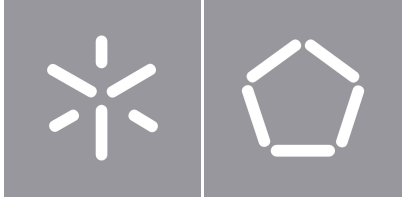
University of Minho
School of Engineering

Adriano Novo Soto Maior

Concurrent Data Structures

december 2023



University of Minho
School of Engineering

Adriano Novo Soto Maior

Concurrent Data Structures

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
José Orlando Roque Nascimento Pereira

december 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

I would like to express my sincere gratitude to some people who were important during this year and during my academic life.

I am deeply thankful to Professor José Orlando Roque Nascimento Pereira for guiding me and for all his availability to help throughout the entire research process. I also want to thank Nuno Faria for all the support, shared material and for his insight in the area.

I am grateful to INESC TEC for providing support and resources for this research. Their contribution has played a crucial role in the successful completion of this dissertation.

Finally, I would like to thank my family, especially my mother and my father for always being present when I needed, for giving me a place to grow and all the support I needed during these years. I would also like to extend my acknowledgements to my friends for their all their help, and for all the moments of companionship we shared together during these five years.

This work is co-financed by Component 5 - Capitalization and Business Innovation, integrated in the Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021 - 2026, within project ATE, with reference 56.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, december 2023

Adriano Novo Soto Maior

Abstract

The existence of items that are accessed and modified with high frequency in database management systems is a big obstacle to obtaining good performance. As such, it becomes more relevant to find methods allowing operations to execute efficiently and concurrently in order to increase application performance. Some concurrent approaches which can also deal with high concurrency, such as phase reconciliation [Narula et al., 2014] or CRDTs [Preguiça, 2018] already have a variety of data structures which can make use of these approaches. However, they are either not suitable for distributed systems or not designed for relational databases.

This project aims to create a library of data structures able to withstand high concurrency through partition and randomness, as proposed for MRVs [Faria and Pereira, 2023] and suitable for distributed systems, but offering a more diverse range of structures to support different applications. This proposal is then evaluated with an implementation in an SQL database, which shows that this method brings an increase in performance, especially when contention is higher.

Keywords SQL, MRVs, Concurrency, Data, Structures

Resumo

A existência de itens que são acedidos e modificados com grande frequência em sistemas de gestão de dados é um obstáculo significativo à obtenção de elevado desempenho. Assim, torna-se cada vez mais relevante encontrar métodos de lidar com a execução simultânea de operações que manipulam dados de forma eficiente, proporcionando às aplicações melhores tempos de resposta. No entanto, as abordagens existentes, como Phase Reconciliation [Narula et al., 2014] ou CRDTs [Preguiça, 2018], têm pressupostos estritos sobre a concorrência ou não foram desenhados para ser utilizadas em bases de dados relacionais.

Este trabalho tem como objetivo a construção de uma biblioteca de estruturas de dados que toleram elevada concorrência através da partição e aleatoriedade, tal como proposto para os MRVs [Faria and Pereira, 2023] e adequado a sistemas distribuídos, mas oferecendo uma gama de estruturas mais diversa, para suportar diferentes aplicações. Esta proposta é avaliada com uma implementação em sistemas de bases de dados SQL que demonstra um aumento de desempenho, particularmente quando existe maior contenção.

Palavras-chave SQL, MRVs, Concorrência, Dados, Estruturas

Contents

List of Figures

List of Tables

Acronyms

2PL Two Phase Locking.

CRDTs Conflict-free Replicated Data Types.

DBMS Database Management System.

MRVs Multi-Record Values.

OCC Optimistic Concurrency Control.

Chapter 1

Introduction

In 2015, out of the entire world population, 3.4 thousand million people [Roser et al., 2015] already had internet access and made regular usage of it, leading to many platforms being affected by high data access concurrency. Since these numbers have been increasing over time, finding methods to deal with this influx of users efficiently becomes more and more relevant, so that applications may find better response times.

Database Management System (**DBMS**) are essential for data intensive applications. However, these systems may need to sacrifice performance in order to guarantee consistency and data integrity when multiple users interact with the stored data. Since database systems are designed to store and manage large amounts of data, while providing efficient and reliable access to it, they are required to support concurrent access from multiple processes. However, concurrent access can lead to conflicts and data inconsistencies. So, in order to deal with conflicts, database systems use concurrency control techniques to guarantee that the data remains consistent and that the performance of the system is not significantly impacted by concurrent access.

There are some general purpose approaches which can be used for any data type and are commonly used in current database systems. On one hand, there is **Two Phase Locking** (**2PL**) [Bernstein et al., 1986] which relies on locking mechanisms and makes transactions block the data they need until they are completed. This method uses waits in order to deal with concurrency, by making concurrent transactions wait until the necessary locks are freed. In high contention scenarios, where hotspots are common, **2PL** will result in large waiting periods, since transactions will acquire locks for all the records that are used until they commit. On the other hand, there is **Optimistic Concurrency Control** (**OCC**) [Kung and Robinson, 1981], which lets multiple transactions interact with copies instead of the actual records. This method relies on aborts to deal with concurrency, by checking if the changes made by concurrent transactions interfere with each other during commit, and if so, it aborts the necessary transactions. In high contention scenarios, where hotspots are common, using OCC will lead to many aborts, since transactions will only notice each other's changes during validation, even if they are updating the same records.

There are also contention reduction techniques which increase availability, while still providing some data consistency guarantees. As such, different contention reduction techniques have been developed for databases, but also for different contexts. Escrow Locking [O'Neil, 1986] is a method which allows multiple transactions to make changes to a record simultaneously. This technique uses a journal to keep information needed to apply updates or perform rollbacks without compromising serializability. Escrow Locking can only be used with numeric values and requires access to all record journals to execute operations, which may not be possible in a distributed system.

Multi-Record Values (**MRVs**) [Faria and Pereira, 2023] are a technique that increases parallelism in SQL systems through randomness and data partition. This technique splits a value into many records in order to allow transactions to interact randomly with a part of the value when they do not need to make changes to the value as a whole. Since **MRVs** access records in a probabilistic manner, it does not require strong consistency in order to reduce data contention. In addition, this technique can be used without changing the **DBMS** source code, which is useful since many **DBMS** are not open-source. At the moment, **MRVs** has only been considered for integer values.

