



UHASSELT

KNOWLEDGE IN ACTION



Maastricht University

Faculteit Wetenschappen **School voor Informatietechnologie**

master in de informatica

Masterthesis

Robustness for multiversion concurrency control in relational databases

Bram Droogmans

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Frank NEVEN

BEGELEIDER :

dr. Brecht VANDEVOORT

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.



UHASSELT

KNOWLEDGE IN ACTION

www.uhasselt.be

Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

2023
2024



Maastricht University

Faculteit Wetenschappen ***School voor Informatietechnologie***

master in de informatica

Masterthesis

Robustness for multiversion concurrency control in relational databases

Bram Droogmans

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Frank NEVEN

BEGELEIDER :

dr. Brecht VANDEVOORT

Preface

First, I would like to thank those who helped me with this thesis and its research. I am especially grateful to Prof. Dr. Frank Neven and Dr. Brecht Vandevoort, who guided me throughout the year, always ready to assist with any problems and to help determine the best next steps. I also want to thank my parents, friends, and girlfriend for their moral support and for believing in me.

The purpose of this work is to support further research and experimentation in the field. This is achieved by developing three tools that facilitate experimentation, allowing researchers to configure settings, run experiments, and analyze results using visualizations. Additionally, the robustness and allocation algorithm has been implemented and tested, and it is ready for use in future research.

Throughout this project, I learned how to manage a large project structurally, improved my time management skills, learned how to write a thesis, and gained a better understanding of my strengths and weaknesses.

Samenvatting

Introductie

De thesis behandelt een cruciale uitdaging in databasebeheersystemen: het balanceren van transactiedoorvoer met het minimaliseren van anomalieën. Met de steeds toenemende hoeveelheid en complexiteit van data is het belangrijker dan ooit dat databases zowel performant als betrouwbaar blijven. Dit werk richt zich op robuustheid in transactieallocaties, met name binnen het multi-version concurrency control (MVCC) framework. Het hoofddoel was het ontwikkelen en implementeren van een Python-gebaseerde tool die transactiedoorvoertests faciliteert en het implementeren van het robuustheids- en allocatie-algoritme voor gemengde isolatieniveaus.

Motivatie en doelen

De motivatie achter dit onderzoek komt voort uit de noodzaak om gelijktijdige transacties af te handelen op een manier die de prestaties niet opoffert om de dataintegriteit te behouden. Traditionele mechanismen behouden weliswaar de dataintegriteit, maar offeren hiervoor vaak de prestaties op. Deze scriptie heeft als doel de afweging tussen doorvoer en anomalieën te onderzoeken en strategieën te ontwikkelen om beide tegelijkertijd te optimaliseren. De specifieke doelstellingen zijn:

- Het ontwikkelen van een tool die transactiedoorvoertests faciliteert.
- Het implementeren van het robuustheids- en allocatie-algoritme voor gemengde isolatieniveaus.

Robustness en isolation levels

Databasebeheersystemen ondersteunen meerdere isolatieniveaus, elk met hun eigen afwegingen tussen doorvoer en minimalisering van anomalieën. De isolatieniveaus die in dit document worden besproken zijn Read Committed (RC), Snapshot Isolation (SI) en Serializable Snapshot Isolation (SSI).

- **Read Committed:** Zorgt ervoor dat de gelezen data op het moment van lezen is gecommitteerd. Dit niveau voorkomt dirty writes, maar biedt geen bescherming tegen andere anomalieën.
- **Snapshot Isolation:** Biedt een momentopname van de database aan het begin van de transactie, waardoor anomalieën zoals non-repeatable reads worden voorkomen. Echter, write skew blijft mogelijk.
- **Serializable Snapshot Isolation:** Het strengste niveau, dat volledige serialiseerbaarheid waarborgt door alle mogelijke anomalieën te voorkomen. Dit gaat echter wel ten koste van de prestaties.

Robuustheid in dit werk verwijst naar de eigenschap die ervoor zorgt dat elke mogelijke volgorde van transacties serialiseerbaar is onder een gegeven allocatie. Een robuuste allocatie garandeert dat de database consistent blijft, ongeacht de volgorde waarin transacties worden uitgevoerd.

Tool ontwikkeling

Een belangrijke bijdrage van de thesis is de ontwikkeling van drie Python-gebaseerde tools die ontworpen zijn om transactiedoorvoertests te faciliteren. De drie tools zijn:

- **Het maken van configuraties:** Maakt het gemakkelijk om configuratiebestanden voor verschillende experimentele opstellingen te creëren.
- **Uitvoering van experimenten:** Ondersteunt de uitvoering van doorvoerexperimenten door gelijktijdige transacties in een gecontroleerde omgeving te simuleren.
- **Visualisatie van de resultaten:** Biedt visualisaties om de resultaten van verschillende configuraties te analyseren en te vergelijken.

Deze tools zijn essentieel gebleken voor het systematisch vergelijken van de resultaten van verschillende configuraties.

Robustness algoritme

De implementatie van het robuustheids- en allocatie-algoritme is het tweede kernonderdeel van deze scriptie. Dit algoritme is ontworpen om ervoor te zorgen dat de optimale robuuste allocatie wordt gegeven voor elke set transacties. Het algoritme werkt door te beginnen bij de strengste allocatie en vervolgens het isolatieniveau één voor één voor elke transactie te verlagen. Deze wijziging wordt behouden of ongedaan gemaakt door te controleren of de nieuw verkregen allocatie robuust is. De implementatie is geverifieerd door gebruik te maken van bekende (optimale) robuuste allocaties. In totaal werden 23 allocaties getest. Het resultaat van al deze tests was correct.

Experimenten en resultaten

De uitgevoerde experimenten zijn gericht op het onderzoeken van de invloed van isolatieniveaus op prestaties en anomalieën. De experimenten toonden aan dat de optimale robuuste allocatie beter presteert dan de allocatie waarbij alle transacties aan SSI zijn toegewezen. Dit was echter niet altijd het geval. Soms vertoonden de experimenten onverwacht gedrag, waarbij een van de programma's veel vaker werd uitgevoerd dan de andere. De reden hiervoor is nog niet vastgesteld, en verder experimenteren is noodzakelijk.

Conclusie

Deze thesis richt zich op het cruciale belang van robuustheid in multiversion concurrency control. Door de doorvoer en het minimaliseren van anomalieën in balans te brengen via intelligent isolatieniveaubeheer, is het mogelijk om de doorvoer te optimaliseren en tegelijkertijd de dataintegriteit te behouden. De ontwikkeling en validatie van een Python-gebaseerde tool en het robuustheids- en allocatie-algoritme ondersteunen toekomstig onderzoek en experimenten, wat kan leiden tot praktische verbeteringen in het veld. De resultaten tonen het belang aan van robuuste allocaties voor de prestaties van databasebeersystemen, hoewel verder onderzoek nodig is om de tool te valideren en te verfijnen.

Summary

Introduction

The thesis addresses a crucial challenge in database management systems: balancing transaction throughput with anomaly minimization. With data's ever-increasing volume and complexity, ensuring that databases remain both performant and reliable is more important than ever. This work focuses on robustness in transaction allocations, particularly within the multi-version concurrency control (MVCC) framework. The main objective was to develop and implement a Python-based tool that facilitates transaction throughput tests and to implement the robustness and allocation algorithm for mixed isolation levels.

Motivation and objectives

The motivation behind this research stems from the need to handle concurrent transactions in a way that does not sacrifice performance to maintain data integrity. Traditional mechanisms do maintain data integrity but often sacrifice performance for this. This thesis aims to examine the trade-off between throughput and anomalies. And develop strategies to optimize both simultaneously. The specific objectives are:

- Developing a tool to facilitate transaction throughput test.
- Implementing the robustness and allocation algorithm for mixed isolation levels.

Robustness and isolation levels

Database management systems support multiple isolation levels, each with its own trade-offs between throughput and anomaly minimization. The isolation levels discussed in this paper are Read Committed (RC), Snapshot Isolation (SI), and Serializable Snapshot Isolation (SSI).

- **Read Committed:** Ensures that data read is committed at the moment it is read. This level prevents dirty writes but does not protect against other anomalies.
- **Snapshot Isolation:** Provides a snapshot of the database at the start of the transaction, preventing anomalies like non-repeatable reads. But still allowing write skew.
- **Serializable Snapshot Isolation:** The strictest level, ensuring full serializability by preventing all possible anomalies. This does result in a lower performance.

Robustness in this work refers to the property of ensuring that every possible schedule of transactions is serializable under a given allocation. A robust allocation guarantees that the database remains consistent regardless of the transaction execution order.

Tool development

A significant contribution of the thesis is the development of the three Python-based tools designed to facilitate transaction throughput tests. The three tools are:

- **Creating configurations:** Allows to create configuration files for different experimental setups easily.
- **Experiment execution:** Supports the execution of throughput experiments by simulating concurrent transactions in a controlled environment.
- **Visualization of the results:** Provides visualizations to analyze and compare the results of different configurations.

These tools have proven to be essential in systematically comparing the results of different configurations.

Robustness algorithm

The implementation of the robustness and allocation algorithm is the second core part of this thesis. This algorithm was designed to ensure that the optimal robust allocation is given due to any given set of transactions. The algorithm works by starting at the strictest allocation and then lowering the isolation level for each transaction at a time. This change is kept or undone by checking whether the newly achieved allocation is robust. The implementation is verified by using known (optimal) robust allocations. In total, 23 allocations were tested. The result of all of them was correct.

Experiments and results

The experiments done aim to examine the influence of isolation levels on performance and anomalies. The experiments showed that the optimal robust allocation outperforms the allocation where all the transactions are mapped to SSI. However, this was not always the case. Sometimes, the experiments showed unexpected behavior, where one of the programs was run much more than the others. The reason for this is still undetermined, and further experimentation is necessary.

Conclusion

This thesis focuses on the critical importance of robustness in multiversion concurrency control. By balancing throughput and anomaly minimization through intelligent isolation level management, it is possible to optimize throughput while maintaining data integrity. The development and validation of a Python-based tool and robustness and allocation algorithm supports future research and experiments, which might lead to practical improvements in the field. The results show the importance of robust allocations in the performance of database management systems, although further research is needed to validate and fine-tune the tool.

Contents

1	Objectives	9
1.1	Introduction	9
1.2	Goals	10
2	Theoretical background	11
2.1	Transaction	11
2.2	Schedule	11
2.3	Conflict serializability	12
2.4	Isolation levels	13
2.4.1	Read committed	14
2.4.2	Snapshot isolation	15
2.4.3	Serializable snapshot isolation	16
2.5	Allocation	16
2.6	Robustness	17
3	Facilitating of transaction throughput tests	19
3.1	Benchmarks	19
3.1.1	Smallbank	19
3.1.2	Micro	20
3.1.3	Microplus	22
3.2	Description	22
3.2.1	Configuration format	23
3.2.2	Generating config files	24
3.2.3	Running the experiment	25
3.2.4	Results format	27
3.2.5	Visualising the results	29
3.3	The Core package	31
3.3.1	File structure	32
3.3.2	The protocol	32
3.4	Verification of the implementation	36
3.4.1	Running existing throughput experiments	36
3.4.2	Running existing anomaly experiments	38
4	Robustness and allocation algorithm	41
4.1	Description	41
4.2	Implementation	42
4.3	Verification	45
5	Experiments	47
5.1	Anomaly/performance trade-off experiment	47
5.1.1	Experimental setup	47
5.1.2	Results	47
5.2	Follow-up experiments	48

5.2.1	Experimental setup	48
5.2.2	Results	50
6	Conclusions	53
A	Example formats	57
A.1	Configuration file	57
A.2	Results file	58
B	Configurations	61
B.1	Throughput experiment	61
B.2	Anomaly experiment	62

Chapter 1

Objectives

1.1 Introduction

It is essential to store data in database systems without it ever becoming corrupt. Hence, database systems have various methods to ensure the data stays intact. One of these is transactions; a transaction is a sequence of operations. What makes it unique is that when using transactions, all operations must be executed, or none are executed. This ensures that the database is always in a state where the data is intact. However, when multiple users execute transactions in the database, it becomes a bit more complicated.

The transactions executed simultaneously are called concurrent. Moreover, transactions are concurrent when the second transaction starts before the first one is finished. When this happens, there are some unwanted scenarios. For example, two people have a shared bank account and person A withdraws €50. But in the meantime, person B does the same. Suppose that initially, the account contains €100. The wanted outcome would be €0 after two withdrawals of €50. However, because both happen simultaneously, both transactions read €100 as the initial value and remove €50, making the new balance €50. Because of this, both persons will have withdrawn €50, but the account will still contain €50 instead of €0. This is not wanted by the bank and is called a dirty write, since the changes of the first transaction are ignored by the second.

Handling concurrent transactions is called concurrency control. There are multiple methods of doing so, but all strive for the same thing: serializability. Serializability makes sure there are no anomalies and thus keeps the data intact. The various methods achieve this by using locking or abort mechanisms. Locking mechanisms ensure the transaction is the only one writing or reading the field and prevent other transactions from writing or reading the field while the initial transaction is still executing. Aborting mechanisms will abort a transaction to restore the database to a valid state. However, this is slower than doing nothing and hoping for the best [Wik24a].

The method this thesis uses is called multi-version concurrency control. This method makes multiple versions of the database. This enables users to immediately read older versions of fields when another transaction has a write lock on this field. These versions can be called snapshots. Thus, each user sees a snapshot of the database. Other users will not see changes a write operation makes until the transaction has been completed [Wik23].

Database management systems also support multiple isolation levels. The names of these can vary from system to system. These isolation levels increase the throughput at the cost of anomalies. The stricter the isolation level, the fewer anomalies are allowed at the expense of throughput. Logically, the less strict an isolation level is, a higher throughput can be achieved at the cost of more anomalies.

Transactions don't always relate to each other in the same way; for example, there can be two transactions, A and B, and one time, A is finished before B. Another time, B is finished before A even is started. How transactions relate to each other can be written in a schedule. This shows when each operation of each transaction is executed. Another concept is allocations; an allocation maps each transaction to an isolation level. For example, two transactions T_1 and T_2 can be mapped to one of the isolations: I_1, I_2, I_3 . When mapping T_1 to isolation I_3 and T_2 to I_1 , we dictate to the database that transaction T_1 has to be run under isolation I_3 . The same goes for transaction T_2 . Another essential concept is the robustness property. When a set of transactions is called robust, every schedule that can be made and that is allowed by the allocation is serializable.

Two problems in this domain are the robustness and allocation problems. The robustness problem is to decide whether a given allocation for a set of transactions is a robust allocation. In other words, how can it be decided if all schedules possible for a given allocation are serializable? The allocation problem, then, is the problem of finding an optimal robust allocation for a set of transactions that does not run and allocate all the transactions to the strictest isolation level.

1.2 Goals

The main goal is to find the answer to the following research question:

How can robustness help towards increasing transaction throughput and decreasing anomalies?

To do this, the objectives of this thesis consist of implementing a tool in Python to support further research of the robustness and allocation problems by;

1. Facilitating transaction throughput tests.
2. Implementing the robustness and allocation algorithm for mixed isolation levels (RC, SI, SSI).

Throughput tests are essential to make a comparison between two allocations. These tests consist of running transactions in a database for a given amount of time and counting how many are successfully executed. With these tests, not only is the throughput measured, but the number of violations is also measured. This enables the user to look at the trade-off between throughput and the number of violations when changing the allocation. Not only is it interesting to change the allocation, but changing other parameters that influence the contention rate might be interesting. It might be that for a certain amount of contention, the behaviour of an allocation changes. Implementing the robustness and allocation algorithm is essential to test whether the algorithm is correct. This can be done by using known robust allocations. Afterwards, it can be used to find the optimal robust allocation.

The structure of the subsequent chapters is as follows. Chapter 2 will explain the necessary definitions. Next, chapter 3 will go over goal 1, how this was achieved, what the encountered problems were and how it is validated. Following, chapter 4 will do the same as chapter 3 but then for the second goal. New experiments will be discussed in chapter 5. Finally, the conclusion will be discussed in chapter 6.

Chapter 2

Theoretical background

Before diving in, it is essential to get some terminology out of the way, hence the following section.

2.1 Transaction

Let's take a look at a transaction. A transaction consists of one or more operations that form a higher-level task. A transaction can consist of three types of operations: a read operation reading field t , written by $R[t]$, a write operation writing field t , written by $W[t]$ and a special commit operation denoted by C . This is always the last operation in the transaction and makes all the changes the transaction made definitive. A vital transaction property is that all of the operations must be executed. This makes sure the data stays intact. An example where a transaction can be used is as follows. You go out with a friend, but they forgot their wallet. So you pay for them, and they promise to wire the money back to your account. The money has to be removed from their account and added to yours. When this is not done by using a transaction, it is possible your friend wired the money, and it is gone from their account. But something went wrong when adding it to your account, so now the money is lost. This scenario would not have been possible if this was done in a transaction.

