lookups needed. Some operations in SynQLite uses the Clock table to fetch the time of last update, but this could be circumvented by fetching the time from other tables which also stored the time. Designs with no Clock table was tested, but as with the previous idea it did not give the desired result.

Going back to the drawing board, using all that was information available, a completely new solution emerged. A middleman which stores the minimum amount of information needed. The new solution provided both the desired performance improvement and kept the logic of SynQLite intact such that nothing had to be changed. As mentioned in the section 6.7 there are still plenty of improvements that could be made. While massively improving on the write performance the given solution is only meant to be a proof of concept. Therefor it was not

implemented into SynQLite's source code. However, all components are made with the intent that if the solution were to be used it can be inserted without much difficulty.

As the current performance of SynQLite is not acceptable for any database system a recommendation would be to implement the solution as is. Then continue the development of the project until the desired performance is achieved. The new solution does not require any changes to SynQLite's functions, i.e., clone, push, pull, or sync, as it was made to be unnoticeable in any way except performance. Using the new solution increases SynQLite's performance to a point where it almost matches SQLite. It is not possible to get the same performance due to the necessity of triggers, and if it were to be further improved it might require a brand-new method.

Looking at what has been achieved throughout the thesis I can confidently say that both the main and secondary goals has been met. A lot of thought and work has gone into the project as the desire was to make SynQLite a viable database system. There is still a lot of work to be done and given more time with the project I am sure both the write and synchronization performance could be improved, especially the synchronization. To conclude, I am satisfied with the work done, and hope the knowledge from this thesis helps SynQLite as well as other similar database systems.

8 Future Work

8.1 Implementing the Journal and JSU

An immediate next step would be to implement the solution into SynQLite. While the components are made there are still a few tasks left. After the tables, triggers, and JSU has been updated three features needs to be added. The ability to enable and disable continuous synchronization. The problem with continuous synchronization is that it cannot easily be implemented. SQLite does not offer any support for running queries after a transaction is successful. It is therefore up to the user to execute the synchronization after a transaction.

A potential solution, and another task, would be to extend the API to support journal synchronization. The new API call should be easily accessible and would only require the name of the database. It is then up to SynQLite to correctly synchronize the journal. In addition to a new API call the JSU needs to be incorporated into the existing features: clone, push, pull, and sync. The only change that needs to be made to each function would be to call the JSU before the data is received or sent.

8.2 Upgrading the JSU

Depending on the amount of data in the Journal before synchronization the time it takes to synchronize will vary. A benefit with the JSU is that the performance of the database system is better the longer it takes between each synchronization. Most of the time is spent looping over the rows to generate values for the CRR- and History queries. In addition to improving the looping, SQLite supports multiple instances to read from the database at once [18]. The JSU executes three, potentially large, read queries, and being able to perform all three at the same time will result in a decrease in time spent synchronizing.

The performance of the JSU can be upgraded by utilizing multithreading/-processing. By default, Python comes with multithreading libraries included, such as *threading* and *multiprocessing*. Both libraries offer good tools for multithreading functions and loops to improve performance. Threading is seen as a more high-level threading library while multiprocessing is more low-level allowing for more customizability. Other non-standard multiprocessing libraries such as *MPI for Python* are excellent options and can offer more flexibility and better performance than previously mentioned ones. A recommendation would

be to implement and test the performance of all libraries and select the best one. Although, in the end, it is up to the developer to select the library they are most comfortable with.

8.3 Re-implementing previous benchmarks

When the custom benchmarking tool was refactored for this thesis three previous benchmarks were left out. These were the *time to clone*, *-push*, and *-pull* benchmarks. They were not seen as important for the project as the performance impact of the Journal could be measured through other methods. Bringing back these benchmarks should be done if the project is to be taken further. Having the benchmarks will both give a more accurate representation of the increased time needed to perform these actions and help debug the solution in case something goes wrong. The benchmarking tool is after all meant as a debugging and benchmarking tool. During testing different test scripts were written utilizing the benchmarking modules to debug the solutions.

9 References

[1] I. T. Tomter and W. Yu, "Augmenting SQLite for Local-First Software," in *New Trends in Database and Information Systems*, Cham, L. Bellatreche *et al.*, Eds., 2021// 2021: Springer International Publishing, pp. 247-257.

[2] S. Brynjulfsen, "Benchmarking a Conflict-Free Replicated Relational Database System," 2022.

[3] A. Pavlo. "Python TPC-C." https://github.com/apavlo/py-tpcc (accessed 2023).

[4] P. Taylor. "Amount of data created, consumed, and stored 2010-2020, with forecasts to 2025." https://www.statista.com/statistics/871513/worldwide-data-created/ (accessed 2023).

[5] "About SQLite." SQLite. https://www.sqlite.org/about.html (accessed 2023).

[6] "Features of SQLite." SQLite. https://sqlite.org/features.html (accessed 2023).

[7] "CREATE TRIGGER." SQLite. https://www.sqlite.org/lang_createtrigger.html (accessed 2023).

[8] "Types of triggers (PL/SQL)." IBM Corporation. https://www.ibm.com/docs/en/db2/11.5?topic=plsql-types-triggers (accessed 2023).

[9] "Temporary Files Used By SQLite." SQLite. https://www.sqlite.org/tempfiles.html (accessed 2023).

[10] "PRAGMA Statements." SQLite. https://www.sqlite.org/pragma.html (accessed 2023).

[11] "PRAGMA Statements - Journal Mode." SQLite. https://www.sqlite.org/pragma.html#pragma_journal_mode (accessed 2023).

[12] "Write-Ahead Logging." SQLite. https://www.sqlite.org/wal.html (accessed 2023).

[13] W. Yu and S. Rostad, "A low-cost set CRDT based on causal lengths," presented at the Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, Heraklion, Greece, 2020. [Online]. Available: https://doi.org/10.1145/3380787.3393678.

[14] "Overview of the TPC-C Benchmark." TPC. https://www.tpc.org/tpcc/detail5.asp (accessed 2023).

[15] S. Brynjulfsen, "Benchmarking a Conflict-Free Replicated Relational Database System," UiT - The Arctic University of Norway, 2022.

[16] "Limits In SQLite." SQlite. https://www.sqlite.org/limits.html (accessed 2023).

[17] F. M. Dekking, *A Modern Introduction to Probability and Statistics: Understanding Why and How*. Springer, 2005.

[18] "Frequently Asked Questions." https://sqlite.org/faq.html#q5 (accessed 2023).