Phase Reconciliation [Narula et al., 2014] is a technique for in-memory transactions that allows splittable operations to run concurrently without causing conflicts. This technique has different phases, and splittable operations can only execute during one of them (Split Phase). In order to change phases, there is a moment where full system synchronization is necessary, which makes it unsuited for distributed systems where nodes might be unavailable due to network partitions. This technique is also not ideal for relational databases, since phase changes might be difficult to implement and only be possible by changing the source code of existing **DBMS**.

1.1 Problem

The existing contention reduction techniques, either were not planned to be used in relational databases or are lacking when it comes to the existing operations. As such, the questions are if the Phase Reconciliation operations can be implemented with **Multi-Record Values**, making use of randomness and partition to obtain performance gains. As well as, if the **Multi-Record Values** approach can be applied to new data structures, in order to evaluate if they are suitable for SQL databases in distributed systems without changing the **DBMS** source code.

1.2 Objectives

This work aims at the development of a library of data structures supporting high concurrency through partition and randomness and its performance evaluation specially in a distributed system environment using **Multi-Record Values** as a building block. This structures will be applied to relational databases using SQL with ACID structure. In addition, we also evaluate the data structures to be developed and compare them with current methods to execute the same operations.

1.3 Main Results

The work done in this dissertation resulted in a library of data structures using the MRVs technique that can be applied to SQL databases. It also led to a publication based on initial results:

- Adriano Maior, Nuno Faria and José Pereira. MRV*: Uma biblioteca de tipos de dados para aplicações concorrentes. INForum 2023

1.4 Document Structure

This document is divided in five chapters. The first chapter contains an introduction, the problem, the main results and this work's objectives. Chapter 2 describes techniques that are currently used in distributed systems in order to obtain higher levels of concurrency, with examples of usage in the context of transactional systems. Chapter 3 introduces the new data structures developed in this dissertation. Chapter 4 evaluates the performance of the new structures using both a microbenchmark and real-world like workload benchmarks. Finally, chapter 5 contains a conclusion about the work made and ideas for future work.

Chapter 2

Concurrency Control and Contention Reduction

Throughout this chapter, to compare the different methods explained, the same instruction will be used as an example for all of them. Considering the existence of an integer value v_x representing the stock level of product X, after selling 5 units of product X through transaction T1 and 10 units through transaction T2, v_x's value needs to be updated to reflect this change. Assuming X had 20 units in stock before these sales, v_x's initial value is 20.

2.1 Concurrency Control Methods

There are two main types of concurrency control techniques: pessimistic and optimistic. With pessimistic concurrency control, locks are used to manage data access, so that, at any given moment, only one process can access a specific item. This approach is effective in preventing conflicts, but it can lead to performance problems, since if many processes access the same data item, and locks can only be acquired by one at a time, the system will act as if it were sequential, thus not making use of parallelism and impacting performance. Optimistic concurrency control, on the other hand, allows multiple processes to access the same data simultaneously, but to guarantee that data remains consistent it needs to go through a process of conflict detection and reconciliation when they do occur. This approach is less restrictive and works well when conflicts are rare, however in scenarios where they are common, the conflict detection and resolution mechanisms will be performance bottlenecks.

2.1.1 Two Phase Locking

Locking is a mechanism commonly used to achieve database consistency. Data items have a lock associated with them, so whenever there is an attempt to write to or read data items, the corresponding lock needs to be acquired. Two transactions can read the same item concurrently without causing any type of conflict, as such read locks may be acquired by multiple transactions at the same time. However,

Lock Type	Read	Write
Read		X
Write	X	X

Table 1: Lock Type Conflicts

anytime one transaction acquires a read lock, no transaction can acquire a write lock until the read lock is released, since writing may change the value of an item and lead to inconsistencies. Write locks are mutually exclusive, and can only be held by one transaction at a time. Whenever a transaction wants to update an item, it needs to check if the item lock is available, if so, it may acquire the lock, otherwise it will need to wait until the lock is released. Table 1 represents conflicts between each lock type. Locking allows the database scheduler to guarantee serializability.

Two phase locking [Bernstein et al., 1986] is a method for controlling when each transaction may acquire each lock. Two phase locking as the name suggest has two phases. At first, the one where each transaction obtains locks for the data items it will access, known as the *growing phase*. The second one is the *shrinking phase* where transactions release locks.

Let us consider the following notation *rl[x]* for read lock on item x and *wl[x]* for write lock on item x. If a transaction T1 needs to read an item x in order to decide what to write to item y, value x cannot be changed until the transaction is finished. Considering the previous transaction and a new one T2 that changes the values of item x and item y, if they are both running at the same time and T1 acquires *rl[x]*, reads the value and unlocks, there is a chance that T2 acquires *wl[x]* and updates the value of x. If T2 acquires *wl[y]* before T1, and updates the value of y and then T1 rewrites the changes, we will not be able to determine which transaction happened first since T1 changed y after T2 but T2 changed the value of x after T1 reading it. To avoid this problem, locks must only be acquired during the first phase, no locks can be acquired after the first unlock. In some particular scenarios, there might be a chance of deadlocks, but this can be resolved by simply ordering the locks. As long as locks are always acquired following the same order, deadlocks will not arise. Two phase locking can be used when working with any data type.

Example

With two phase locking, assuming T1 already obtained v_x's write lock and T2 tries to acquire it, it will not be able to do it, and it will need to wait until it is released. With $v'_x s$ write lock acquired, T1 is the only transaction able to access its value and make changes, as such it will update v_x's value to 15 ($20 - 5$),

commit and release the lock. Since no transaction holds v_x's lock after T1 releasing, T2 will be able to acquire it. Afterwards, similarly T2 will update v_x's value to 5 $(15 - 10)$, commit and release the lock. Now, both transactions have finished and v_x's value holds X's current stock.

2.1.2 Optimistic Concurrency Control

Optimistic Concurrency Control (**OCC**) [Kung and Robinson, 1981] is based on the assumption that transactions will have conflicts rarely. When using an optimistic approach, reading values will never cause a loss of integrity, as such reads do not have restrictions. However, the same cannot be said about writes. Writes have many restrictions, as such it is required that a transaction consists of two to three phases. Before starting, each transaction needs to initialize a read set, a write set, a create set and a delete set. During the first phase, known as read phase, writes are done in local copies of the nodes to be modified. Secondly, during the validation phase, the changes made previously are checked to find possible losses of integrity. If no problems are found, the transaction may proceed to the third phase, the write phase, where the local copies are made global. If the validation fails, the transaction will be backed up and then start over. **OCC** can be used for any data type.