When speaking of a set of transactions \mathcal{T} , we can give every transaction a unique id i . Transaction i can then be written as $T_i \in \mathcal{T}$. The operations of i can then be written as $R_i[t]$, $W_i[t]$ and C_i . We also assume that every transaction only has, at most, one read and one write operation per object.

2.2 Schedule

A (multiversion) schedule s over a set of transactions \mathcal{T} can be represented as a tuple: $(O_s, \leq_s, \ll_s, v_s)$ [VKN23]. Where O_s is the set of all the operations of transactions in \mathcal{T} , as well as a special operation op_0 writing the initial versions of all existing objects, \leq_s denotes the ordering of the operations, \ll_s is the version order denoting the version of the write operations occurring in s and v_s being the version function mapping each read operation a in s to either op_0 or to a write operation in s . If $v_s(a) = op_0$, then the version read by a is the initial version of this object. An essential property of op_0 is that the following conditions are required to be true: $op_0 \leq_s a$ for every operation $a \in O_s$, $op_0 \ll_s a$ for every write operation $a \in O_s$, and for every read operation a , $v_s(a) <_s a$ and if $v_s(a) \neq op_0$ then the operation $v_s(a)$ is on the same object as a . Intuitively, the op_0 operation is the first operation in the schedule, indicating the start.

An example of a simple multiversion schedule is shown in Figure 2.1. Arrows denote the version

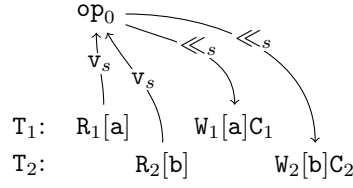


Figure 2.1: Example of a simple schedule

function and the version order. A read operation in transaction T_i on field a is denoted by $R_i[a]$. The same principle is applied for write operations. A schedule is called a single version schedule if \ll_s is compatible with \leq_s and every read operation reads the last value. A single version schedule is single version serial if the operations of multiple transactions are not interleaved. For example, when three operations a, b & c are structured as follows: $a \leq_s b \leq_s c$ and $a, c \in T$ then b must be also be in T .

2.3 Conflict serializability

To understand some definitions, it is crucial to understand the different kinds of conflicts that can occur between two transactions T_i and T_j in a set of transactions \mathcal{T} . Assume two operations b_i and a_j on the same object t , where a_j is an operation from transaction T_j and b_i is an operation from T_i . Then there are three different kinds of conflicts [VKN23], as shown below:

- ww-conflict: $b_i = W_i[t]$ and $a_j = W_j[t]$,
- wr-conflict: $b_i = W_i[t]$ and $a_j = R_j[t]$,
- rw-conflict: $b_i = R_i[t]$ and $a_j = W_j[t]$.

Note that the commit operations and the special operation op_0 never conflict with any other operation. These conflicts can also be written as dependencies [VKN23]. We say a_j depends on b_i in a schedule s over \mathcal{T} , denoted by $b_i \rightarrow_s a_j$ if:

- ww-dependency: b_i is ww-conflicting with a_j and $b_i \ll_s a_j$;
- wr-dependency: b_i is wr-conflicting with a_j and $b_i = v_s(a_j)$ or $b_i \ll_s v_s(a_j)$;
- rw-antidependency: b_i is rw-conflicting with a_j and $v_s(b_i) \ll_s a_j$.

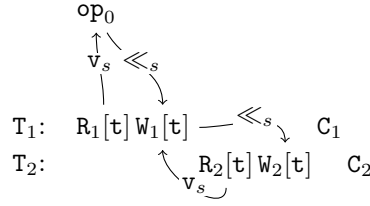
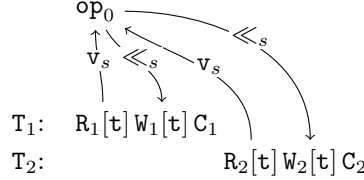
A ww-dependency means that a_j writes a version of an object earlier written by b_i . The latter part of the wr-dependency might look complex, but it just says that the value written by b_i is read by a_j or written before the value read by a_j . The same goes for the rw-antidependency.

Figure 2.2 shows all of these conflicts and dependencies. There is a ww-conflict between the two transactions since both write to t . For an example of a wr-conflict one can look at the write from T_1 and the read of T_2 . A rw-conflict can be found between the read operation of T_1 and the write operation in T_2 .

Definition 1. Two schedules, s and s' , are conflict equivalent if for every couple of conflicting operations b_i and a_j in s , there are a couple of conflicting operations b'_i and a'_j in s' . [VKN23]

Definition 2. A schedule is conflict serializable if it is conflict equivalent to a single version serial schedule. [VKN23]

A single-version serial schedule is a schedule where all the operations of a transaction are executed sequentially, and only when this transaction is finished is the next one executed. An example is shown in Figure 2.3.

**Figure 2.2:** Example of a schedule with all of the dependencies**Figure 2.3:** Example of a single version serial schedule

Another way of checking if a schedule is conflict serializable is by making its serialization graph $SeG(s)$ [VKN23]. Each transaction in the schedule is a node. There is an edge from T_i to T_j when an operation $a_j \in T_j$ depends on operation $b_i \in T_i$, which gives $b_i \rightarrow_s a_j$. Since not only the edges are useful to know but also the operations of the edges, there is a mapping function λ , which maps each edge to the corresponding pair of operations. This can be written as follows, $(b_i, a_j) \in \lambda(T_i, T_j)$ if there is an operation $a_j \in T_j$ depending on an operation $b_i \in T_i$. $SeG(s)$ can then be represented as a set of quadruples (T_i, b_i, a_j, T_j) giving all the possible pairs of the transactions T_i and T_j with all options of operations $b_i \rightarrow_s a_j$. Thus, one of these tuples can be seen as an edge. To check whether a schedule s is conflict serializable, we must check for cycles in the graph. A cycle Γ can be written as a non-empty sequence of edges:

$$(T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \dots, (T_n, b_n, a_1, T_1)$$

in $SeG(s)$. Note that every transaction is mentioned exactly twice. We can write a cycle with T_i as start and end as $\Gamma[T_i]$ while respecting the order of transactions in Γ . Then $\Gamma[T_i]$ is the following sequence:

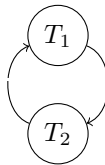
$$(T_i, b_i, a_{i+1}, T_{i+1}), \dots, (T_n, b_n, a_1, T_1), \dots, (T_{i-1}, b_{i-1}, a_i, T_i)$$

Then, a schedule s is conflict serializable if $SeG(s)$ is acyclic. In other words, if there are no cycles in the graph. An example of a serialization graph can be found in Figure 2.4

2.4 Isolation levels

An isolation level states what kind of situations in the schedule the database will allow and what situations are not allowed. The stricter the isolation level, the lower the achieved throughput will be. But this will result in fewer anomalies. The defined isolation levels are the following:

- Read committed (RC),

**Figure 2.4:** Serialization graph $SeG(s)$ for the schedule presented in Figure 2.2

- Snapshot isolation (SI),
- Serializable snapshot isolation (SSI).

2.4.1 Read committed

Let's start with read committed. This isolation level is the least strong out of the three. This means it won't abort a transaction as quickly as the others, but it doesn't guarantee serializability. For a transaction in a schedule s to be allowed under read committed, it has to satisfy some conditions.

We say that two transactions, T_1 and T_2 , in a schedule s follow the commit order when the following is true. Both of the transactions have a write operation to the same field. Assume that T_1 commits before T_2 does. To follow the commit order of the schedule, the write operation from T_1 has to come before the write operation from T_2 . Thus, for every write operation $W_i[t]$ in a transaction $T_i \in \mathcal{T}$ different from $T_j \in \mathcal{T}$ where \mathcal{T} is the set of transactions, we have $W_j[t] \ll_s W_i[t]$ iff $c_j <_s c_i$.

A read operation can be read-last-committed to itself or the transaction's start. This says what value the read operation reads. For example, when a read operation is read-last-committed to the start of a transaction T , written as $\text{first}(T)$, it reads the value from a snapshot taken at the beginning of T . While a read operation that is read-last-committed to itself reads the value from a snapshot taken right before the operation. Formally, a read operation $R_j[t]$ in a transaction $T_i \in \mathcal{T}$ is read-last-committed in s relative to an operation $a_j \in T_j$ if the following holds:

- $v_s(R_j[t]) = op_0$ or $C_i <_s a_j$ with $v_s(R_j[t]) \in T_i$; and
- there is no write operation $W_k[t] \in T_k$ with $C_k <_s a_j$ and $v_s(R_j[t]) \ll_s W_k[t]$.

The first condition states that $T_j[t]$ either reads the initial version or a committed version. The second condition states that $T_j[t]$ reads the most recently committed version of t .

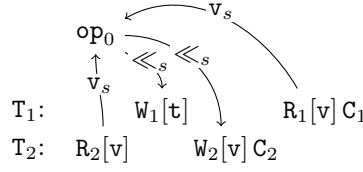
A transaction can contain a dirty write, this is defined as follows. Take two transactions, T_1 and T_2 , in the schedule as shown in Table 2.1. Each transaction contains a write to some field a . If transaction one writes to a first and transaction two writes to a before the other transaction can be committed, one of these values will be overwritten. This will be the value of the transaction that commits first, in this case, T_1 . Formally, a dirty write exists when there are two write operations $b_i \in T_i$ and $a_j \in T_j$ with $T_i \neq T_j$ such that $b_i <_s a_j <_s C_i$.

Now that it is clear when a write operation respects the commit order, a read operation is read-last-committed to itself or to the start of the transaction and what a dirty write is. Read committed [VKN23] can be defined as:

Definition 3. Let s be a schedule over a set of transactions \mathcal{T} . A transaction $T_i \in \mathcal{T}$ is allowed under isolation level read committed in s if:

- Every write operation in a transaction T_i where $T_i \in \text{schedule } s$ follows the commit order of s ,
- Every read operation from $b_i \in T_i$ is read-last-committed in s relative to b ,
- T_i does not contain a dirty write in s .

Figure 2.5 shows an example of a schedule where not all transactions are allowed under RC. This is because T_1 doesn't satisfy the second condition. $R_1[v]$ is read-last-committed to $\text{First}(T_1)$ instead of to itself. If it had read the value written by T_2 , T_1 would have been allowed under RC since both the other conditions are satisfied.

**Figure 2.5:** Schedule with a transaction that doesn't satisfy RC

T1	T2
Begin	
W(A)	
	Begin
	W(A)
Commit	
	Commit

Table 2.1: Example of a dirty write given two transactions T1 & T2

2.4.2 Snapshot isolation

The next isolation level is snapshot isolation. It is stricter than read committed but doesn't guarantee serializability. Thus, errors might still be made when running under this isolation level. Before looking at the definition of SI, it is important to understand what a concurrent write is. Take two transactions T_i and T_j with $b_i \in T_i$ and $a_j \in T_j$ two write operations on the same field:

- $\text{First}(T_j) <_s C_i$,
- $b_i <_s a_j$.

Definition 4. A transaction $T_i \in \mathcal{T}$ is allowed under isolation level snapshot isolation [VKN23] in s if:

- Every write operation in a transaction T_i where $T_i \in \text{schedule } s$ follows the commit order of s ,
- Every read operation from $b_i \in T_i$ is read-last-committed in s relative to $\text{first}(T_i)$,
- T_i does not contain a concurrent write in s .

Notice that the first condition is the same as that for read committed. The second condition, on the contrary, is a little different. Instead of being read-last-committed to itself, it has to be relative to $\text{first}(T)$. The last condition states that when there is a concurrent write between T_i and some other transaction, both transactions are not allowed under SI.

Figure 2.6 shows an example of a schedule that contains a transaction that isn't allowed under snapshot isolation. This is because it doesn't satisfy the second condition. Operation $R_1[v]$ is not read-last-committed to $\text{first}(T_1)$ since it reads the value written by $W_2[v]$. This value is written after the start of T_1 ; thus, it is not read-last-committed to $\text{first}(T_1)$. Figure 2.7 also shows an example of a schedule with a transaction that isn't allowed under snapshot isolation. Contrary to the previous example, this isn't allowed because of the last condition. This can easily be checked by looking at both conditions for a concurrent write. T_2 starts before T_1 commits, and the first transaction writes before the second does. Thus, there is a concurrent write between T_1 and T_2 .

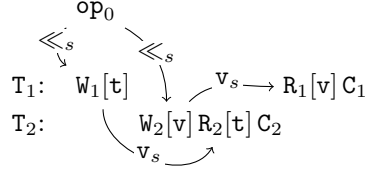


Figure 2.6: Schedule with a transaction that violates the second condition of SI

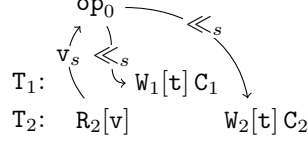


Figure 2.7: Schedule with a transaction that violates the last condition of SI

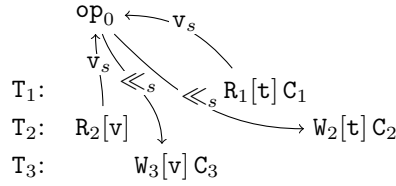


Figure 2.8: Schedule that contains a dangerous structure

2.4.3 Serializable snapshot isolation

The last isolation level is serializable snapshot isolation. This is the most strict and thus guarantees serializability. For a transaction to be allowed under SSI, it has to satisfy a condition over the whole schedule instead of a set of conditions defined for the transactions [VKN23]. The condition the schedule has to meet is that there cannot be a dangerous structure in it.

A dangerous structure consists of three transactions, $T_1 \rightarrow T_2 \rightarrow T_3$ in the schedule s . These transactions form a dangerous structure when the following conditions are met:

- There is a rw-antidependency between T_1 and T_2 and between T_2 and T_3 ;
- T_1 and T_2 are concurrent in s ;
- T_2 and T_3 are concurrent in s ;
- $C_3 \leq_s C_1$ and $C_3 <_s C_2$.

Notice that in the fourth condition, C_3 can equal C_1 . This is in case T_1 and T_3 are the same transaction. This is shown in Figure 2.8. There is a rw-antidependency from $R_1[t]$ to $W_2[t]$ and from $R_2[v]$ to $W_3[v]$. T_2 is concurrent with both other transactions. And the third transaction commits before transaction one or two does. Since all the above conditions are met, there is a dangerous structure $T_1 \rightarrow T_2 \rightarrow T_3$. Thus, none of these transactions are allowed under SSI. If a fourth transaction were not part of any dangerous structure, it would have been allowed under SSI, contrary to the other three.

2.5 Allocation

An allocation maps each transaction of the schedule to an isolation level [VKN23]. For example, two transactions T_1 and T_2 both present in schedule s , a possible allocation is $\mathcal{A}(T_1) = \text{RC}$ and $\mathcal{A}(T_2) = \text{SI}$. Thus, an allocation dictates the isolation level under which a transaction has to be run.

	T_1	T_2	T_3	T_4
\mathcal{A}_1	SSI	RC	SSI	SSI
\mathcal{A}_2	SI	SI	SSI	SSI
\mathcal{A}_3	SI	RC	SSI	SSI

Table 2.2: Robust allocations for \mathcal{T}_{ex}

Definition 5. A schedule s over a set of transactions \mathcal{T} is allowed under an allocation \mathcal{A} over \mathcal{T} if:

- for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) = RC$, the transaction is allowed under RC;
- for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) \in \{SI, SSI\}$, the transaction is allowed under SI;
- there is no dangerous structure $T_i \rightarrow T_j \rightarrow T_k$ in s formed by three transactions T_i, T_j, T_k with $\mathcal{A}(T_i) = \mathcal{A}(T_j) = \mathcal{A}(T_k) = SSI$.

The first condition says that when a transaction is allocated under RC. It has to satisfy the conditions to be allowed under RC. The second condition says that all the transactions allocated under SI or SSI have to satisfy the conditions to be allowed under SI. The third condition states that there cannot be a dangerous structure between three transactions when they are allocated under SSI.

2.6 Robustness

The robustness property is defined, which guarantees serializability for all schedules over a given set of transactions \mathcal{T} for a given allocation \mathcal{A} [VKN23].

Definition 6. A set of transactions \mathcal{T} is robust against an allocation \mathcal{A} for \mathcal{T} if every schedule for \mathcal{T} that is allowed under \mathcal{A} is conflict serializable.