Read Phase

During the read phase, whenever a transaction tries to write to a given object, instead of changing the global object directly, a local copy will be made and stored in the local write set and all writes will result in changes to the copy. If the transaction tries to read a value, it will be added to the read set. If a new object is created, then it will be added to the create set. When the transaction completes, the local changes will be subjected to validation.

Validation Phase

To verify the correctness of transactions running concurrently, serial equivalence needs to be guaranteed. Considering a transaction, a function that receives a given value and returns another of the same type, a concurrent execution of transactions is considered correct if the final value of an object can be obtained from its initial value by applying some permutation of the transactions that executed. This can be done through functional composition. This criterion assumes that each transaction writes to a consistent data state and that when one transaction occurs, the previous one as already written its changes.

To verify if the serial equivalence is valid, transactions need to be given an integer identifier such that if T_i comes before T_j, then $i < j$ and one of the following three conditions must be verified:

1. T_i completes before T_j starts.
2. T_i's writes do not affect T_j's reads and T_i finishes writing before T_j starts.
3. T_i's writes do not affect either T_j's reads or writes.

If none of these conditions can be verified, the transaction will be rolled back and restarted later.

Write Phase

Whenever validation is completed successfully, the copies made in the read phase (the content of the write set) will simply replace the global values, this is done quickly because objects are referred by name and not physical address, as such all that needs to be done is to replace the pointers.

Example

Considering the same scenario as in the previous example, using **OCC**, first we need to initialize T1's and T2's sets. Since, both T1 and T2 want to make changes to v_x, they will both need to make a copy of v_x and add it to their read set and update their local copies. T1 will update its local v_x to 15 ($20 - 5$), while T2 will update its version to 10 ($20 - 10$). Assuming T1 completes its read phase first, it will get a smaller identifier than T2. During validation phase, since T1's ID is lower than T2's and T2 has already started before T1 completes, the first condition fails. The second condition also fails, since whenever T2 has already started and T1 has not finished writing. The third condition will fail as well, since T1 and T2 write to the same object. As such, T2 will need to roll back and be executed later with a need ID. Meanwhile, T1 may proceed to the write phase since it has the lower ID and there are no other transactions running with a lower ID than T1's. During the write phase, T1's will swap its write set local objects with the global storage ones. Afterwards, clean up is executed to delete the old objects and the transaction is committed.

2.2 Contention Reduction Methods

2.2.1 Escrow Locking

Escrow Locking [O'Neil, 1986] is a method which allows multiple transactions to make changes to a numeric record in certain circumstances. If a system uses this method, it will guarantee that any update accepted will be able to execute eventually in any order with any subset of updates that have already been accepted.

In Escrow Locking, an update is composed of two parts. The first one is a test of a condition that needs to be true when the update occurs, and the second is the update itself. Therefore, if the system accepts an update, it will guarantee that the condition is valid until the update is applied. This also means if a new update is accepted, it will not interfere with the truthfulness of other concurrent update conditions which had already been accepted. As such, with this method, all updates can be committed in any order without compromising serializability.

Whenever an update is accepted, an Escrow journal is created for this request. An Escrow journal is used to keep the information needed to perform a rollback or a recovery and is composed by the transaction ID, Escrow poll ID, request parameters (test criteria and change) for the record being updated. When a transaction commits or aborts, the associated Escrow journal will be applied as an update or discarded. In both cases, the Escrow journal will be deleted.

Any Escrow journaled record A has three values representing it at any time:

- **val(A)**: The value A will assume if all accepted updates of live transactions are applied without any aborts.
- **inf(A)**: The lowest value A might take through any combination of commits and aborts of live accepted transactions on this record.
- **sup(A)**: The highest value A might take through any combination of commits and aborts of live accepted transactions on this record.

Whenever an Escrow request is accepted, two of these values change. Let us assume an Escrow request wanting to reserve a quantity C of a record's value. Assuming C to be positive, if the transaction which requested C completes successfully, the lowest value A can take which is $inf(A)$ will be reduced accordingly, so we will have $inf(A) := inf(A) - C$. Likewise, the current expected value of A will also be reduced, so $val(A) := val(A) - C$. Meanwhile, the highest value A can take will not change since we requested a positive quantity, if C were negative, than $inf(A)$ would not change and $sup(A) := sup(A) - C$.

If the transaction is completed without using all of C, so that a quantity K is left after committing, this value would need to be put back on the corresponding record. So, $inf(A) := inf(A) + K$ and $val(A) := val(A) + K$. The used quantity U, would be removed from sup(A), $sup(A) := sup(A) - U$, since the highest value A could take would be less than it used to be by U units. If C were negative, the roles of $sup(A)$ and $inf(A)$ would be reversed. If no other transactions using record A are active, after committing $inf(A) = val(A) = sup(A)$.

Since the changes made when requesting take place before the transaction commits, there is also a

need to think about what would happen in case of a rollback. Assuming C to be positive one more time, all it needs to be done is revert the change, so $inf(A) := inf(A) + C$ and $val(A) := val(A) + C$. If C were negative, we would change $sup(A)$ instead of $inf(A)$, likewise $sup(A) := sup(A) + C$.

Example

Considering the same example, with Escrow Locking, there would be an Escrow journal for record v such that $val(v) = 20$, $inf(v) = 20$ and $sup(v) = 20$, assuming no transactions were accepted and not committed before T1 and T2. Assuming T1 was received first, and we want to reserve 5 units of X for use during the transaction, the first thing we need to check is if $v \geq 0$ is valid until and right after committing. Afterwards, the Escrow journal values need to be updated, such that $inf(v) := 20-5 = 15$, $val(v) := 20 - 5 = 15$. The value of $sup(v)$ is kept as it was, since the highest value v may hold does not change. When processing T2, 10 units of X need to be reserved, as such we also check the condition $v \geq 0$ which will hold true and update $inf(v) := 15 - 10 = 5$, $val(v) := 15 - 10 = 5$, while maintaining $sup(v)$'s value. Since the reserved quantity is completely used by both transactions, there is nothing to be put back to the record's value so, all it remains is to commit both transactions. Assuming no other transactions are executing on record v, $sup(v)$ will be updated to be equal to $20 - 5$ after T1 commits and $15 - 10$ after T2 commits. By the end, all Escrow journal values will be equal and hold 5 as their value.