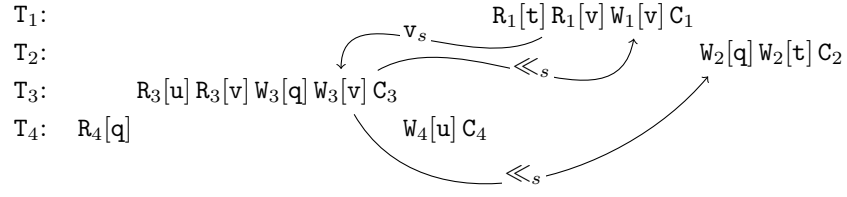
As an example, define the set of transactions $\mathcal{T}_{ex} = \{T_1, T_2, T_3, T_4\}$ over four different objects t, u, v and q as follows:

- $T_1 = R_1[t]R_1[v]W_1[v]C_1;$
- $T_2 = W_2[q]W_2[t]C_2;$
- $T_3 = R_3[u]R_3[v]W_3[q]W_3[v]C_3;$
- $T_4 = R_4[q]W_4[u]C_4.$

The most apparent robust allocation is, of course, the allocation that allocates every transaction to SSI. But this allocation is not interesting since it is not very fast. Some other robust allocations are given in Table 2.2. How it is determined whether an allocation is robust or not will be discussed later in chapter 4. For now, the most important thing to note is that the starting point is the allocation mapping each transaction to SSI. Notice how \mathcal{A}_1 only lowered the isolation of one transaction. The remaining two allocations each go a step further. An allocation that is not robust is the following:

$$\mathcal{A}_4(T_1) = SI, \mathcal{A}_4(T_2) = RC, \mathcal{A}_4(T_3) = SI \text{ and } \mathcal{A}_4(T_4) = SSI.$$

Notice that this allocation is precisely like \mathcal{A}_3 but now $\mathcal{A}(T_3) = SI$ instead of SSI. To know why this allocation is not robust, looking at the schedule that is not conflict-serializable, as shown in Figure 2.9, is helpful. Note that not all version functions and version orders are drawn to keep it simple. Since transactions one and two are not concurrent with any other transactions, these will not conflict with any other transaction and thus satisfy the conditions of all three isolation levels. Transactions three and four, however, are concurrent. But still, both are allowed under SI or SSI. However, the schedule is not conflict-serializable since there would be a cycle in the serialization graph between T_3 and T_4 . The allocation problem, as mentioned before, is the

**Figure 2.9:** Schedule of T_{ex}

problem of finding an optimal robust allocation. Finding a robust allocation is not tricky. Map each transaction to SSI, and we will have a robust allocation. Finding the best robust allocation is more complicated. We look for an algorithm that favours RC over SI and favours SI over SSI. This algorithm will be explained in chapter 4.

Chapter 3

Facilitating of transaction throughput tests

3.1 Benchmarks

The implementation supports a variety of three benchmarks. A benchmark is a set of predefined transactions that can be run on the database. The implementation supports three benchmarks: smallbank, micro and microplus. All of the benchmarks have some standard functionalities. For example, the methods used to select one or more rows for the transaction. Each of the benchmarks uses the same techniques.

3.1.1 Smallbank

The first benchmark that was implemented is smallbank [VKKN21]. The purpose of smallbank is to measure the throughput and abort rate with different contention parameters and allocations. The idea of smallbank is that it represents a bank; thus, the transactions are themed around this. The five transactions are Balance, Amalgamate, DepositChecking, TransactSavings and WriteCheck. The database for smallbank consists of three tables: Account, Savings, and Checking. The account table consists of a name and a customerID. The savings account has two columns: CustomerID, a foreign key to the Account table, and a Balance field, which specifies the balance in the savings account. The last table of smallbank is the Checking table, which is the same as the savings table but keeps track of the savings account's balance.

Balance

As mentioned before, smallbank has five transactions: Balance, Amalgamate, DepositChecking, TransactSavings and WriteCheck. The first, Balance, consists of three read operations. The first is retrieving the CustomerID, which is given a name. The second retrieves the savings account balance with the retrieved CustomerID, and the last transaction retrieves the sum of the balance from the checking account and the previously fetched savings account balance. In Figure 3.1 pseudocode of this transaction can be found.

Amalgamate

The second transaction, amalgamate, takes two parameters: the two accounts that need to be used. First, the transaction fetches the customerID of the first account. After this, it fetches the customerID of the second account. Now that it has both the customer IDs of the accounts, it updates the balance of the savings account of the first customer. The balance is set to zero, and the value it had is returned. Now that the balance of the savings account from the first

```

Balance(V):
    SELECT CustomerId INTO :C
    FROM Account
    WHERE Name = :V

    SELECT Balance INTO :B
    FROM Savings
    WHERE CustomerId = :C

    SELECT Balance + :B
    FROM Checking
    WHERE CustomerId = :C

```

Figure 3.1: Pseudocode balance

customer is emptied, the same can be done for the checking account. Now that the transaction has emptied both accounts of the first customer but has the values of the accounts before they were emptied, these values can be added to the balance of the checking account of the second customer.

DepositChecking

The third transaction, DepositChecking, takes two parameters: a customer's name and a numerical value representing the amount to deposit. The transaction first reads the customer ID for the customer with the name that was given as a parameter. Then, this ID is used to update the balance of the corresponding checking account. The new balance is the sum of the old value and the given value as a parameter.

TransactSavings

The fourth transaction, TransactSavings, also takes two parameters. Like DepositChecking, it takes a customer's name and a numerical value representing the amount to add to the savings account. Thus, the transaction starts by reading the customer ID of the customer with the given name. This ID is then used to update the balance of the corresponding savings account.

WriteCheck

The final transaction, WriteCheck, takes two parameters. Like the previous transactions, these are a customer's name and a numerical value, this time used to subtract from the balance. The transaction starts by reading the customer's ID with the given name. This ID then reads the corresponding savings and checks the account balance. The last operation depends on the relation between the balances and the given value. If the sum of both balances is smaller than the given value, the balance of the checking account is subtracted by the given value plus 1. If the sum of the balances is higher or equal to the given value, the current balance of the checking account is subtracted by the given value.

3.1.2 Micro

The second benchmark is the Micro benchmark [FGA09]. This benchmark was designed to measure the violation rate, which is defined as the number of anomalies divided by the number of transactions completed. Before explaining how this is done, it is essential to understand what the tables look like. The benchmark requires two tables, A and B. Both tables contain an ID and a value. The value is a numerical field that is later changed by the transactions. This field in the A table is initialized for every row with a value between 0 and 99. The value from the B table with the same row is 99 minus that from A. The constraint to check whether a

```

ChangeA(I):
    SELECT valueA INTO :A
    FROM A
    WHERE id = :I

    sleep(sleepTimeAB)

    SELECT valueB INTO :B
    FROM B
    WHERE id = :I

    sleep(sleepTimeBU)

    delta = 0
    if 0 <= :A + :B <= 99:
        if :A + :B < 50:
            delta = 50
        else:
            delta = -50

    UPDATE A
    SET valueA = valueA + delta
    WHERE id = :I

```

Figure 3.2: Pseudocode ChangeA

row is an anomaly is as follows: for every possible ID, the sum of the corresponding row values from both tables must be between 0 and 99, with 0 and 99 inclusive. For example, take ID=1, and then the value of A with ID=1 plus that of B with ID=1 must be bigger or equal to 0 and smaller or equal to 99. This ID is counted as an anomaly when this is not the case. Note that this constraint is not present in the database schema. This would have aborted the transaction when making a violation, thus resulting in no violations. Important to remark is that when transactions are run by themselves, they will never create a violation. Therefore, violations are only made when non-serializable operations are performed.

ChangeA

The first transaction, ChangeA, takes one argument: the row id to change the value. The transaction starts by reading valueA from the row with the given id. Before the transaction does anything else, it sleeps for a given amount of time. When this period is over, the value of B with the same ID is read. Then, the transaction sleeps again. The final operation that is executed is updating the value of A. But before this can be done, the necessary calculations are needed to determine the new value of A. This is done by checking if the row is in violation. If it already is in violation, the transaction doesn't change it because if it does, it might resolve the violation. Thus, if it isn't in violation, it checks whether the sum is lower than 50. When the sum is lower, 50 is added to A; when the sum is higher, 50 is subtracted from A. The pseudocode of this transaction can be found in Figure 3.2.

ChangeB

The second transaction, ChangeB, takes one argument, like changeA. This is the row ID to change the value. It might seem that this transaction does precisely the same as ChangeA but then for B, but this isn't entirely the case, as will be explained now. Just as ChangeA, this transaction starts by reading the value of A, followed by a sleep, and then by reading the value

of B. If this transaction had been the same as ChangeA, but then for B, the reads had to be in reverse order. Now that the transaction has the values of both A and B, it sleeps again before doing the calculations to change B. The transactions sleep as simulated busy times. And increase the duration of the transactions so that experiments don't need to run as long to have the same number of violations. The duration of each sleep is chosen from a normal distribution given by the parameters in the configuration file. The calculations to determine the value B has to be changed by are the same as those from ChangeA; check whether the row is already in violation; if the sum is lower than 50, this is added. When higher, 50 is subtracted.

ChangeAB

The final transaction, ChangeAB, like the previous transactions, takes one argument. This is the row ID where the values have to be changed. Just like the other two transactions, the values of A and B are being read within between sleep. Then, the transaction sleeps once more. As the name of the transaction says, it doesn't change A or B, but both. This happens by doing the same calculations as the other transactions, but the value is divided by two and added or subtracted from A and B.

3.1.3 Microplus

The final benchmark is Microplus. As the name says, this benchmark is based on the micro benchmark. The purpose remains the same: measuring the violation rate. Remember that the micro benchmark only had two tables, A and B; this benchmark adds a third table, C, with the same attributes as the other tables, an ID and a value. When it comes down to the transactions, the three transactions from the micro benchmark stay the same. But a fourth transaction is added: transferAB.

TransferAB

The fourth transaction of the microplus benchmark is the transferAB transaction. Like the others, this transaction takes one argument, the ID of the rows, to change. The first thing that happens is fetching the value from table C, where the ID is given. This value isn't used later in the transaction. This read increases the difference between RC and SI, SSI when two concurrent transactions write to the same tuple. This difference gets larger because when running under RC, one transaction waits until the other commits. But when running under SI or SSI, the transaction aborts and starts over. After this read, the transaction sleeps. Now that the sleep is done, the value of A is updated, and 100 gets subtracted. This is later added to the value of B, but not before the transaction sleeps. After the value of B is changed, the transaction is finished and committed. The pseudocode from this transaction is shown in Figure 3.3.

3.2 Description

The throughput tests are essential to compare the trade-off between throughput and the number of violations, ideally for multiple configurations. This process can be divided into three parts. They are creating configurations that might yield interesting results and using them to run the experiments. And finally, use the results to visualize them so they are easy to understand. This process can be seen as a pipeline. Creating the configurations can be seen as the first block, running the experiment as the second block, and visualizing the results as the final block. This pipeline can be seen in Figure 3.4. This section will go into detail about all of these parts. With the addition of the configuration format and the result format. These formats are essential to understand the implementation of the first and last part of the pipeline.

```

transferAB(I):
    SELECT valueC INTO :C
    FROM C
    WHERE id = :I

    sleep(sleepTimeBU)

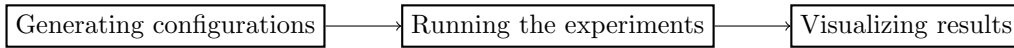
    delta = 100

    UPDATE A
    SET valueA = valueA - delta
    WHERE id = :I

    sleep(sleeptimeBU)

    UPDATE B
    SET valueB = valueB + delta
    WHERE id = :I

```

Figure 3.3: Pseudocode transferAB**Figure 3.4:** Pipeline showing the different parts

3.2.1 Configuration format

Before we could implement the tool, defining a format for the configurations was essential. All configuration files are in JSON; there are several reasons for this. JSON is easy to read, has smaller file sizes than XML, and supports fast data transmission. The information needed in a configuration can be split into three groups: information necessary for the database connection, general experiment info, and benchmark-specific information. Table 3.1 shows the information each group contains.

The first group contains attributes necessary to connect to the database, except for the ‘concurrent clients’ attribute. This attribute dictates how many clients will connect to the database while running the experiment. The next group contains attributes that provide general information about the experiment. Notice that there are three attributes related to the timing of the experiment. Warmup is needed because each client must establish a database connection. Since it doesn’t always have the same duration, this is a short time when clients start running transactions, but the system hasn’t started measuring. The attribute ‘experiment time’ is the time clients run transactions, and it is measured. The attribute ‘extra time’ is used for cooldown so that each client can finish its current transaction and the connection can be closed, but again, it isn’t measured for the results. The following two attributes of the general info are ‘super runs’ and ‘runs’. A run is simply the amount of times a timing cycle is run. A super run is how often each run is done; for example, when a configuration has two super runs and three runs, the timing cycle is run six times. These super runs are necessary for the experiments related to the micro and microplus benchmarks. In these experiments, the number of violations will be counted after the experiment is finished. When running a small number of runs but longer runs, we will reach the maximum number of violations, the number of rows. This is because once a row is a violation, it is not removed. Thus, with each violation, there are fewer options to create a new violation, so the rate drops. Because of this, we need the super runs. This allows us to run the experiment for the same total time but shorter runs. The idea is that with shorter runs, the point where the violation rate drops is not yet achieved. The following attribute in the configuration is the ‘experiment name’, which names the generated file(s). The ‘benchmark’

Database	concurrent clients
	URL
	port
	username
	password
General info	name
	warmup time
	experiment time
	extra time
	super runs
	runs
Benchmark specific	experiment name
	benchmark
	rows
	allocation for every transaction
	sampling weight for every transaction
	sampling method

Table 3.1: Table showing attributes per group

attribute, the last of the general info, is used so the program knows which benchmark the benchmark-specific attributes are for.

The benchmark-specific attributes are more complicated to explain since these are specific for each benchmark. Still, a couple of them are present in each benchmark, often under slightly different names. The number of rows states how many rows the database consists of. This could have been placed in the general info group, but since there was only one benchmark and this parameter was called ‘number of accounts’, it was renamed but never moved. The following attribute in the Table 3.1 is ‘allocation for every transaction’. This is for each transaction the isolation level in which it has to be run. Depending on the benchmark, the number and names of these attributes change; smallbank has five, while micro and microplus have three and four. The following attribute is the sampling weight for each transaction; this indicates how likely a transaction will be executed.

Finally, the last attribute is the sampling method, which defines how the row is chosen for the transaction. There are two options: hotspot or zipfian. The hotspot distribution has two parameters: the size of the hotspot and the probability of sampling from the hotspot. These parameters heavily influence the number of conflicts that will occur. The other distribution, zipfian, takes one attribute: zipfian skew. A higher number corresponds to more skew towards a small amount of rows. Hence, a higher skew results in more conflicts. The micro and microplus benchmarks have four additional attributes that smallbank doesn’t have. These attributes are related to the sleep used in these benchmarks’ transactions. Remember, each transaction sleeps two times. These four attributes give two normal distributions to determine these two durations. The attributes provide the mean and standard deviation of the normal distribution from which to sample the duration. An example of a configuration can be found in appendix A.1.

3.2.2 Generating config files

Having defined the configuration format, we can now look at the tool for generating these files. The tool was made using the Streamlit Python package. This is a very easy-to-use package for creating a web UI. The tool asks for two big groups of parameters: general information and benchmark-specific information.

Let’s start with the first group shown in Figure 3.5. It asks the values for the database and

general attributes mentioned in Table 3.1. However, two extra checkboxes exist: ‘custom setup’ and ‘generate config files for all isolation levels’. The custom setup checkbox allows the user to mix isolation levels when checked. When unchecked, the tool creates configuration files for all isolation levels: RC, SI, and SSI, which means a configuration where all transactions are run under read committed, one where all transactions are run under snapshot isolation and one where everything is run under serializable snapshot isolation. However, when creating a configuration file with mixed isolation levels, the custom checkbox is checked, and it is possible to turn off the creation of these files as shown in Figure 3.6. Notice the extra field ‘Experiment name’. This attribute is necessary for subsection 3.2.5 and will be explained there.