2.2.2 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types (**CRDTs**) [Preguiça, 2018] are abstract data types designed to be replicated at multiple nodes that satisfy two main properties. First, any replica can be modified without coordinating with other replicas, which allows changes to be made without having to await confirmation. Second, any two replicas that receive the same set of operations, will eventually reach the same state, thus guaranteeing state convergence. This method uses multiple replicas to assure redundancy, due to this, **CRDTs** can also be used to execute concurrent operations in different replicas.

There are two main types of **CRDTs**, they can be either operation-based or state-based. Operation-based **CRDTs** propagate information about operations to other replicas, allowing them to execute them to their local state. Due to the possibility of lost messages, operations could be lost when sending them to other replicas, as such there is a need for a messaging algorithm that guarantees all replicas receive all operations. If not, some replicas will change their local state while others, who have not received the operation, will not, which would result in divergent replicas. It is also necessary that each operation is

received only once, since receiving the same operation multiple times will result in applying it multiple times.

State-based **CRDTs** propagate data by sending a replica state and merging it with the receiving replica current state, since these states might be received in any order (assuming multiple replicas are sending their states simultaneously), it is necessary that merging states will always reach the same final state no matter the order in each the states were received, as such the effects of merging states needs to be commutative. State-based **CRDTs** do not need to use a reliable messaging algorithm for propagation since even if a state that was sent is lost, no information will be lost because every replica has its state stored locally, and it may resend it later. Furthermore, sending the same state multiple times will not result in unwanted changes, since receiving an older state will not cause changes to the current state.

Bounded Counter CRDT

A Bounded Counter [Balegas et al., 2015] is a state-based **CRDTs** built on top of a key/value-store that maintains information for enforcing numeric invariants. Bounded Counters use reservations similar to those of Escrow Locking in which different replicas receive rights over a value of the counter, in other words, each replica can execute changes to the counter's value as long as they respect their assigned rights. If a counter has a value $n = 40$ and an invariant $n \geq 10$ then there are a total of 30 rights to execute decrement operations before the invariant is broken, for each unit decremented one right is consumed. While decrements consume rights, increments add new rights in this scenario, if the condition was $n \leq 50$ would swap their effects on rights. If there are multiple replicas, rights may be distributed among them in any way, in case of replicas not having enough rights to execute an operation, the operation will fail. Replicas may also transfer rights among them as they see fit, if an operation requires more rights than those available locally, replicas may communicate in order to acquire rights to execute the operation. To store information about used and available rights, replicas hold a matrix R where each entry $R[i][j]$ keeps the rights transferred from replica i to j, and a vector U which for each replica keeps track of how many rights have been used. This **CRDTs** can be used to deal with integer values, such as counters.

Example

Considering the same example, with a Bounded Counter **CRDTs**, assuming there are 2 replicas, R1 and R2, and that T1 runs on R1 and T2 runs on R2, if the v_x holds a value of 20 and the invariant is $v_x \geq 0$ there are a total of 20 rights. Assuming the following matrix R and vector U:

R	R_1	R_2	U
R_1	10	0	0
R_2	0	10	0

Figure 1: Example of Bounded Counter data

Since each replica holds 10 rights, and neither as used any of them, then both transactions may execute since for T1 $10 \geq 5$ and for T2 $10 \geq 10$. When T1 executes, $U[1]$ will be set to 5 without changing anything else. After T2, $U[2]$ will be updated to 10, and all the other values will remain the same. In order to get the current value of v_x we first need to sum all $R[i]$ columns values and subtract $U[i]$'s values. In this case, we have $10 + 0 - 5 = 5$ for row $R[1]$ and $0 + 10 - 10 = 0$ for row $R[2]$. Now, we just add this values in order to get v_x, so by the end $v_x = 5 + 0 = 5$.

2.2.3 Phase Reconciliation

Phase Reconciliation [Narula et al., 2014] is a concurrency control method for multicore databases divided in three phases. There are two phases which allow transactions to run, one of them allows any type of transaction, joined phase, while the other, split phase, allows some commutative operations to be executed in parallel for specific records by making local copies of such records and marking them as split. These records change during execution according to which operation type is most requested and to which records suffer from most contention. The last phase is called reconciliation phase and is responsible for joining records changed during split phase to the global storage. Phase Reconciliation can be used when working with numeric and set fields.

Joined Phase

During joined phase all kinds of operations can be executed, as such all transactions may be accepted. This phase may use any kind of concurrency control protocol, but, usually, **OCC** is used since it works better when there are few conflicts which is expected during this phase (since most conflicting operations are expected to run during split phase). Any transaction that starts its execution during joined phase must be completed during joined phase, since transactions do not transition between states, therefore before changing states all transactions started must be completed or aborted.

Split Phase

Split phase allows some operations that usually cause conflicts to be executed in parallel. Access to reconciled data is done as in the previous phase using **OCC**, however, split data executes on local copies of the records. During split phase not all operations may be executed, for each split record only one type of operation can be executed during a specific split phase, so this operation can only change between different split phases. If any transaction tries to execute an operation that is not the selected one, it must abort and be rescheduled. When transactions commit during split phase, split records changes can be applied directly to the local changes without using locks, since the local slices cannot be viewed by other processes. As in the joined phase, all split-phase transactions must commit or abort before transitioning to a new joined phase.

Reconciliation Phase

During the reconciliation phase, no new transactions are executed. This phase can be executed in parallel with split phase occurring in different processes and is responsible for merging the split records into their global storage versions. After all records are updated, they are marked as reconciled and a new joined phase begins.

Splittable Operations

Currently, the available operations for split phase are *Max*, *Min*, *Add*, *OPut* and *TopKInsert*. *Max(k,n)*,*Min(k,n)* and *Add(k,n)* are operations for integer values, where *Max* and *Min* replace k's value with n if n is higher/lower then k and otherwise they keep k's value, while *Add* sums n and k and stores the result in k.

The *OPut(k,o,x)* is an operation on ordered tuples, where each tuple, represented by k, is composed of *o* which is a number representing the order, j which is the ID of the core that wrote the tuple and x which is an arbitrary byte string. When *OPut(k,o',x')* is executed by core j', this operation will replace k's value with the new one if $o' \geq o$ or if $o' = o$ and $j' \geq j$. If k had not had a value, then it is assumed $o = -\infty$.