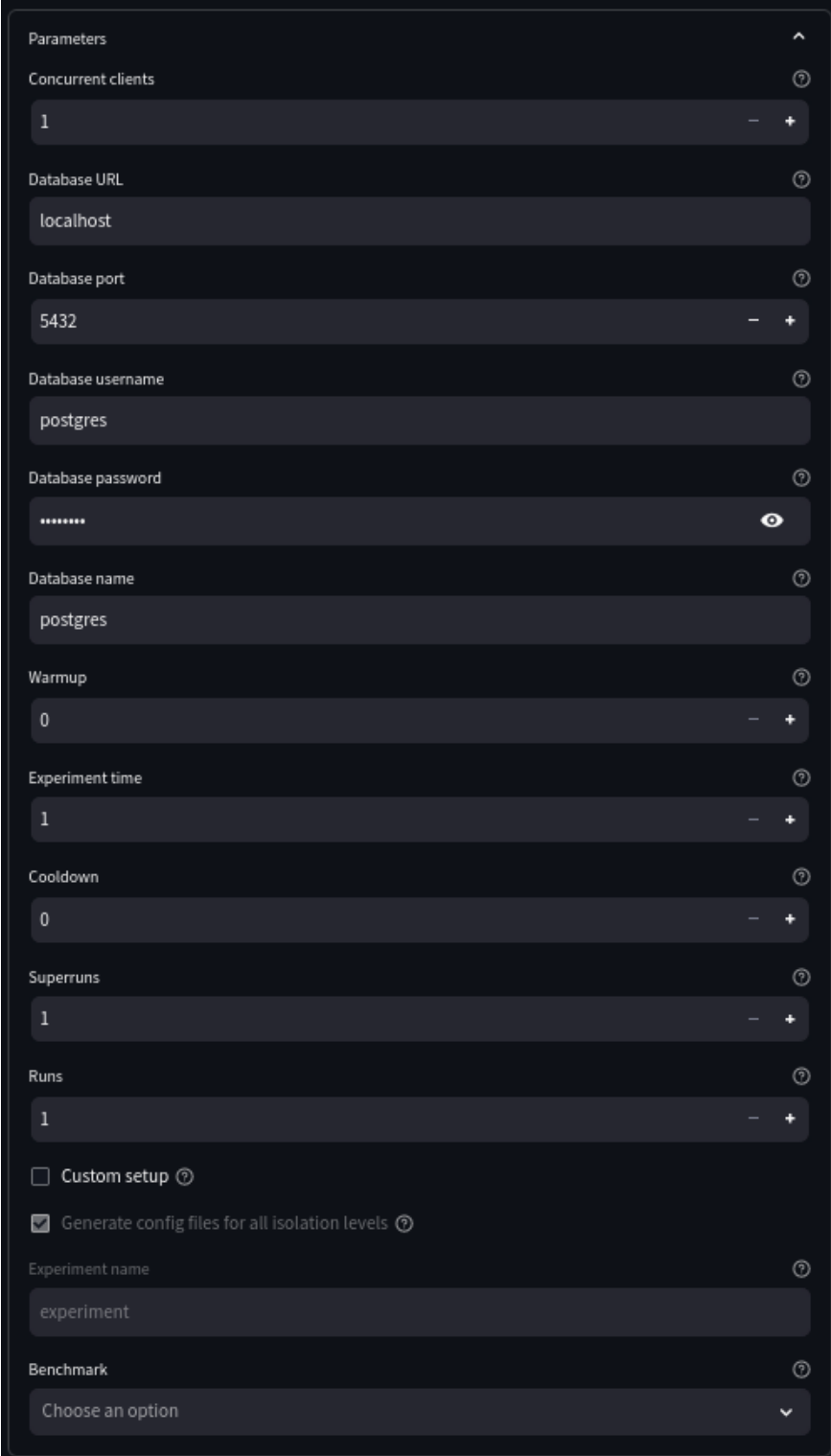
When the user has filled in all these attributes and chooses a benchmark, the benchmark-specific attributes become visible, as shown in Figure 3.7. Since the checkbox ‘custom setup’ is checked, the fields allowing the user to choose the isolation levels are shown. As seen in the figure, multiple values are possible for specific attributes. This is done by generating a configuration file for each value. This functionality helps create multiple files for multiple configurations at once rapidly. For example, it might be interesting to run experiments where the zipfian skew changes to see the effects this might or might not have. This prevents the user from needing to generate the configurations one by one and gives the option to do it in one click. A limitation is that using multiple values for only one parameter is possible.

The code for this tool isn’t that difficult. A number input in streamlit can be done using the provided ‘number_input’ function. This function takes various arguments to specify the range and other parameters of the number and returns the inputted value. This way, all of the fields are shown. The results of the fields are directly saved in a dictionary, which will later be added to the configuration file. Due to Streamlit’s intelligent way of dealing with changes made by the user, the data automatically updates in the dictionary. Notice that there are two expanders; the first shows the fields for the database and general attributes. The second shows the benchmark-specific fields. The first is made in the tool’s code itself, while the second is made by calling the ‘list_config’ function from the Core package, which will be discussed later in section 3.3. This function returns the dictionary containing all of the benchmark-specific attributes. Since this is stored as a JSON object in the config file, it is easy to merge with the database and general attributes. Writing the dictionaries to the files can be done by looping through the dictionaries and writing with the ‘dump’ function from the json package. In the case of multiple files, the value of the parameter iterated over is included in the name, together with the experiment name.

3.2.3 Running the experiment

Having generated the config files for the experiments, the user can now run the experiments with the second tool. This tool takes three command line arguments, two required and one optional: the path to the configuration file, the path to the results file, and the path to the folder where the logs need to be saved. The tool starts by reading the benchmark from the config file. If the config file hasn’t changed since the generation, this shouldn’t be a problem. But just in case, this is checked. With this name, the ‘benchmark’ function from the Core package (section 3.3) can be called. And an object from the corresponding benchmark is retrieved. The next thing that happens is calling the ‘check_config’ method from the Core package. This ensures the configuration file is in the correct format and contains all the right attributes. Remember the experiment needs to be executed $superrun \times run$ times. This can be achieved by using two nested loops. The first thing that happens before the experiment is executed is initializing the database. This can be done by the ‘init_db’ method from the Core package. When this is done, the helper function ‘start_processes’ is called.

The ‘start_processes’ function doesn’t run the experiment but makes the desired amount of subprocesses, which will, in turn, run the experiment. For this, the multiprocessing package from Python is used. From this, the Manager object is used. This provides a way to create data which can be shared between different processes. Three data types from this package are used:



The image shows a dark-themed configuration window for a database testing tool. It contains several sections with adjustable parameters:

- Parameters** (expandable section):
 - Concurrent clients**: A numeric input field set to 1, with minus and plus buttons for adjustment.
 - Database URL**: A text input field containing 'localhost'.
 - Database port**: A numeric input field set to 5432, with minus and plus buttons.
 - Database username**: A text input field containing 'postgres'.
 - Database password**: A password input field with masked characters (dots) and a toggle icon to show/hide the password.
 - Database name**: A text input field containing 'postgres'.
 - Warmup**: A numeric input field set to 0, with minus and plus buttons.
 - Experiment time**: A numeric input field set to 1, with minus and plus buttons.
 - Cooldown**: A numeric input field set to 0, with minus and plus buttons.
 - Superruns**: A numeric input field set to 1, with minus and plus buttons.
 - Runs**: A numeric input field set to 1, with minus and plus buttons.
- Custom setup**: A checkbox that is currently unchecked.
- Generate config files for all isolation levels**: A checkbox that is currently checked.
- Experiment name**: A text input field containing 'experiment'.
- Benchmark**: A dropdown menu currently showing 'Choose an option'.

Figure 3.5: General info in the tool

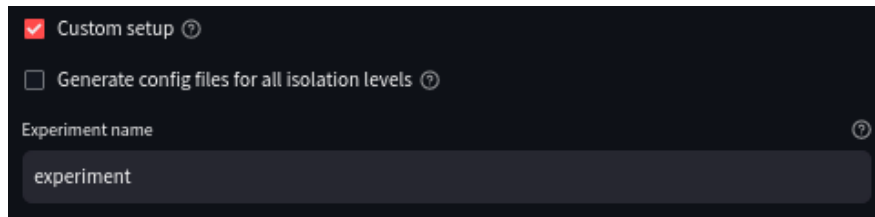


Figure 3.6: Figure showing the second checkbox can now be unchecked.

Value, Array and Dict. These make a synchronized version of a type—for example, an integer or an array of integers. These variables are necessary to merge the results atomically. Processes are created similarly to threads. Then, use the Process constructor from the multiprocessing package. This constructor takes a target as an argument, which is another function.

This function is the ‘run_benchmark’ function. This function takes a couple of arguments: the configuration dictionary, the object representing the benchmark-specific functionality, and arguments containing the results after running, such as the number of failed transactions and the number of completed transactions. The first thing the transaction does is set up the connection with the database. After the connection is established, the timer for the warmup can be started. A while loop then runs for the given warmup time in the configuration. When the warmup time is over, the header in the logfile is written. Now, the client can start the timer that will time the experiment. This, again, is measured with a while loop. In this loop, the client calls the ‘run_transact’ method from the Core package; this method will handle everything from now on and return how many times an abort was present, etc. Returning the data is done by getting a write lock on the results’ arguments. This prevents multiple clients from writing at the same time, resulting in the loss of data.

In doing this, I encountered some difficulties; the first problem was that I had never used JSON schema. Making the schema file for smallbank was the first challenge to overcome. I used an online validator to check incrementally whether my configuration was correct. Another issue was merging the results of all the different clients in the parent process. Since this needs to happen atomically, the data is not guaranteed to be intact if it does not. This is fixed by using the synchronized types from the multiprocessing package from Python, as mentioned before. Since clients can get locks on the variable, dirty writes can not happen anymore. Another difficulty I encountered was the errors PostgreSQL returns on aborts. The initial idea to identify the abort reason was to look at the type of error thrown by PostgreSQL. However, the types from the errors of SI and SSI are from the same type. Since the error messages differed, this issue was solved by looking at the keywords in this message. For SI, the words ‘concurrent update’ are always present, and when aborting due to a dangerous structure(SS1), the word ‘pivot’ is always present.

3.2.4 Results format

Before looking at the visualizations of the results, it is essential to understand the format in which the results are outputted. Just like the configurations, it is in JSON format. As shown in Figure 3.8, the file can be built as a tree. The first element is ‘superruns’, an array of JSON objects. Each object starts with the ‘run’ element, like ‘superrun’, an array of JSON objects. This is necessary since it is interesting to know how many transactions are completed per run. This array again consists of JSON objects. But all of them contain the same attributes.

The first attribute in this object is the ‘completedTotal’. As the name indicates, this represents the total amount of completed transactions in this run. Next up is the amount of failed transactions. However, unlike the completed, we are not interested in the total amount. However, knowing how many transactions failed under each isolation level is interesting. Thus, the ‘failed’ attribute consists of three attributes representing the three isolation levels. Since

micro parameters ^

The number of rows

100 - +

The weight(s) of the change A transaction

0.1 or 0.1|0.3 for multiple values

The weight(s) of the change B transaction

0.1 or 0.1|0.3 for multiple values

The weight(s) of the change AB transaction

0.1 or 0.1|0.3 for multiple values

The mean(s) of distribution used to request delay between read(valueA) and read(valueB) in ms

500 or 500|400 for multiple values

Standard deviation(s) of distribution used to request delay between read(valueA) and read(valueB) in ms

90 or 90|30 for multiple values

The mean(s) of distribution used to request delay between read(valueB) and updates in ms

500 or 500|400 for multiple values

Standard deviation(s) of distribution used to request delay between read(valueB) and updates in ms

90 or 90|30 for multiple values

The isolation level allocated to the change A program

RC v

The isolation level allocated to the change B program

RC v

The isolation level allocated to the change AB program

RC v

The sampling method used to sample accounts

zipfian v

The skew value used for the zipfian sampling method

0.1 or 0.1|0.3 for multiple values

Figure 3.7: Figure showing the parameters of the micro benchmark

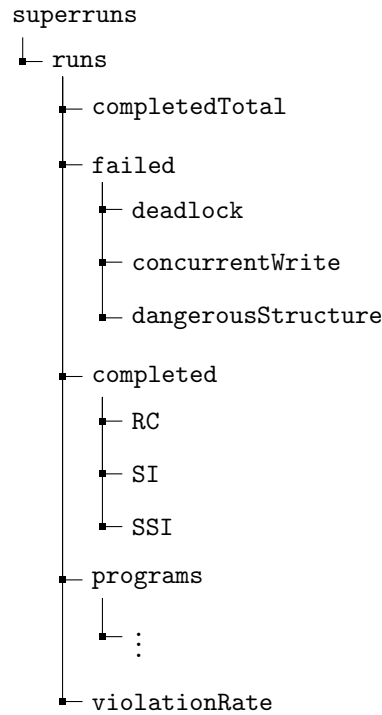


Figure 3.8: Tree structure of the result format

it is interesting to know how many transactions are completed per isolation level, the same is done here. Knowing the number of completed and aborted transactions per isolation level is also interesting per transaction type, also called a program. Thus, the following attribute is ‘programs’, which consists of JSON objects representing this for each program. Smallbank will have five, but micro and microplus will have three and four. Besides this, each program has an additional attribute called ‘runtime’, which dictates how much time was spent running this type of transaction. This was added to check if the weights given in the configuration are being respected. The final attribute each element in the array of runs contains is the ‘violationRate’. This attribute is only present in the results of micro and microplus since this can not be measured with smallbank. A complete results file can be found in appendix A.2

3.2.5 Visualising the results

Now that it is clear what a result file looks like, we can look at the third tool, which visualizes the results. The tool supports a variety of visualizations, changing for each benchmark. For smallbank, there are three visualizations available. The first shows the throughput shown in Figure 3.9, the second shows the Aborts per second shown in Figure 3.10, and the last is a stacked bar chart that shows the number of aborts per abort type shown in Figure 3.11. These three visualizations support iteration over the zipfian skew, hotspot probability and hotspot size attributes. To run the tool, three command-line arguments have to be provided: the folder containing the config files, the folder containing the result files, and the param to plot. The tool will automatically read all the files in these folders; thus, only relevant config files must be present.

Regarding the other benchmarks, the tool supports two types of visualization: one showing the violation rate on the y-axis and the parameter with multiple values on the x-axis. This visualization will later be shown in an experiment to validate the tool. The second plots the number of anomalies over the throughput as shown in Figure 3.12. This visualization is useful to see how an allocation compares against other allocations. Each dot represents an allocation.

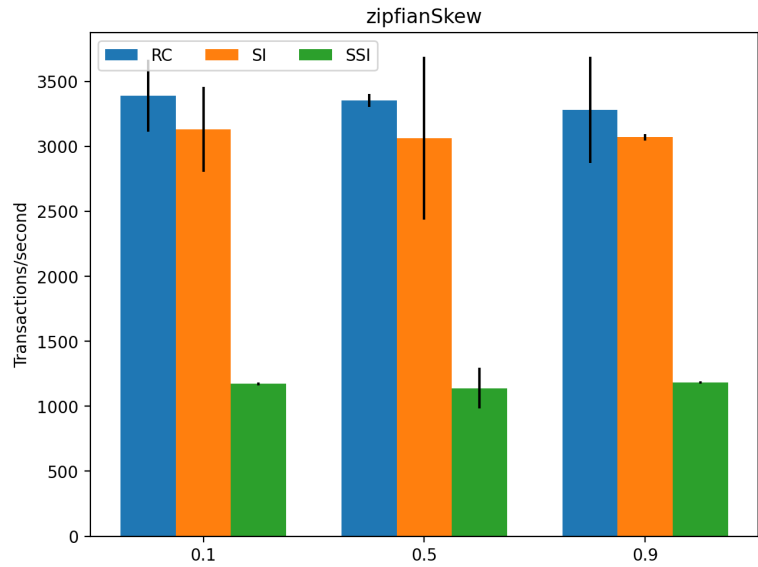


Figure 3.9: Plot showing the throughput

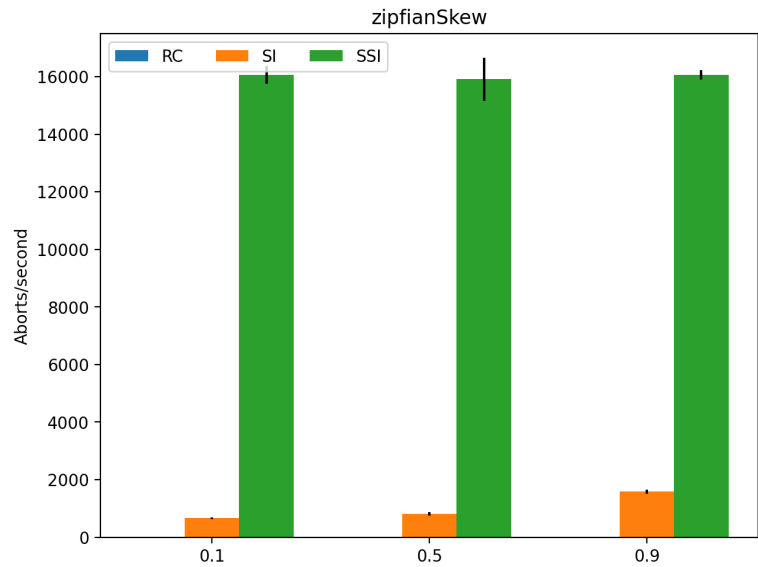


Figure 3.10: Plot showing the aborts/second

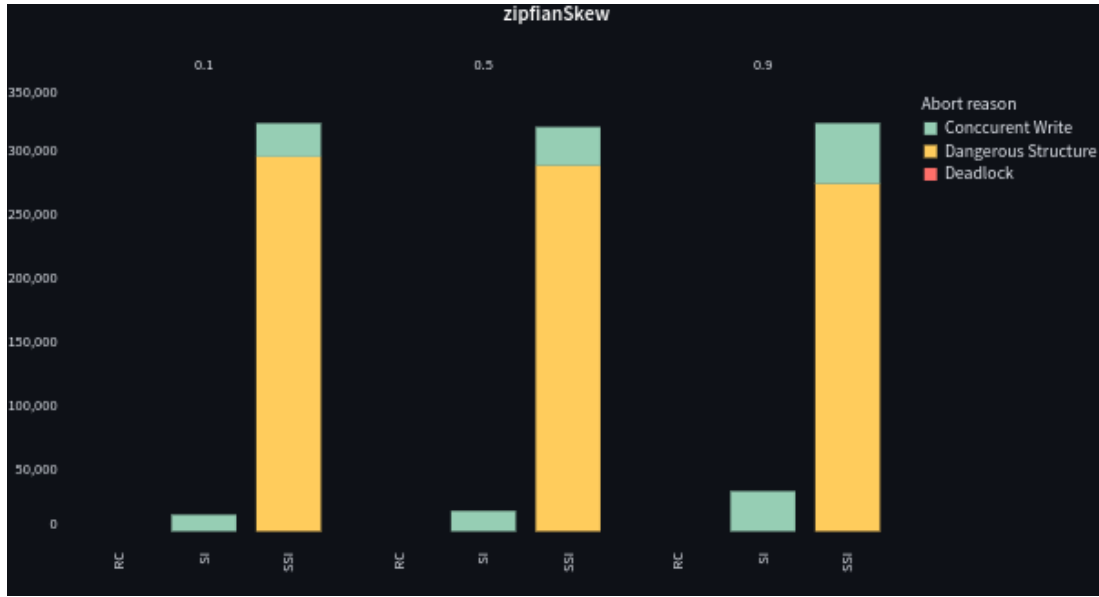


Figure 3.11: Plot showing aborts due to each abort type

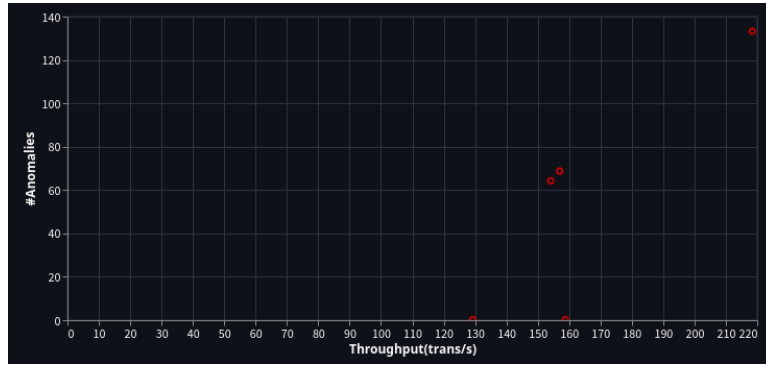


Figure 3.12: Figure showing the number of anomalies over the throughput for a configuration

When hovering over a dot, the allocation is shown. In this example, the two points with zero anomalies are the SSI allocation and the robust allocation.

3.3 The Core package

The core package is a package containing all the functionality specific to benchmarks. For example, the implementation of all the transactions, such as sampling the rows, running the transaction until it successfully executes, and providing a way to retrieve all the needed parameters, are included. When a tool needs to retrieve or execute specific functionality from the benchmark, the methods provided by this package are used. Since all of the benchmark-specific functionality is contained in this package, it is easy to add a benchmark since the tools utilizing this do not have to be changed. This section will start by explaining the file structure of the package and what these files are for. Then, the package will be discussed in depth. Moreover, the code structure and how each provided method works.

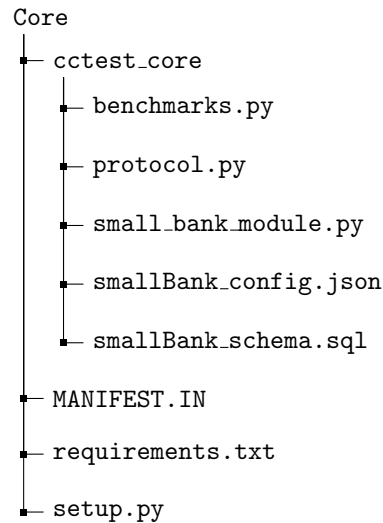


Figure 3.13: Snippet of the file structure of the Core package

3.3.1 File structure

The structure of the package is as shown in Figure 3.13. Notice that not only files for smallbank are shown; the files of the other benchmarks are in the same place as those of smallbank. The files directly in the Core folder are necessary to install the package. The requirements file lists all the packages needed, and the setup dictates how the package must be installed. The MANIFEST.IN includes all the necessary data files, such as the smallBank.config.json and smallBank.config.sql.

The actual code of the package is located in the cctest_core folder. The benchmarks.py file contains one function. This function takes one argument, a string representing a benchmark. For example, 'smallBank'. The function checks which benchmark this argument matches and returns the corresponding benchmark object. This is useful since the functions from benchmarks being called from outside the package are the same for each benchmark. Next is the protocol.py file. A protocol in Python is a way to define structural typing within the language. A protocol consists of a set of methods or attributes an object must have to be compatible with the protocol. In doing this, inheriting from a base class is unnecessary. However, using a protocol and still explicitly declaring has its advantages. For example, type checkers can statically verify that the class implements the protocol correctly. This is done in the package since it is easier to know if the protocol is implemented correctly.

3.3.2 The protocol

The protocol requires six methods, five of which are abstract and must be implemented by the benchmark classes. The first method is called 'init_db' and takes one argument, the dictionary containing all the information from the configuration file. The second method is 'run_transact'. This method takes two required arguments and one optional. A dictionary containing the configuration and the object representing the connection to the database is required. The optional argument is the name of the log file. Since the user can choose not to log this isn't needed. The second method is the 'check_config' method, which has one argument. Like the others, this is a dictionary containing the configuration parameters. The fourth method is the 'list_config' method, which takes two arguments: 'custom_isolation_levels' and 'default_isolation_levels'. Both arguments are booleans and are provided by the tool used to generate the configuration files. The final abstract method is 'check_consistency'; this method takes the dictionary representing the configuration file. The last method, which is not abstract, is the 'zipfian' method. This method is not abstract since this way of sampling stays the same

```

DROP TABLE IF EXISTS Account CASCADE;
DROP TABLE IF EXISTS Savings CASCADE;
DROP TABLE IF EXISTS Checking CASCADE;

CREATE TABLE Account (
    name VARCHAR(255) NOT NULL PRIMARY KEY,
    CustomerID INT GENERATED ALWAYS AS IDENTITY,
    UNIQUE (CustomerID)
);

```

Figure 3.14: Snippet of the `smallBank.sql` file

for all the benchmarks. It takes two arguments: the skew and n , where n represents the number of rows in the database.

Init_db

As mentioned, this method takes one argument: the dictionary representing the configuration. Since the database schema differs for each benchmark, it makes sense that this function cannot be the same for all the benchmarks. Before filling the database with the generated data, it is essential to make the tables. The corresponding SQL file is used; for `smallbank`, this is: ‘`smallBank.schema.sql`’ as mentioned before. A snippet of this file is shown in Figure 3.14. Notice the drop table commands in the beginning. These are present to spare the user from cleaning the database by hand each time an experiment is run. It also allows the user to run multiple experiments directly after each other using a script.

Now that the necessary tables have been created, the database can be filled with data. For `smallbank`, this is done by a for-loop, which goes from zero to the number of accounts specified in the configuration. In this for loop, three queries are run; the first inserts an account into the account table. The two other queries insert an entry in the checking and savings table with the ID of the previously inserted account. The balance of these accounts is randomly chosen between 100 and 10000. Since there is no constraint on what these balances need to be, this doesn’t matter too much. The principle stays the same regarding `micro` and `microplus`: the database is created by executing the corresponding SQL file. And then the data is added. This will be done for `micro` with two inserts, one for A and one for B; for `microplus`, there will be a third for C. The value for A is randomly generated between 0 and 99, B is 99 minus the value from A, and C is generated similarly to A.

Run_transact

The `run_transact` method, as the name says, runs the transaction. It starts by selecting which transaction to run. For this, the weights of each transaction are used in the configuration file. These weights illustrate how likely it is for a transaction to be executed. Before the transaction is executed, the function samples the necessary row(s) from the database. The ‘`sample_account`’ helper function does this. This function looks at what sampling method to use and the parameters of the chosen method. After successfully sampling, it returns the selected row(s) to the ‘`run_transact`’ function. Now that the rows have been sampled, the start time is written to the log file together with the chosen transaction and the row it will work on. Then, a while loop is started. This loop goes until the transaction is successfully executed; in case of an abort, the transaction is executed with the same parameters.

In this while loop, the helper function ‘`run_transact_once`’ is called; this function actually runs the chosen transaction. The result of this function is `None` in case of successful execution; in case of an abort, the error is returned. The number of aborts before the transaction successfully executes is counted, and the error why it aborts. When the transaction is executed, the timer stops, and the duration is saved. Now, all diagnostic data is written in the log file.

```

"accountSamplingMethod": {
  "type": "enum",
  "values": ["zipfian", "hotspot"]
},
"if": {
  "properties": {
    "accountSamplingMethod": { "const": "zipfian" }
  }
},
"then": {
  "properties": {
    "zipfianSkew": { "type": "number" }
  },
  "required": ["zipfianSkew"]
},
"else": {
  "properties": {
    "hotspotSize": { "type": "integer" },
    "hotspotProbability": {
      "type": "number",
      "minimum": 0,
      "maximum": 1
    }
  },
  "required": ["hotspotSize", "hotspotProbability"]
}

```

Figure 3.15: Snippet of smallbank's JSON schema

This is the transaction, number of aborts due to deadlocks(RC), number of aborts due to concurrent writes(SI), number of aborts due to dangerous structures(SSl), start time, end time, duration and finally, the row that was used. Finally, the method is finished and can return the results. These results are almost the same as the data being logged: number of aborts due to deadlocks(RC), number of aborts due to concurrent writes(SI), number of aborts due to dangerous structures(SSl), the isolation level the transaction was completed under, the transaction that was chosen and executed and finally the duration.

Check_config

The 'Check_config' function takes the configuration dictionary as an argument and checks whether all required attributes are present and of the correct type. This is done by using JSON schema. JSON schema allows the definition of a json template, which can be used to validate json files against. This schema is different for each benchmark since each benchmark requires different parameters. A small snippet of the schema of smallbank can be seen in Figure 3.15. Notice that this snippet contains details on how the sampling method's attributes. Thus, this is present in all the JSON schemas of the benchmarks. The function loads this schema and uses the validate function provided by the JSON schema package to validate if the given dictionary is correct. If it isn't, the error is shown to the user. However, this isn't the case when using a configuration file the tool generates unless it was changed.

List_config

The 'List_config' function takes two parameters: 'custom_isolation_levels' and 'default_isolation_levels'. As noted, both of these are booleans. Representing whether or not a corresponding checkbox in the tool for generating configuration files was checked. The first boolean represents

the checkbox whether the user wants to mix isolation levels. The second boolean represents the checkbox whether the user wants the configurations for default isolation levels generated. This function makes the UI for the benchmark-specific attributes earlier shown in Figure 3.7. The main difficulty in this function lies in checking the filled-in data. For example, it needs to be checked that only one attribute contains multiple values.

Another difficulty is generating the wanted files. For this, the two parameters are required. Until this point, the function has constructed an array containing dictionaries representing the filled-in values. However, these dictionaries either have custom isolation levels or no isolation level yet. There are a couple of possible scenarios regarding these two parameters: either the user has checked both, has only checked the box for custom isolation levels or hasn't checked the custom isolation level box and thus wants the default isolation levels. When both are checked, it is easy; we need to add four to the final output for each dictionary we already have. One with the custom allocation and three for the default allocations. For this, a helper function is used, which always adds the default allocations and can also add the custom allocation. Because of this, the code for the case where only the default allocations need to be added remains the same. Except for the call to the helper function, which now doesn't allow for a custom allocation. When only the custom allocation is requested, we add the isolation levels to the dictionaries in the array. This logic is necessary for all the benchmarks; the difference lies in the fields that must be filled in and checked.

Check_consistency

The last abstract method is the 'Check_consistency' function. This function is a bit different from the others since there is no data consistency constraint in the smallbank benchmark. The function there doesn't return anything. But in the other benchmarks, this isn't the case. Since the data consistency constraint is the same for the micro and microplus benchmarks, this function remains the same for both. The function starts by connecting to the database; since this method is always called after the experiment is finished, the previously used connections are already closed. Then, a loop is present, from zero to the number of rows in the database. In this loop, two queries are run, one to select the value from A and one to get the value from B. When both values are fetched, the sum can be calculated and checked whether it is in the allowed interval.

Zipfian

The zipfian function is the only function that is the same for all the benchmarks. It takes two parameters: the skew and the number of rows. Before looking at the implementation, it is important to know what the zipfian distribution looks like [Wik24b]. The distribution on N elements assigns to the element of rank k the probability:

$$f(k; N) = \frac{1}{H_N} \frac{1}{k}$$

Where H_N is a normalization constant, the Nth harmonic number:

$$H_N = \sum_{k=1}^N \frac{1}{k}$$

However, in the implementation, the generalized version is used, where s is the skew:

$$f(k; N, s) = \frac{1}{H_{N,s}} \frac{1}{k^s}$$

where $H_{N,s}$ is a generalized harmonic number:

$$H_{N,s} = \sum_{k=1}^N \frac{1}{k^s}$$

The zipfian function will first calculate this generalized harmonic number by looping from zero to n , where n is the number of rows and adding the value to the total sum. When the harmonic number is calculated, a random number between zero and one is generated and multiplied by this harmonic number. This gives a number a with: $0 \leq a \leq H_{N,s}$. Now, the interval at which this value lies needs to be checked. This can be done by calculating the harmonic numbers again, but now, in this loop, it needs to be checked if the current harmonic number is bigger than the generated value. If so, this is the row that will be used.

3.4 Verification of the implementation

An excellent way to know if the tool's implementation is correct is by running existing experiments. This is how the tool has been verified. The experiments that have been redone can be split into two groups: the experiments correlated to the throughput benchmark, smallbank, and the experiments related to the anomaly benchmarks, micro and microplus.

3.4.1 Running existing throughput experiments

The experiments run again are those of the 'Robustness against Read Committed for Transaction Templates' paper. The experiment that has been redone is the one shown in Figure 3.16. This experiment aims to learn the impact of different contention parameters on the transactions/second and aborts/second. This results in configurations with both sampling methods: hotspot sampling & zipfian distribution, and different parameters for hotspot sampling such as hotspot probability: 0.1, 0.3, 0.5, 0.7 & 0.9 and hotspot size: 1000, 100 & 10. The experiment shows that with a larger hotspot size, the throughput drops for SI and SSI while RC stays around the same throughput. With a smaller hotspot, this effect only becomes more prominent. This makes sense since a smaller hotspot will result in more conflicts and, thus, lower throughput, even for RC. A very small hotspot (10 accounts) even shows that with a high hotspot probability, all three isolation levels barely achieve some throughput. The same happens when the zipfian skew increases, as seen in Figure 3.16d.

Now that it is clear how the existing experiment works, it is possible to look at the tool's results. These results are shown in Figure 3.17. We can see that the throughput behaves differently than in the experiments from the paper. The throughput of read committed even rises instead of dropping to almost zero, as seen in Figure 3.16a. When looking at snapshot isolation, the trend is that it falls as expected but not as much as in the paper. It reaches a higher throughput for a low hotspot probability than read committed. However, the confidence interval of the RC is quite large and overlaps with the bar of SI. Thus, this might have been a couple of bad runs. When it comes down to SSI, the throughput is always low. And stays the same when changing the hotspot probability. These remarks can be seen in Figure 3.17b, which shows the number of aborts per second. The figure shows that there are no aborts when running under RC. This follows the results of the paper, which are near zero. The abort rate of SI increases when the hotspot probability increases, which makes sense since the throughput dropped. The abort rate of SSI drops when the probability rises, which is not as expected since the throughput stays around the same value.