The *TopKInsert(k,o,x)* is an operation on top-K sets, which are like a bounded set of ordered tuples. Each one contains a maximum of K items, where each item is a tuple with the *(o,j,x)* structure referred above. If a core executes a *TopKInsert(k,o',x')*, the tuple *(o',j',x')* will be inserted in the relevant top-K set. For each order value there can only be at most one tuple, in case of duplicate order, the highest core ID will be used as a decider. If a top-K has more than K values, the lowest order value will be dropped.

Example

Considering the same example, with Phase Reconciliation, assuming we are currently in a Split Phase in order to subtract we need to use the Add operation with a negative value of n. During Split Phase, per-core slices are created and data is split through the slices, as such, assuming T1 executes on slice 1 and T2 executes on slice 2, no conflicts will arise, since they are accessing different records. As such, T1 may subtract 5 units from slice 1 without interfering with T2 removing 10 units from slice 2. When Split Phase is completed, all operations will be applied to the global storage during Reconciliation Phase.

2.2.4 Multi-Record Values

Multi-Record Values (**MRVs**) [Faria and Pereira, 2023] is a technique which allows higher parallelism on data access through the splitting of values. **MRVs** relies on splitting a value from a particular column in multiple records using an auxiliary table and intercepting operations executed on the original table, so they act on the new auxiliary table. During execution, values stored in multiple records need to be adjusted and balanced, therefore this method also uses background worker with this intent. Currently, this method is used for numeric fields.

Data Structure

To make use of **Multi-Record Values**, for each column that needs to be transformed a new table needs to be created with a 1-to-n relation from the original, with the original table primary keys as foreign keys for each corresponding record. The original value will then be divided by multiple records on the new table identified by a unique integer between [0,N-1], which works as reservations of part of the value. Therefore, the original value will no longer be stored in the original table, but will be calculated through the sum of all records on the new table with the corresponding foreign key.

Operations

MRVs allow read, write, bound (checks a lower bound), add and sub operations on an integer value. The key advantage of **MRVs** is that add, sub and bound operations cause no conflicts with each other, which means they can be combined or executed concurrently.

To read a value, the auxiliary table needs to be scanned and all records with the corresponding foreign key have to be summed. This operation will cause conflicts with all others updating the same value if not using Snapshot Isolation or something similar.

In order to write any value, all existing records on the new table will need to be deleted, and a new one has to be inserted with a random key between [0, N-1]. Since all records need to be interacted with, this operation will conflict with all others accessing the same value.

To perform the add operation of quantity *q* to a given value, a random key *rk'* in the range [0, N-1] needs to be picked, then one must look for the record with *rk* equal to *rk'* and add *q* to it. If there are no records with *rk* equal to *rk'* then the next record that comes immediately after *rk'* will be used.

When it comes to subtracting some δ from value v considering $v \geq 0$, a random key *rk'* needs to be picked and if this record's value *vi* is greater than δ, then *vi* just needs to be replaced by $vi - \delta$. Otherwise, *vi* is set to 0 and δ is set to the remainder of $\delta - vi$ which will be carried to the next record where the same validation will be executed until δ equals 0. If $\delta \geq v$ then the entire **MRVs** will be scanned and the transaction will roll back in the end.

The bound operation checks lower bounds for a given value. To execute this operation, records with the desired foreign key need to be visited until the desired quantity is met. This operation can be combined with sub, allowing subtractions only if a lower bound is met.

Example

Considering the same example, with **Multi-Record Values**, assuming v_x's is divided in 3 different records as shown in Figure 2. In order to perform a sub operation, each transaction will need to select a random key to decide which record to access. Assuming T1 starts processing first, and gets $rk = 2$, since $10 \geq 5$, we just need to do $10 - 5$ and update v_x's value on the row with $rk = 2$. When T2 is received assuming $rk = 0$, we have v_x's value equal to 5 on this row, however we want to subtract 10, as such we set v_x's to 0 and update our current value to subtract from 10 to 5 (since we already took five from the current record). Now, we proceed to the following record which is $rk = 1$, and we check if $v_x \geq 5$ which is true, so we set v_x to 0 and commit. To obtain the total value of v_x we just need to sum all rows of column v_x. If both transactions get the same random key and cause conflicts, the underlying database management system can deal with conflicts according to the selected isolation level.

v_x	f_key	rk
5	15	0
5	15	1
10	15	2

(a) Before the operations

v_x	f_key	rk
0	15	0
0	15	1
5	15	2

(b) After the operations

Figure 2: Example of **MRVs**

2.3 Summary

Table 2 associates each concurrency control method discussed earlier with the type of consistency it provides, the supported data types for each of them, and the parallelism technique used. Consistency is what guarantees that a transaction's change to the database will maintain data integrity. Different systems require different kinds of consistency, which can be either strong or weak.

Strong consistency ensures that all clients see the same data at the same time. This means that once a change is made to the database, it is immediately visible to all clients. Strong consistency is typically achieved through mechanisms which ensure that only one client can update the database at a time. This guarantees that the database remains consistent, but it can lead to higher contention and lower concurrency. There is also strong eventual consistency, which allows replicas to diverge temporarily, but data will eventually return to a consistent state.

Weak consistency, allows for more concurrent access to the database and enables clients to work with different versions of the data. This means that clients might not see the same data at the same time, and that changes made by one client might not be immediately visible to other clients. This can lead to increased performance and scalability, but it can also introduce additional complexity in managing data consistency. Weak consistency is not useful for what we are trying to achieve with this work, since we aim to expand the work done by Faria and Pereira [2023] with **Multi-Record Values** (**MRVs**).

Method	Consistency Type	Parallelism Technique	Supported Data Types
Two Phase Locking	Strong	Reservations	Any
OCC	Strong	Timestamping	Any
Escrow Locking	Strong	Reservations	Numeric Fields
Bounded Counter CRDT	Strong Eventual	Reservations and Commutativity	Numeric Fields
Phase Reconciliation	Strong	Reservations and Commutativity	Numeric and Set Fields
Multi-Record Values	Strong	Reservations and Commutativity	Numeric Fields
Multi-Record Values*	Strong	Reservations and Commutativity	Numeric and Set Fields

Table 2: Concurrency control and contention reduction methods

Chapter 3

Data Structures

In this chapter, we propose a structure that generalizes the Max, OPut and TopKInsert operations of Phase Reconciliation, the NTop-K structure, and a structure that works as a read and increment to a register, generating a different number any time an operation has been executed, the Serial structure.