When comparing the results of the tool with the results from the paper, it is crucial to know the differences in parameters and the specifications of the machine on which it was run. First, look at the experimental setup from the paper. The database system was run on a server with two 2.3 GHz Xeon Gold 6140 CPUs with 18 cores each, 192 GB RAM, and a 200 GB SSD local disk. A separate machine issued the transactional workload to the database through a low-latency connection. Now, look at the setup of the experiments that were done to test the tool. The database was run on a laptop with a 4.463 GHz AMD Ryzen 7 5800H CPU with 8 cores, 16 GB RAM, and 146 GB SSD local disk. The tool itself was also run on this machine. It is also essential to compare the parameters of the configurations. The only difference is the duration of the experiment. The experiments done in the paper have a duration of 60 seconds

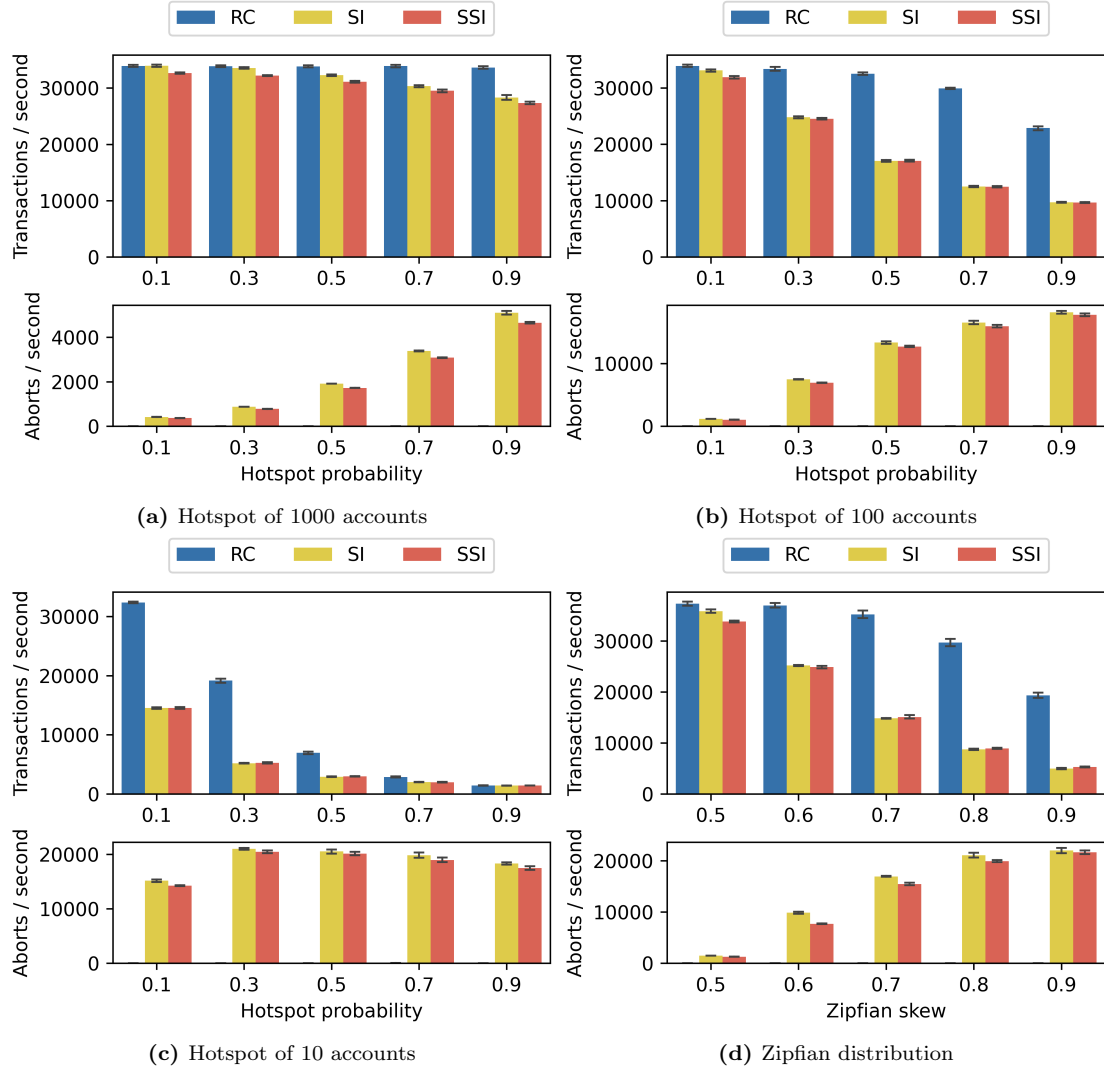


Figure 3.16: Results from the ‘Robustness against Read Committed for Transaction Templates’ paper [VKKN21]

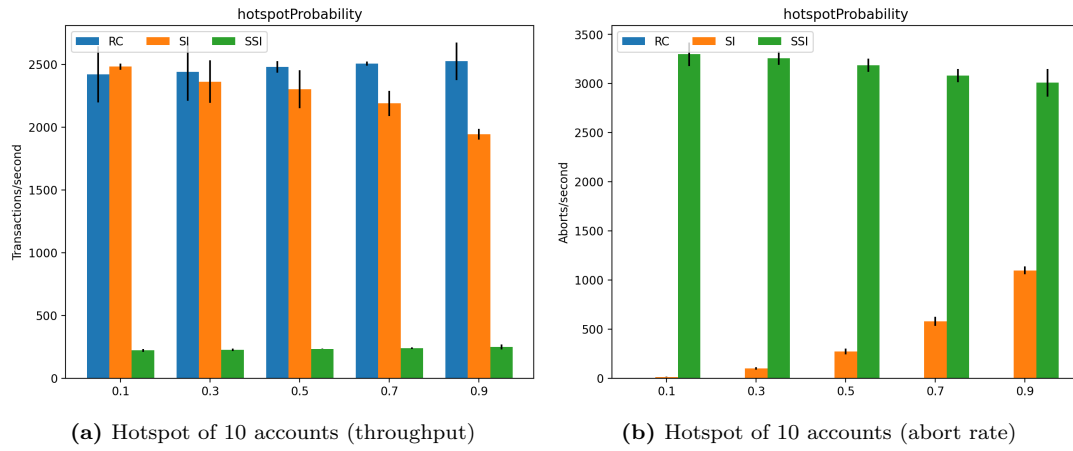


Figure 3.17: Results of the tool

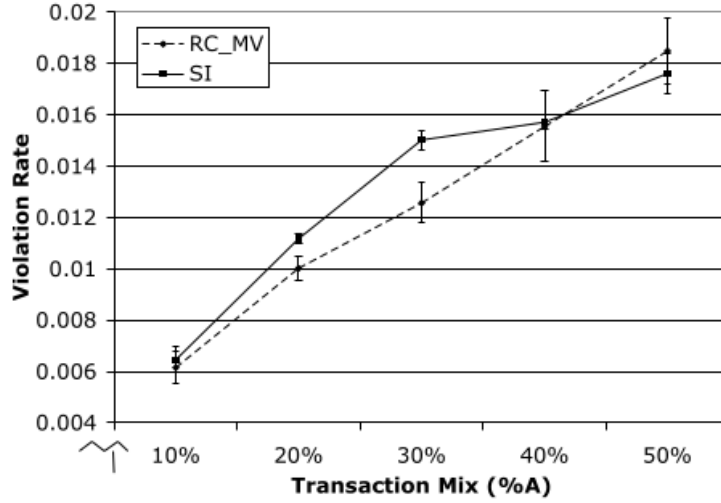


Figure 3.18: The inversion of the paper [FGA09]

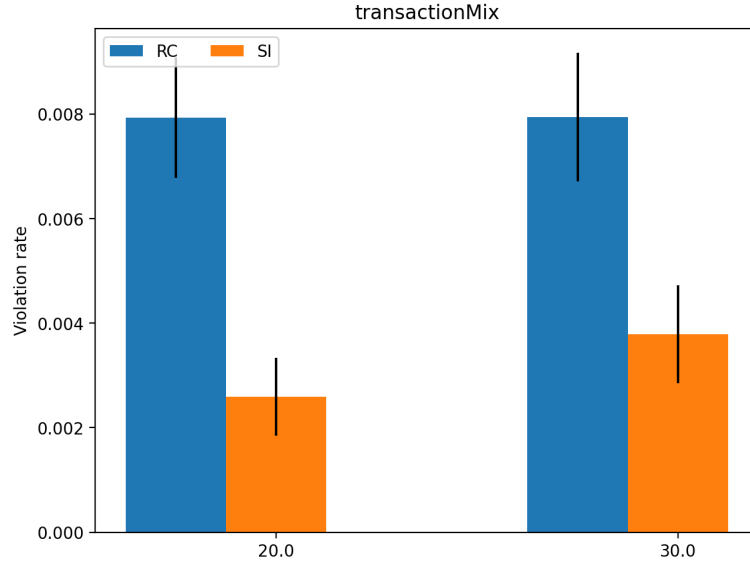
per run, while the experiments done with the tool have a duration of 20 seconds. The complete configuration can be found in appendix B.1.

The main difference in results is that the throughput of RC and SSI stays the same when the hotspot probability is increased. RC remains high, while SSI remains low. For SI, the tendency from the experiments in the paper is the same as the tool's tendency. The only difference is that the result from the tool is less extreme than that from the paper. The differences in setup can declare these differences in results. The amount of possible concurrent clients that execute transactions on the database is much more prominent with the setup from the paper. This explains the difference in results. Since there are not as many clients running transactions, the contention parameters, although the same, are a different percentage when compared with the concurrent clients. For example, a hotspot size 10 with 16 or 50 clients is a big difference; with 16 clients, only 6 choose a row already picked. But with 50, this immediately becomes 40. Thus resulting in less contention. This explains why RC does not drop as much as in Figure 3.16a. And why SI remains higher as well. Knowing this, the result of the tool with a hotspot size of 10 can be compared to the result from the paper with a hotspot size of 100 Figure 3.16b or 1000 Figure 3.16c. Now, the tool's results look more like the results from the paper. Considering this, the tool's results are trustworthy enough to be used in other possible new experiments.

3.4.2 Running existing anomaly experiments

The second experiment run again comes from the 'Quantifying Isolation Anomalies' paper [FGA09]. The experiment we are interested in uses the micro benchmark. It changes the mix of transactions and shows the violation rate. The specific experiment we are looking at is an experiment with an inversion in violation rate. SI got more violations than RC. This result is shown in Figure 3.18. The transaction mix refers to the percentage of 'ChangeA' transactions. Note that this benchmark has three transactions, 'ChangeA', 'ChangeB' and 'ChangeAB'. In this experiment, 'ChangeAB' was not used. The result shows that the inversion is most significant with a transaction mix of 20 and 30 per cent. The confidence intervals overlap at all the other points; thus, we can not be sure whether there is an inversion.

Since the inversion is clearest with a mix of 20 and 30 percent, these are the ones that have been run in the tool. The result of the tool is shown in Figure 3.19. Note that this is a bar chart instead of a line chart, but it still indicates whether or not there is an inversion. We can see that the inversion is not present when looking at the tool's result. However, when comparing

**Figure 3.19:** The result of the tool

Parameter	Value
Mean sleeptime AB	900
Sdev sleeptime AB	120
Mean sleeptime BU	100
Sdev sleeptime BU	30

Table 3.2: Parameters and values of the inversion experiment

the violation rate of RC with the violation rate of the experiment, this is around the same value. However, when looking at the violation rate of SI, we can see that it is much lower. This is the reason why the inversion is not present. To understand why it is essential to compare both experiments' experimental setup and parameters.

Start with the parameters. The essential parameters for the inversion are the sleep times and the transaction mix. The sleep times used for Figure 3.18 are shown in Table 3.2. The paper calculated these as the best values for creating the inversion. Thus, these values are also used in the experiment with the tool. Other parameters, like the number of concurrent clients, etc., also use the same values as described in the paper; a complete configuration is shown in appendix B.2. Therefore, this does not explain the difference in outcome. Let us now compare the experimental setup. The experiment with the tool was done on a laptop with a 4.463 GHz AMD Ryzen 7 5800H CPU with 8 cores and 16 GB RAM. When it comes down to the setup of the experiment done in the paper, we do not know. Since this was not mentioned, it is hard to make a comparison. Therefore, we cannot be sure if the difference in setup explains the difference in results. However, this is probably the case. If the setup from the paper was slower than the setup used to test the tool, it might be that the achieved throughput of the tool's experiment was higher, but only the same number of violations was reached. This would explain the lower violation rate.

Chapter 4

Robustness and allocation algorithm

4.1 Description

The second part is the algorithm for finding the optimal robust allocation of a schedule. The algorithm gives the optimal robust allocation for a given schedule. This is useful since finding a robust allocation is not hard. Allocate each transaction to SSI. But this is slow; hence, finding more optimal robust allocations, if not the most optimal, is interesting. Before discussing the algorithm for finding the optimal allocation, examining the algorithm that checks robustness against an allocation is necessary. This is shown in algorithm 1. The algorithm works by searching for a schedule that is not conflict serializable; thus, the allocation is not robust. For this, we introduce multiversion split schedules. We represent conflicting operations from transactions in a set \mathcal{T} as quadruples (T_i, b_i, a_j, T_j) with $b_i \in T_i$ and $a_j \in T_j$ conflicting operations, with $T_i, T_j \in \mathcal{T}$. These quadruples can be called conflicting quadruples for \mathcal{T} . For an operation b_i , we define $\text{prefix}_b(T)$ as the restriction of T to all operations that are before or equal to b according to \leq_T . The same goes for $\text{postfix}_b(T)$, the restriction of T to all operations strictly after b .

Definition 7. (Multiversion split schedule). Let \mathcal{T} be a set of transactions, \mathcal{A} an allocation for \mathcal{T} , and $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \dots, (T_m, b_m, a_1, T_1)$ a sequence of conflicting quadruples for \mathcal{T} such that each transaction in \mathcal{T} occurs in at most two different quadruples. A *multiversion split schedule* for \mathcal{T} and \mathcal{A} based on C is a multiversion schedule that has the following form:

$$\text{prefix}_{b_1}(T_1) \cdot T_2 \cdot \dots \cdot T_m \cdot \text{postfix}_{b_1}(T_1) \cdot T_{m+1} \cdot \dots \cdot T_n$$

where

1. there is no operation in T_1 conflicting with an operation in any of the transactions T_3, \dots, T_{m-1} ;
2. there is no write operation in $\text{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in T_2 or T_m ;
3. $\mathcal{A}(T_1) \in \{\text{SI}, \text{SSI}\}$, then there is no write operation in $\text{postfix}_{b_1}(T_1)$ ww-conflicting with a write operation in T_2 or T_m ;
4. b_1 is rw-conflicting with a_2 ;
5. b_m is rw-conflicting with a_1 or $(\mathcal{A}(T_1) = \text{RC} \text{ and } b_1 <_{T_1} a_1)$;
6. $\mathcal{A}(T_1) \neq \text{SSI}$ or $\mathcal{A}(T_2) \neq \text{SSI}$ or $\mathcal{A}(T_m) \neq \text{SSI}$;

7. if $\mathcal{A}(T_1) = \text{SSI}$ and $\mathcal{A}(T_2) = \text{SSI}$, then there is no operation in T_1 wr-conflicting with an operation in T_2 ; and
8. if $\mathcal{A}T_1 = \text{SSI}$ and $\mathcal{A}(T_m) = \text{SSI}$, then there is no operation in T_1 rw-conflicting with an operation in T_m .

Furthermore, T_{m+1}, \dots, T_n are the remaining transactions in \mathcal{T} (those not mentioned in C) in an arbitrary order. [VKKN21]

The multiversion split schedule splits a transaction T_1 into two parts. This split occurs at operation b_1 ; the two parts are the pre-and postfix, as mentioned. Between these parts, transactions 3 to m are placed. The eight conditions must be met to decide whether these transactions can be placed between the two parts of b_1 . These are the conditions in the last for loop on line 19. Note that these are broken into two groups as well. Conditions 1-3 make sure that the transactions in the split schedule s do not exhibit concurrent or dirty writes not allowed by \mathcal{A} . Conditions 4 and 5 ensure dependencies $b_1 \rightarrow a_2$ and $b_m \rightarrow a_1$ to occur in s and conditions 6-8 enforce that there is no dangerous structure over transactions allocated to SSI.

The algorithm does not search for a multiversion split schedule by iterating over all possible sequences C of conflicting quadruples since this number can be exponential in the size of \mathcal{T} . Instead, a mixed-iso-graph structure will be made (line 7 in algorithm 1). For a transaction T_1 and a set of transactions \mathcal{T} define $\text{mixed-iso-graph}(T_1, \mathcal{T})$ as the graph with as nodes all transactions in \mathcal{T} that do not have a conflicting operation with T_1 , and with an edge between transactions T_i and T_j if T_i has a conflicting operation with T_j . The algorithm iterates over all possible triples of transactions T_1, T_2 and T_m as defined in definition 7. And verifies if a path exists from T_2 to T_m as seen in algorithm 1 on line 20 where the reachable function is called. Thereby witnessing the existence of a corresponding sequence of conflicting quadruples between T_2 and T_m . By the definition of $\text{mixed-iso-graph}(T_1, \mathcal{T} \setminus \{T_1, T_2, T_m\})$, condition 1 of definition 7 is satisfied. The other conditions of algorithm 1 verify whether the other conditions of definition 7 hold for at least one option of $b_1, a_1 \in T_1, a_2 \in T_2$ and $b_m \in T_m$.