3.1 NTop-K

The NTop-K structure stores the k highest/lowest elements that have been inserted and returns them sorted. This structure uses a value and an order column to represent a value's position in the Top-K. If k is 1, then the structure will return the maximum element of all inserted, providing the same features as the Max operation. This structure also allows other columns to be dependent on the values it stores. As such, if k is 1 and values have payloads, this structure can also accomplish the same as the OPut operation. If k is any other value, then this structure provides the same effects as the TopKInsert operation but allows elements to have payloads. Since the NTop-K structure can do the same as the individual operations proposed by Phase Reconciliation [Narula et al., 2014], we will also propose a custom implementation of each operation per comparison.

Implementing these operations without using any contention reduction technique in high contention scenarios can lead to conflicts. Conflicts happen when more than one transaction updates the same registers. Using the Max operation as an example, if we let all transactions run, they can result in incorrect values, since this operation requires reading the current value, comparing it with the new one and update it. With **2PL**, if multiple transactions try to update the same record's value, they will need to acquire the record's lock, which can only be held by one transaction at a time, resulting in transactions having to wait to access the data. Since in **2PL**, transactions will acquire locks for all the records that are used until committing, if transactions are long-lived, this will result in records being locked for long periods, which will make all other transactions who need to update any of those records having to wait until the locks are

released. Using **OCC**, if transactions are long-lived and conflicts are common, will lead to many aborts. Since transactions will only notice each other's changes during validation and as they are all updating the same record, they need to abort to avoid creating serialization errors. The NTop-K structure allows multiple transactions to interact with data concurrently by replicating records and making use of commutativity.

The Max structure allows keeping the maximum of all values inserted in this structure. To reduce contention, instead of using just one record to keep the maximum value, we instead use k records, where k can be any positive integer value. When a write operation in the table containing this structure is attempted, the operation is intercepted and is applied instead to one of the k records randomly.

The OPut structure keeps a record's most recent value according to an order number. Each record contains a value v and an order number o. Using MRVs*, this structure is similar to the previous one. There are k records with the value and the order number columns. To perform an update on the table, a random record is selected. If the new value has an order number higher than the one stored, the value and order number are updated to the new ones.

The Top-K structure stores the K highest/lowest elements that have been inserted and returns them sorted. As in the previous structures, instead of just using one record to store the Top-K, there are multiple available and inserts can affect either record.

3.1.1 Operations

NTop-K allows two operations, Add and Read. The Add operation adds a value to the NTop-K if it is higher than at least one of the current elements. The same can be done for the lowest K, comparing if the values we are inserting are lower than the ones stored. After a value has been added to the NTop-K structure, it cannot be removed. This operation can be executed concurrently with other Add operations.

The Read operation allows users to get the current NTop-K values. Using Snapshot Isolation, reads can be executed concurrently with other Reads and Adds. This may lead to some Adds having to roll back, since concurrent changes to the same data are not be seen by other transactions until committing.

3.1.2 Usage Example

Max

In an online auction with multiple users trying to bid on the same product at once it is necessary to allow users to bid concurrently, furthermore, the highest bidding has to be available for read operations. This structure is ideal for these scenarios, since it allows multiple users to bid at the same time while also

Procedure for U_1

1. U_1 selects rk'_1.
2. Select the record with random key rk_1.
3. If $2000 > v_1$, update v_1 to 2000.

Procedure for U_2

1. U_2 selects rk'_2.
2. Select the record with random key rk_2.
3. If $1900 > v_2$, update v_2 to 1900.

Figure 3: Max operation's procedure example.

keeping data available for reads.

Considering now an example of a situation where 2 users, U_1 and U_2, try to bid simultaneously on a vehicle that is being auctioned where the initial bidding starts at 1500 units of currency (u.c.). In addition, U_1 wants to make a bid of 2000 *u.c.* and U_2 wants to make a bid of 1900 *u.c.*.

If the structure is not being used and U_1 acquires the record's lock first, U_1 will read the current value, check that $2000 > 1500$, update the record's value and release the lock. Meanwhile, U_2 also attempts to acquire the lock intending to make a bid, however they will not be able to until U_1 releases, so they have to wait until they can to proceed. By then, U_2 realises that the current highest bid has changed and needs to attempt to make a higher bid. If we consider this scenario but with thousands of users trying to bid simultaneously in the same vehicle, transactions will have long wait periods before they can apply their changes.

Using the Max structure, for the same scenario, both U_1 and U_2 will receive a random key rk'_1 and rk'_2, respectively, and each of them will interact with records rk_1 with value v_1 and rk_2 with value v_2. This way, each user will acquire the lock for their own record and both will be able to make changes simultaneously without having to wait for each other to complete. If $2000 > v_1$ and $1900 > v_2$ both records will be updated, leading to $v_1 = 2000$ and $v_2 = 1900$. In order to read the value of the current highest bidder, one needs to go over all the records with the corresponding key and select the highest one. As such, we have two operations, add 2000 by U_1 and add 1900 by U_2.

Considering $rk_1 = 5$ and $rk_2 = 7$, Figure 4a shows an example table. In detail, since $rk_1 = 5$, we see in Figure 4a that $v_1 = 1500$, as $2000 > 1500$ we can simply replace the record's value to $v_1 = 2000$. For U_2, we select the record $rk_2 = 7$ with $v_2 = 1500$, since $1900 > 1500$ we update v_2 to 1900. Figure 4b shows the result of executing both operations, containing the updated v_1 and v_2 values. If a read operation is executed after Figure 4b, the returned value will be 2000 since it is the highest value on the table.

$value$	rk	f_key
1500	1	10
1500	5	10
1500	7	10
1500	14	10

(a) Before executing the operations.

$value$	rk	f_key
1500	1	10
2000	5	10
1900	7	10
1500	14	10

(b) After executing the operations.

Figure 4: Max example.

$order$	u_id	rk	f_key
1	8	0	15
2	5	1	15
9	7	2	15
7	12	3	15

(a) Before executing the operations.

$order$	u_id	rk	f_key
10	2	0	15
2	5	1	15
9	7	2	15
7	12	3	15

(b) After executing the operations.

Figure 5: OPut example.

OPut

Consider the case where the highest bid in an auction keeps getting updated. This structure can be used in scenarios where it is only necessary to store the most recent value according to some order.

Let us assume there are 2 processes, P_1 and P_2, trying concurrently to bid on the same item the order-value pairs $(10, 2)$ and $(5, 5)$, respectively, where the first component is the bidding amount is and the second the user's ID. The MRV* is initially composed of 4 records, as shown in Figure 5a. If P_1 receives the random key $rk' = 0$ and P_2 $rk' = 2$, both processes are able to proceed with the transaction concurrently without creating conflicts, since they will be modifying different records $rk = 0$ and $rk = 2$, respectively.

For P_1, as their order number is higher than the stored order for the chosen record ($10 > 1$), the record will be updated to $order = 10$ and $value = 2$. In P_2's case, they have an order number of 5, and they are attempting to update the record with $rk = 2$. Here, since the stored order number is higher than the one they are trying to insert, the write operation will fail and the record with $rk = 2$ will stay as it was. The final results are shown in Figure 5b. If a transaction attempts to read, the MRV* will be scanned and the final value will be $(10, 2)$, as it is the record with the highest order number.

$value$	f_key	rk
[1,2,7,12]	15	0
[1,4,9,14,17]	15	1

(a) Before executing the operations.

$value$	f_key	rk
[1,2,7,8,12]	15	0
[1,4,9,14,17]	15	1

(b) After executing T1.

$valor$	f_key	rk
[1,2,7,8,12]	15	0
[4,5,9,14,17]	15	1

(c) After T1 and T2.

$value$	f_key
[8,9,12,14,17]	15

(d) Reading the Top-K.

Figure 6: Top-K example.

$Top-K$
[1,2,7,12]

(a) MRV* Top-K version.

$value$	$order$
12	1
7	2
2	3
1	4

(b) MRV* NTop-K version.

Top-K

This structure can be used, for example, for storing the most sold products in an online store. Considering the example in Figure 9a where someone intends to keep the 5 most sold products. In detail, this MRV* has two records with sub-keys 0 and 1.

For the Top-K there are 2 options when it comes to the MRV* table. The same data can be stored either as in Figure 7a or as in Figure 7b. Figure 7a considers the Top-K as an array, and a value's position in the array represents its position on the Top-K. While Figure 7b considers the Top-K different records with an additional order column, which allows it to be used in databases that do not have an array data type.

Considering now three transactions, T_1, T_2 and T_3 that insert the values 8, 5 and 3, respectively. For the array version, T_1 first selects a random key $rk' = 0$, which will get the record $rk = 0$. To insert 8 in that record, the transaction will have to determine the position where it will insert the new value, which in this case is the one with index 4, since $8 > 7$ and $8 < 12$. Since the Top-K is not full, no value has to be removed, as such it is enough to shift 12 to the next position and insert the 8 where the 12 was. The final result is shown in Figure 9b. In detail, the array $[1, 2, 7, 12]$ was converted to $[1, 2, 7, \underline{8}, 12]$.

$value$	f_key	rk
1	15	0
2	15	1
7	15	3
12	15	9
1	15	13
4	15	14
9	15	15
14	15	29
17	15	33

(a) Before executing the operations.

$value$	f_key	rk
8	15	0
2	15	1
7	15	3
12	15	9
1	15	13
4	15	14
9	15	15
14	15	29
17	15	33

(b) After executing T1.

Figure 8: MRV* Top-K example.

T_2 selects the random key $rk' = 1$ and tries to insert the value 5. First, it will have to determine the position where it should insert, which is the one with index 2. As the Top-K is full, it is necessary to remove the value with the lowest index and shift all values lower than 5 one position down. Afterwards, it is enough to just insert the 5 in the position with index 2.

After T_2, T_3 selects the random key $rk' = 1$ and tries to insert value 3. This transaction will notice that this is not possible since the Top-K is full and the lowest value is higher than 3. As such, the transaction finishes without making changes to the table.

To read the Top-K the highest values from all MRV* records with the corresponding key are selected, which results in the final value shown in Figure 6d. In detail, the final top is composed of $[8, 9, 12, 14, 17]$. The stored values lower than 8 are excluded from the end result.

Considering now the NTop-K version and the same transactions, T_1, T_2 and T_3. Since there are only 4 elements higher than 8 in the NTop-K structure, we know we the new value has to be inserted in the structure. As such, T_1 selects a random key $rk' = 0$, which will get the record $rk = 0$ and as $8 > 1$ we can just replace the value. The final result is shown in Figure 9b.

T_2 tries to insert the value 5. First, it will have to determine if the value needs to be inserted. As the Top-K's lowest value is 8 and $5 < 8$ this value is not part of the Top-K and so it will not be inserted.

After T_2, T_3 tries to insert value 3. This transaction will notice that this is not possible since the NTop-K's lowest value is higher than 3. As such, the transaction also finishes without making changes to

$value$	f_key	rk
1	15	0
2	15	1
7	15	3
12	15	9
1	15	13
4	15	14
9	15	15
14	15	29
17	15	33

(a) Before executing the operations.

$value$	f_key	rk
8	15	0
2	15	1
7	15	3
12	15	9
1	15	13
4	15	14
9	15	15
14	15	29
17	15	33

(b) After executing T1.

Figure 9: MRV* Top-K example.

id	$order$	$value$
15	1	8
15	2	9
15	3	12
15	4	14
15	5	17

Figure 10: NTop-K view.

the table.

To read the NTop-K, the highest values from all MRV* records with the corresponding key are selected, which results in the final value shown in Figure 10. In detail, the final top is composed of $[8, 9, 12, 14, 17]$. The stored values lower than 8 are excluded from the end result.

3.1.3 Implementation

The Max operation was implemented as shown in Listing 1. This operation selects a random record to update (lines 2 to 14) and updates the value if the new one is higher than the one stored in the record (lines 16 to 19).

The OPut operation, implemented as in Listing 2, is similar to Max. First, it selects a random record

```
1  BEGIN
2      SELECT rk INTO rk_ FROM(
3          (SELECT rk
4              FROM {table}_{mrv.name}
5              WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
6              AND rk >= rk_
7              ORDER BY rk)
8              UNION ALL
9          (SELECT rk
10             FROM {table}_{mrv.name}
11             WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
12             AND rk < rk_
13             ORDER BY rk);
14     LIMIT 1;
15
16     UPDATE {table}_{mrv.name}
17     SET {mrv.name} = {mrv.name}_
18     WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
19     AND rk = rk_ AND {mrv.name}_ > {mrv.name};
20 END
21 $$ LANGUAGE plpgsql;
```

Listing 1: Max operation code.

to update (lines 2 to 14) and updates the value and order columns if the new order is higher than the one stored in the record (lines 15 to 18).