Now that algorithm 1 is clear, it is possible to look at algorithm 2. Note that the allocation stays robust when increasing ($\text{RC} \rightarrow \text{SI}, \text{SI} \rightarrow \text{SSI}$) an isolation level in the optimal robust allocation. The difference is that it is not the optimal allocation anymore. This principle can be used when finding the optimal allocation. Start with the allocation that allocates all transactions on SSI. Now, we iterate over the transactions and try lowering the isolation level. First, to RC, if this allocation is not robust, try if the allocation is robust when running this transaction under SI. If this is not the case, move this back up to SSI and move to the next transaction. When an allocation is robust with a transaction allocated to RC or SI, we can also move to the next transaction. This is precisely what algorithm 2 does. The check whether an allocation is robust is done by using algorithm 1.

4.2 Implementation

Now that it is clear what the algorithm looks like and how it works, it is possible to look at the implementation of it. The implementation is done in Python and uses one package, networkx. I started by implementing algorithm 1 since this is needed by algorithm 2. Let us begin with explaining the reachable function, defined on line 1 in algorithm 1. Instead of three arguments shown in the algorithm, I added a fourth, \mathcal{T} . This was necessary for making the mixed-iso-graph. Because of this, the other three arguments are just the id of the transaction in \mathcal{T} .

The first part of this function is exactly as in algorithm 1, but note that in the pseudocode, only one for loop is present that iterates over two things. This is done by a nested for loop—one for each operation to iterate over. The construction of the mixed-iso-graph is done in a helper function named ‘mixed_iso_graph’. This function takes the same four arguments as the reachable

Algorithm 1: Deciding robustness against an allocation.

Input: Set of transactions \mathcal{T} and allocation \mathcal{A} for \mathcal{T}
Output: True iff \mathcal{T} is robust against \mathcal{A}

```

1 def reachable( $T_2, T_m, T_1$ ):
2   if  $T_2 = T_m$  then
3     return True;
4   for  $b_2 \in T_2, a_m \in T_m$  do
5     if  $b_2$  conflicts with  $a_m$  then
6       return True;
7    $G := \text{mixed-iso-graph}(T_1, \mathcal{T} \setminus \{T_1, T_2, T_m\})$ ;
8    $TC := \text{reflexive-transitive-closure of } G$ ;
9   for  $T_3, T_{m-1}$  in  $TC$  do
10    for  $b_2 \in T_2, a_3 \in T_3, b_{m-1} \in T_{m-1}, a_m \in T_m$  do
11      if  $b_2$  conflicts with  $a_3$  and  $b_{m-1}$  conflicts with  $a_m$  then
12        return True;
13  return False;

14 def wr-conflict-free( $T_i, T_j$ ):
15   for  $b_i \in T_i, a_j \in T_j$  do
16     if  $b_i$  is wr-conflicting with  $a_j$  then
17       return False;
18  return True;

19 def ww-conflict-free( $b_1, T_1, T_2, T_m$ ):
20   for  $c_1 \in T_1$  do
21     if  $c_1 \in \text{prefix}_{b_1}(T_1)$  or  $\mathcal{A}(T_1) \in \{SI, SSI\}$  then
22       for  $c_2 \in T_2$  do
23         if  $c_1$  is ww-conflicting with  $c_2$  then
24           return False;
25       for  $c_m \in T_m$  do
26         if  $c_1$  is ww-conflicting with  $c_m$  then
27           return False;
28  return True;

29 for  $T_1 \in \mathcal{T}, T_2 \in \mathcal{T}, T_m \in \mathcal{T} \setminus \{T_1\}$  do
30   if reachable( $T_2, T_m, T_1$ ) and
31   ( $\mathcal{A}(T_1) \neq SSI$  or  $\mathcal{A}(T_2) \neq SSI$  or  $\mathcal{A}(T_m) \neq SSI$ ) and
32   ( $\mathcal{A}(T_1) \neq SSI$  or  $\mathcal{A}(T_2) \neq SSI$  or
33   wr-conflict-free( $T_1, T_2$ )) and
34   ( $\mathcal{A}(T_1) \neq SSI$  or  $\mathcal{A}(T_m) \neq SSI$  or
35   wr-conflict-free( $T_m, T_1$ )) then
36     for  $b_1 \in T_1, a_1 \in T_1, a_2 \in T_2, b_m \in T_m$  do
37       if ww-conflict-free( $b_1, T_1, T_2, T_m$ ) and
38        $b_m$  conflicts with  $a_1$  and
39        $b_1$  is rw-conflicting with  $a_2$  and
40       ( $b_m$  is rw-conflicting with  $a_1$  or
41       ( $\mathcal{A}(T_1) = RC$  and  $b_1 <_{T_1} a_1$ )) then
42       return False;

43 return True;

```

Algorithm 2: Computing the optimal robust allocation**Input:** Set of transactions \mathcal{T} **Output:** Optimal robust allocation \mathcal{A} for \mathcal{T}

```

1  $\mathcal{A} := \mathcal{A}_{SSI}$ 
2 for  $T \in \mathcal{T}$  do
3   if  $\mathcal{T}$  is robust against  $\mathcal{A} [T \rightarrow RC]$  then
4      $\mathcal{A} := \mathcal{A} [T \rightarrow RC]$ 
5   else if  $\mathcal{T}$  is robust against  $\mathcal{A} [T \rightarrow SI]$  then
6      $\mathcal{A} := \mathcal{A} [T \rightarrow SI]$ 
7 return  $\mathcal{A}$ 

```

function. Note that a graph needs to be constructed here. The networkx package is used for this. This package allows us to easily create an empty graph and add nodes and edges whenever possible. The nodes are added in a for loop that iterates over all the transactions in \mathcal{T} and checks whether the current transaction has an operation which conflicts with an operation of T_1 . After the nodes have been added, the edges are added in a two-deep for loop. The for loops iterate over all possible tuples in the graph, for each tuple is checked if an operation in one transaction conflicts with an operation in the other transaction. When this is the case, an edge is added. Now, the graph is finished and can be returned.

The second part of the reachable function (line 8-13 in algorithm 1) starts with the reflexive-transitive-closure of G . This is done with the built-in function from the networkx package. Note that this function only calculates the transitive closure. However, reflexive is an argument option. The outer for loop of the algorithm can be done as one for loop in the implementation due to the edges function of networkx. This returns a list of edges (tuples). The inner for loop, however, is done by a four-deep nested loop since this iterates over four operations.

Note that the algorithm often checks whether operations are conflicting. To do this, a helper function is made. Named ‘conflict_free’ and takes two arguments. Two transactions or two operations. However, both have to be the same; transactions and operations are not allowed to mix. When two transactions are given, the function iterates over all possible couples of operations. For each couple of operations, the function is recursively called again. In the case of two operations, it is checked if they both operate on the same field and if at least one is a write.

The following two functions are wr-conflict-free and ww-conflict-free as described in algorithm 1. The first checks whether there is a wr-conflict between the two transactions’ operations. This function iterates over the operations of both transactions with two nested for loops. The check is done like the ‘conflict_free’ function. But now, it checks if the operation of the outer query is a read and the operation of the inner query is a write. The ww-conflict-free function works a bit differently because it receives four arguments. However, the implementation of this function is exactly as described in algorithm 1. The prefix is calculated with the help of a helper function. This helper loops over the operations of the given transaction and adds them to a list until it comes across the given operation b_1 . Then, b_1 is added, and the list is returned.

The for loop on line 29 of algorithm 1 is almost the same as the algorithm dictates. The difference is how the transactions are given. The set of transactions \mathcal{T} and allocation \mathcal{A} are provided as a dictionary. The key of a dictionary is a number used as the identifier of the transactions. The value is a tuple. This tuple consists of the allocation followed by a list representing the transactions’ operations. Each operation is described as ‘X.y’, where X is the operation. Thus, R or W. And y is the field. Because of this, all the for loops that iterate over transactions iterate over the dictionary’s keys, hence why the implementation works with indices.

Now it is possible to look at the implementation of algorithm 2. This function takes one

	Bal	Am	DC	TS	WC
\mathcal{A}_1	RC				
\mathcal{A}_2		RC	RC	RC	
\mathcal{A}_3	RC		RC		
\mathcal{A}_4	RC			TS	
\mathcal{A}_5	SSI	RC	RC	SSI	SSI

Table 4.1: Robust allocations for the smallbank benchmark [VKKN21]

	T_1	T_2	T_3	T_4	robust?
\mathcal{A}_1	SSI	RC	SSI	SSI	yes
\mathcal{A}_2	SI	SI	SSI	SSI	yes
\mathcal{A}_3	SI	RC	SSI	SSI	yes
\mathcal{A}_4	RC	RC	SSI	SSI	no
\mathcal{A}_5	SI	RC	SI	SSI	no
\mathcal{A}_6	SI	RC	SSI	SI	no
\mathcal{A}_7	SI	RC	SI	SSI	no

Table 4.2: Robust allocation from the running example in [VKN24]

argument, just like the function that checks for robustness. It is \mathcal{T} and \mathcal{A} . This differs from the description in algorithm 2 where only \mathcal{T} is given. This is so that the default isolation \mathcal{A}_{SSI} can be given. And the format does not have to change when calling the ‘robust_against’ function.

Further, the function does exactly as described in the algorithm. The only difficulty was that tuples were immutable, so the tuple was converted into a list to change the isolation level. Then, the isolation level is updated, and the list is converted into a tuple again. So why work with tuples in the first place? This was to calculate sets like $\mathcal{T} \setminus T_1$. To do this, a copy is made of the original dictionary. However, when working with lists, the pointer is copied instead of created with a new value. This is not the case with tuples; hence, tuples are used instead of lists.

4.3 Verification

It is essential to make sure the implementation results are correct. For this, sets of transactions known as robust are run. These sets of transactions come from a couple of papers. These sets are shown in Table 4.1 and Table 4.2. The first table shows robust subsets of the smallbank benchmark. Note that all allocations are subsets of the possible transactions except for \mathcal{A}_5 . This shows the optimal robust allocation of the complete set of transactions. The second table shown in Table 4.2 shows allocations of the running example discussed in the paper and whether they are robust. In addition to these allocations, the following allocation of the microplus benchmark is also tested.

$$\mathcal{A}(\text{ChangeA}) = \text{SSI}, \mathcal{A}(\text{ChangeB}) = \text{SSI}, \mathcal{A}(\text{ChangeAB}) = \text{SI} \text{ and } \mathcal{A}(\text{TransferAB}) = \text{RC}$$

A separate testing script was made in Python to test all of these allocations. This script runs the ‘robust_against’ function (algorithm 1) for each allocation. In addition to the allocations given, changes were made. To the subsets of smallbank ($\mathcal{A}_1 - \mathcal{A}_4$ from Table 4.1), additional transactions were added and allocated under RC as well, these additional transactions can be new transactions not yet in the subset, or transactions that are already in the subset but need to be present multiple times not to be robust. These new allocations have to result in not being robust with a correct implementation. No changes were made for the allocations of Table 4.2,


```

(.venv) bram@archlinux ~/D/s/2/H/1/robustness (main)> python testing.py -smallbank True
(Bal)
(Bal, Am)
(Bal, WC)

(Am, DC, TS)

(Bal, DC)
(Bal, DC, TS)
(Bal, DC, WC)
(Bal, DC, Am)

(Bal, TS)
(Bal, TS, Am)
(Bal, TS, WC)

(Bal, WC, TS, DC, Am)

```

Figure 4.1: Results of the allocations in Table 4.1

```

(.venv) bram@archlinux ~/D/s/2/H/1/robustness (main)> python testing.py -highlights True
(Highlights RC RC SSI SSI)
(Highlights SSI RC SSI SSI)
(Highlights SI SI SSI SSI)
(Highlights SI RC SSI SSI)
(Highlights SI RC SI SSI)
(Highlights SI RC SSI SI)
(Highlights SI RC SI SSI)

```

Figure 4.2: Results of the allocations of Table 4.2

and these seven allocations are tested. Then, three variants of the microplus allocation are run. The isolation level is lowered for each of the transactions not yet allocated under RC. Thus, SSI \rightarrow SI and SI \rightarrow RC.

The results of all these allocations are shown in Figure 4.1, Figure 4.2 and Figure 4.3. Note that the green printed lines are robust, and the red lines are not robust according to the algorithm. When we compare this to the allocations in the mentioned tables, we see that this is correct. Even the added allocations with added transactions or lowered isolation levels give the correct result. From these experiments, we can conclude that the implementation of the algorithm works as expected. The implementation can be used to experiment further and find robust allocations.

```

(.venv) bram@archlinux ~/D/s/2/H/1/robustness (main)> python testing.py -micro True
(Microplus)
(Microplus SI SSI SI RC)
(Microplus SSI SI SI RC)
(Microplus SSI SSI RC RC)

```

Figure 4.3: Results of the microplus allocations

Chapter 5

Experiments

5.1 Anomaly/performance trade-off experiment

This experiment examines the trade-off between the number of anomalies and performance when using different allocations. A common assumption is that running with a lower isolation level increases the number of anomalies and the throughput. The experiment aims to prove that robust allocations indeed do not have any anomalies and achieve higher throughput than \mathcal{A}_{SSI} . The experiment hypothesizes that this will indeed be the case.

5.1.1 Experimental setup

The experiment was run on the Flemish supercomputer centre. The nodes used consist of two 2.3GHz Xeon Gold 6140 CPUs with each 18 cores, 192 GB RAM, and 200 GB SSD local disk. For the experiment, two nodes were used—one for the database and one for running the clients that execute the transactions. The PostgreSQL database used version 16.2; for Python, version 3.11 was used. The experiment focuses on the microplus benchmark and has the following parameters. The values of the parameters are shown in Table 5.1.

Note that the parameters needed to set up the database connection are not displayed. The parameters regarding the transactions' isolation levels and sampling weights are also not displayed. The isolation levels are not shown since multiple allocations were run. The run allocations are: \mathcal{A}_{RC} , \mathcal{A}_{SI} , \mathcal{A}_{SSI} , the robust allocation and the reversed allocation. The reversed allocation is the reverse of the robust allocation. Remember that the optimal robust allocation looks like:

$$\mathcal{A}(\text{ChangeA}) = SSI, \mathcal{A}(\text{ChangeB}) = SSI, \mathcal{A}(\text{ChangeAB}) = SI \text{ and } \mathcal{A}(\text{TransferAB}) = RC$$

Then the reversed allocation looks like:

$$\mathcal{A}(\text{ChangeA}) = RC, \mathcal{A}(\text{ChangeB}) = SSI, \mathcal{A}(\text{ChangeAB}) = SSI \text{ and } \mathcal{A}(\text{TransferAB}) = SSI$$

To compare these allocations with each other, all of the allocations needed to be run on the same nodes. Since the two nodes might be two nodes next to each other, they can also be far away from each other. The sampling weights are not shown since all of the weights are the same. Hence, every transaction has the same chance of being run.

5.1.2 Results

Now that the setup is clear, it is possible to discuss the results. The results were unexpected, and the robust allocation was slower than the allocation mapping all transactions to SSI. This can be seen in Figure 5.1. But this was not always the case, as shown in Figure 5.2. This raised

Parameter	Value
#concurrent clients	50
warmup	5
experiment	30
extra time	5
#super runs	2
#runs	1
#rows	200
Sampling method	hotspot
hotspot size	20
hotspot probability	0.5
sleep time AB	100
sdev sleep AB	10
sleep time BU	100
sdev sleep BU	10

Table 5.1: Parameters of the experiment

the question of why this was the case. The theory of why this was is that some programs run longer than others. The runtime per program was added to the result format to confirm this. When this was added, it was immediately apparent that this was indeed the case. In the case of Figure 5.1, the robust allocation ran ChangeA and ChangeB 800 seconds longer than the other programs. This only happened in the robust allocation, which is quite weird. The question arose about why these programs were run so much longer than the others. A possibility is that a client is stuck on a transaction right before the end of the experiment but keeps aborting, even after the experiment. Thus explaining the much higher runtime. To verify this, the option to log was added to the experiments. Each client has its log file, in which the transactions' start and end are written, along with the amount of aborts and final duration.

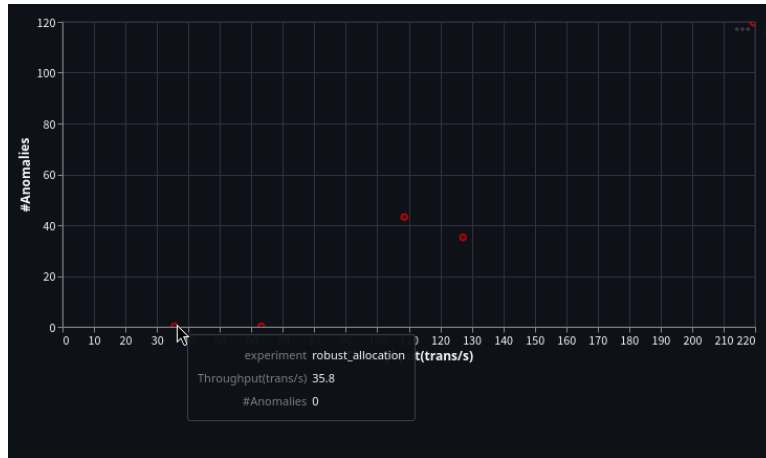
After examining these log files, it became clear that the high runtime was not caused by a transaction that keeps being aborted in the end. What did happen is that some transactions do abort often, more than 50 times. But this is not necessarily the last transaction that is run. The reason why these transactions have been aborted so many times is not yet clear. This does not happen every time the experiment is run. To get a better view of this, the experiments in section 5.2 have been done. Since they sometimes behave as expected, it is possible to show this result as shown in Figure 5.2. This gives the impression that when the programs behave as expected, the robust allocation is indeed faster than \mathcal{A}_{SSI} .

5.2 Follow-up experiments

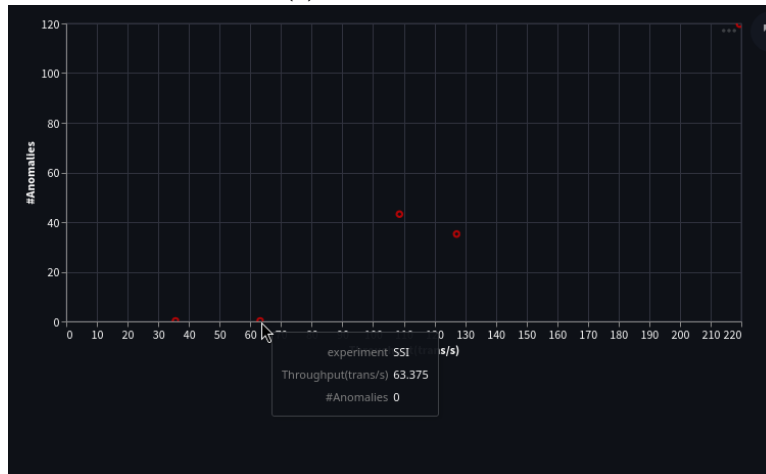
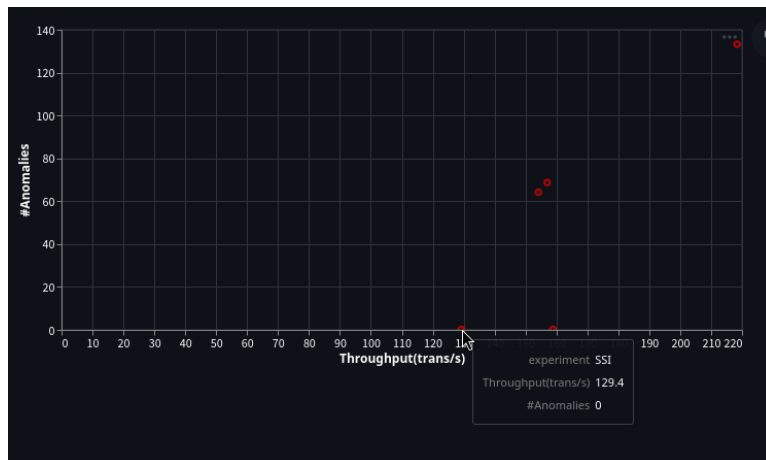
It is important to know when the runtime of a transaction is much higher than the runtime of other transactions. That is the purpose of these experiments. To map the contention parameters where this situation occurs.

5.2.1 Experimental setup

These experiments have not been done at the Flemish supercomputer center. Instead, they have been done locally on a laptop with a 4.463 GHz AMD Ryzen 7 5800H CPU with 8 cores, 16GB of RAM and 140GB SSD local disk. The PostgreSQL and Python versions are 16.2 and 3.11, respectively. The experiments focus on the microplus benchmark like the experiments that were previously done. The configurations that were done combine different values for some parameters. These parameters are the number of clients, the hotspot size, and the hotspot probability. The values used for the number of clients go from five clients up to 20 in steps of five. The values of the hotspot size go from five up to 25 in steps of five. As for the hotspot



(a) Robust allocation

(b) \mathcal{A}_{SSI} **Figure 5.1:** Unexpected result of the trade-off experiments**Figure 5.2:** Expected result of the trade-off experiments

	0	0	0	0	0	0	0
	0	0	0	0	1	1	
	⚠ changeA	⚠ changeAB	⚠ changeB	⚠ transferAB	⚠ changeA	⚠ changeAB	⚠ changeB
10_10_0-5RC	69.876407	61.142984	72.342532	97.074033	64.923139	71.834397	67.0611
10_10_0-5SI	62.500880	69.647613	58.203615	111.424083	62.069949	74.333245	67.0894
10_10_0-5SSI	74.612646	64.648849	61.465138	101.337705	75.303914	69.629120	61.9711
10_10_0-5robust	71.451231	75.740499	70.151846	84.110086	77.661257	66.760976	68.8695
10_10_0-6RC	69.120577	66.846460	66.119495	98.464637	65.655119	71.324422	63.3254
10_10_0-6SI	68.389839	69.856757	65.653100	98.200685	60.611107	59.869541	70.6074
10_10_0-6SSI	64.701659	69.177293	65.175890	102.129588	67.418486	63.302554	70.2463
10_10_0-6robust	72.018368	81.267266	70.665992	77.075434	76.614604	74.166595	61.8713

Figure 5.3: Table showing the runtimes of the experiments

82.015789	174.618606	67.672065	310.395197
-----------	------------	-----------	------------

Figure 5.4: Biggest difference in runtime

probability, it goes from 0.5 up to 0.9 in steps of 0.1. All combinations were run. That results in $4 \times 5 \times 5 = 100$ combinations. Then, each configuration has four different allocations since the reversed allocation was not rerun to save some time. This comes down to 400 experiments run.

5.2.2 Results

A visualization was added to analyze the results as shown in Figure 5.3. The table shows the superrun, run, and program in the first three rows. Each row has a unique name corresponding to the configuration. This name is the file's filename that contains the result concatenated with the isolation level. For each configuration, the highest runtime is marked for easy comparison. When this value is higher than usual, the configuration becomes interesting. Some interesting takeaways from these runtimes are that the differences in runtime seem to get higher when the amount of clients increases. This makes sense since the total amount of time transactions are being run increases. With 20 clients, the most significant difference was about 200 seconds, as shown in Figure 5.4. When plotting the trade-off of this configuration, we get the graph from Figure 5.5. So, even with the most significant difference of these 100 configurations, it still is not large enough to get an inversion between \mathcal{A}_{SSI} and the optimal robust allocation. Thus, the question remains when this exactly occurs and why.

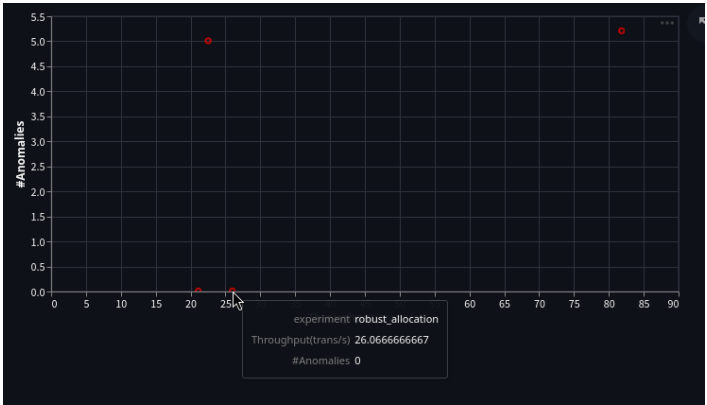


Figure 5.5: Trade-off visualization of the big runtime difference

Chapter 6

Conclusions

The purpose of the thesis is to explore the impact of robustness in the trade-off between throughput and anomaly minimization in database management systems. By developing a Python-based tool, we aimed to facilitate transaction throughput tests and to implement the robustness and allocation algorithm for mixed isolation levels.

Throughout the research, it became evident that robustness plays an essential role in enhancing transaction throughput while simultaneously minimizing anomalies. The trade-offs when using different isolation levels and the impact on both throughput and anomaly rates were widely tested and analyzed. The results show that stricter isolation levels like Serializable Snapshot Isolation (SSI) reduce the number of anomalies and the throughput. Less strict isolation levels like Read Committed (RC) increase the throughput heavily but also allow a lot of anomalies.

The throughput tests revealed that robustness, defined as ensuring serializability across all possible schedules for a given allocation, is crucial for balancing performance and data integrity. The tool not only facilitated these tests but also provided a variety of visualizations to compare different allocations, helping to identify the optimal balance for various scenarios. However, further experimentation is necessary to determine when specific programs take up almost all the runtime.

Moreover, the robustness and allocation algorithm's implementation and verification against known robust allocations proved successful, establishing a foundation for future research and practical applications. The work opens opportunities for further exploration into dynamic and adaptive allocation strategies that can respond to changes in workload and contention rates in real-time. Potentially changing the database management practices.

In conclusion, the thesis highlights the importance of robust allocation strategies in database management systems. This is done by balancing throughput and anomaly minimization through carefully managing the isolation. It is possible to optimize performance while maintaining the integrity of the data. The tools and implemented algorithm provide a strong foundation for further research in the field.

When implementing the tools, I learned how PostgreSQL does concurrency control and how this works; I had to create an in-depth understanding of the concepts to start implementing the tools. During the year, I learned how to manage my time. The first semester, I succeeded pretty well. The second semester, however, was a bigger challenge; the courses I had this semester were harder to combine and demanded more time. This, in combination with solicitations, did not make it any easier. I noticed that implementing the robustness and allocation algorithm did not interest me as much as implementing the other tools and the corresponding experiments, which I found to be more interesting, seeing how the different allocations influenced the throughput and the anomaly rate. I also learned how to manage a large project structurally, learned how to write a thesis, and gained a better understanding of my strengths and weaknesses.

Bibliography

- [FGA09] Alan Fekete, Shirley N Goldrei, and Jorge Pérez Asenjo. Quantifying isolation anomalies. *Proceedings of the VLDB Endowment*, 2(1):467–478, 2009.
- [VKKN21] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. Robustness against read committed for transaction templates. *arXiv preprint arXiv:2107.12239*, 2021.
- [VKN23] Brecht Vandevoort, Bas Ketsman, and Frank Neven. Allocating isolation levels to transactions in a multiversion setting. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 69–78, 2023.
- [VKN24] Brecht Vandevoort, Bas Ketsman, and Frank Neven. Allocating isolation levels to transactions in a multiversion setting. *SIGMOD Rec.*, 53(1):16–23, may 2024.
- [Wik23] Wikipedia contributors. Multiversion concurrency control — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Multiversion_concurrency_control&oldid=1187695173, 2023. [Online; accessed 1-April-2024].
- [Wik24a] Wikipedia contributors. Concurrency control — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Concurrency_control&oldid=1212090482, 2024. [Online; accessed 13-May-2024].
- [Wik24b] Wikipedia contributors. Zipf’s law — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Zipf%27s_law&oldid=1225573378, 2024. [Online; accessed 26-May-2024].

Appendix A

Example formats

A.1 Configuration file

```
{
  "concurrentClients": 50,
  "dbUrl": "localhost",
  "dbPort": 5432,
  "dbUsername": "postgres",
  "dbPassword": "postgres",
  "dbName": "postgres",
  "timing": {
    "warmup": 5,
    "experiment": 30,
    "extraTime": 5
  },
  "numberOfSuperruns": 2,
  "numberOfRuns": 1,
  "experimentName": "RC",
  "benchmark": "microplus",
  "microplus": {
    "numberOfRows": 200,
    "programChangeAIsolationLevel": "RC",
    "programChangeBIsolationLevel": "RC",
    "programChangeABIsolationLevel": "RC",
    "programTransferABIsolationLevel": "RC",
    "samplingMethod": "hotspot",
    "programChangeASamplingWeight": 0.1,
    "programChangeBSamplingWeight": 0.1,
    "programChangeABSamplingWeight": 0.1,
    "programTransferABSamplingWeight": 0.1,
    "sleepTimeAB": 100.0,
    "sdevsleepAB": 10.0,
    "sleepTimeBU": 100.0,
    "sdevsleepBU": 10.0,
    "hotspotSize": 20,
    "hotspotProbability": 0.5
  }
}
```

Listing 1: Example of a microplus configuration file

A.2 Results file

```

{
  "superruns": [
    {
      "runs": [
        {
          "completedTotal": 2860,
          "failed": {
            "deadlock": 0,
            "concurrentWrite": 1742,
            "dangerousStructure": 2420
          },
          "completed": {
            "RC": 0,
            "SI": 0,
            "SSI": 2860
          },
          "programs": {
            "changeB": {
              "failed": {
                "deadlock": 0,
                "concurrentWrite": 310,
                "dangerousStructure": 449
              },
              "completed": {
                "RC": 0,
                "SI": 0,
                "SSI": 724
              },
              "runtime": 293.92844343185425
            },
            "changeA": {
              "failed": {
                "deadlock": 0,
                "concurrentWrite": 489,
                "dangerousStructure": 784
              },
              "completed": {
                "RC": 0,
                "SI": 0,
                "SSI": 753
              },
              "runtime": 409.4000663757324
            },
            "changeAB": {
              "failed": {
                "deadlock": 0,
                "concurrentWrite": 536,
                "dangerousStructure": 874
              },
              "completed": {
                "RC": 0,
                "SI": 0,
                "SSI": 643
              }
            }
          }
        }
      ]
    }
  ]
}

```

```

    },
    "runtime": 415.00815749168396
  },
  "transferAB": {
    "failed": {
      "deadlock": 0,
      "concurrentWrite": 407,
      "dangerousStructure": 313
    },
    "completed": {
      "RC": 0,
      "SI": 0,
      "SSI": 740
    },
    "runtime": 400.0276367664337
  },
  "violationRate": 0.0
}
]
}
]
}

```

Listing 2: Result file of an experiment from the microplus benchmark

Appendix B

Configurations

B.1 Throughput experiment

```
{
  "concurrentClients": 16,
  "dbUrl": "localhost",
  "dbPort": 5432,
  "dbUsername": "bram",
  "dbPassword": "mapr-app-2",
  "dbName": "mapr-app-2",
  "timing": {
    "warmup": 5,
    "experiment": 20,
    "extraTime": 5
  },
  "numberOfSuperruns": 1,
  "numberOfRuns": 5,
  "experimentName": "RC",
  "benchmark": "smallBank",
  "smallBank": {
    "numberOfAccounts": 18000,
    "programDepositCheckingSamplingWeight": 1,
    "programBalanceSamplingWeight": 1,
    "programTransactSavingsSamplingWeight": 1,
    "programAmalgamateSamplingWeight": 1,
    "programWriteCheckSamplingWeight": 1,
    "accountSamplingMethod": "hotspot",
    "hotspotSize": 10,
    "hotspotProbability": 0.1,
    "programDepositCheckingAllocatedIsolationLevel": "RC",
    "programBalanceAllocatedIsolationLevel": "RC",
    "programTransactSavingsAllocatedIsolationLevel": "RC",
    "programAmalgamateAllocatedIsolationLevel": "RC",
    "programWriteCheckAllocatedIsolationLevel": "RC"
  }
}
```

Listing 3: Configuration for the throughput experiment

B.2 Anomaly experiment

```

{
  "concurrentClients": 10,
  "dbUrl": "localhost",
  "dbPort": 5432,
  "dbUsername": "bram",
  "dbPassword": "mapr-app-2",
  "dbName": "mapr-app-2",
  "timing": {
    "warmup": 1,
    "experiment": 30,
    "extraTime": 0
  },
  "numberOfSuperruns": 5,
  "numberOfRuns": 50,
  "experimentName": "RC",
  "benchmark": "micro",
  "micro": {
    "numberOfRows": 600,
    "samplingMethod": "hotspot",
    "hotspotSize": 500,
    "hotspotProbability": 0.9,
    "sleepTimeAB": 900.0,
    "sdevsleepAB": 120.0,
    "sleepTimeBU": 100.0,
    "sdevsleepBU": 30.0,
    "programChangeASamplingWeight": 0.2,
    "programChangeBSamplingWeight": 0.8,
    "programChangeABSamplingWeight": 0.0,
    "programChangeAIsolationLevel": "RC",
    "programChangeBIsolationLevel": "RC",
    "programChangeABIsolationLevel": "RC"
  }
}

```

Listing 4: Configuration for the inversion experiment