The Top-K operation is more complex than the previous ones. Listings 3 and 4 show a possible implementation of this operation. First, in Listing 3, the operation selects a random record (lines 1 to 13) and reads this record's array containing the Top-K and its size (lines 15 to 23). After that, in Listing 4 we find the insert position (lines 1 to 11) and if the array's size is less than K we can just insert in the right position by shifting the values higher than the new one forward (lines 13 to 23). If the size of the array is higher than K, we need to check if the loop ran at least one time, if so we insert in the new value in the right position by shifting the lower values backwards and the higher values forward and this way we also remove the value that is no longer part of the Top-K (lines 24 to 37).

To implement the NTop-K structure, a new table is created containing the primary keys from the original, a new column containing a random key, the value column and any other payload column necessary, as shown in Figure 12b. Each primary key, (pk_1, ..., pk_n, order), in the original table includes the order column, however when converting to the NTop-K structure this column is not needed since we can simply store the highest elements, sort them and limit the results to K when a Read occurs. For each K records on the original table, we need to create one record in the table shown in Figure 12a and at least K in the MRV* table in Figure 12b. As such, if we apply the NTop-K structure to the table in Figure 11, we obtain

```
BEGIN
    SELECT rk INTO rk_v FROM(
        (SELECT rk
            FROM {table}_{mrv.name}
            WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
            AND rk >= rk_
            ORDER BY rk)
            UNION ALL
        (SELECT rk
            FROM {table}_{mrv.name}
            WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
            AND rk < rk_
            ORDER BY rk);
    LIMIT 1;
    UPDATE {table}_{mrv.name}
    SET {mrv.name} = {mrv.name}_, {order} = order_
    WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
    AND rk = rk_v AND order_ > {order};
END
$$ LANGUAGE plpgsql;
```

Listing 2: OPut operation code.

pk_1	...	pk_n	$order$	$value$	$payload$	$other_1$	...	$other_n$

Figure 11: Original schema.

the two tables in Figure 12. The one in Figure 12a contains all records that are not MRV related, while the one in Figure 12b contains all MRV related records.

The Add operation, shown in Listing 5, adds a value to the Top-K by going over all the stored values and counting until all have been seen or until finding K higher values (lines 5 to 13). If K higher values are found, we know the that the one we are trying to add is not part of the Top-K and as such it will not be inserted. Otherwise, we select a random record (lines 16 to 23) out of the ones that are not part of the Top-K, and we replace it as shown between line 25 and 29.

To read the result, we need to merge the two MRV* generated tables, using Figure 13 as an example, for the final table we want the same records that were present in the original table. As such, we merge the table in Figure 13a with the table in Figure 13b using the ID as the merging column. Then, we order the values, add an order column accordingly and drop the random key column to originate the view in Figure 14.

Using custom indexes might also lead to performance gains, however they were not considered in this work. The only indexes used were primary key indexes. Existent indexes in the original table are also converted to work in the MRV* tables.

```
SELECT rk INTO rk_v FROM(
(SELECT rk
    FROM {table}_{mrv.name}
    WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
    AND rk >= rk_
    ORDER BY rk)
    UNION ALL
(SELECT rk
    FROM {table}_{mrv.name}
    WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
    AND rk < rk_
    ORDER BY rk)
LIMIT 1;

SELECT {mrv.name} INTO arr_topk
FROM {table}_{mrv.name}
WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
AND rk = rk_v;
SELECT array_length(arr_topk,1) INTO size;

IF size IS NULL THEN
    size := 0;
END IF;
```

Listing 3: Top-K operation code.

pk_1	...	pk_n	$other_1$	...	$other_n$

(a) Original table without the order, value and payload columns.

pk_1	...	pk_n	rk	$value$	$payload$

(b) MRV* table.

Figure 12: Modified schema with MRV* NTop-K structure.

3.2 Serial

The Serial structure allows users to obtain a serial number. Each time the next operation is executed, a new value is returned, as the same value cannot be returned more than once. The Serial structure guarantees that when a read occurs, all values lower than the returned value have already been used. Using the Next operation will always return a value that is either equal or higher than the one returned by a prior read operation.

If implemented directly without the structure, the Serial operation would be composed of two steps. First, reading the current value and, then, increment it. This is a small operation and can be executed quickly. However, in the context of long-lived transactions, using **2PL**, this leads to transactions holding the record's lock for long periods, which will not allow other transactions to access the serial value. Using **OCC**, transactions are able to run concurrently and access the serial value without waiting. However,

id
1

(a) Original table without the order, value and payload columns.

id	rk	$value$	$payload$
1	1	1	"payload_1"
1	3	6	"payload_6"
1	4	4	"payload_4"
1	8	2	"payload_2"
1	9	3	"payload_3"

(b) MRV* table.

Figure 13: Merge example tables.

id	$order$	$value$	$payload$
1	1	6	"payload_6"
1	2	4	"payload_4"
1	3	3	"payload_3"
1	4	2	"payload_2"
1	5	1	"payload_1"

Figure 14: NTop-K view.

since the record is updated to increment the serial value, and this is only noticed by other transactions during validation, all transactions read the same value. During validation, all transactions updating the same records abort to avoid inconsistent data. The Serial structure allows multiple transactions to interact with data concurrently by replicating records.

Some databases have auto-increment data types, which provide a similar functionality to the Serial structure. In PostgreSQL, this is the serial data type and it is implemented using sequences. The problem with this data type is that gaps may appear in the column, even if no rows are ever deleted [PostgreSQL, 2023]. This happens since a value from the sequence is considered used even if the transaction rolls back. Using the MRV* Serial structure, gaps exist, but are not permanent and will be filled throughout execution.

3.2.1 Operations

This structure allows 2 operations, Next and Read. Next is responsible for providing the next serial value to the user in a strictly ascending order without gaps. However, in practice, this not always possible since

f_key	rk	$serial$	$valid$
1	8	12	true
1	5	10	true
1	7	14	true
1	12	15	true

(a) Before executing the operations.

f_key	rk	$serial$	$valid$
1	8	12	true
1	5	10	false
1	7	14	false
1	12	15	true

(b) After executing the operations.

f_key	rk	$serial$	$valid$
1	8	12	true
1	5	16	true
1	7	17	true
1	12	15	true

(c) After running the Refresh Worker.

Figure 15: MRV* Serial example.

we have multiple serial records. When this operation is used, 1 of the k records is selected, and its valid column is changed to false, marking it as used. Read allows users to obtain the minimum serial value that has not been used thus far.

3.2.2 Usage Example

The Serial structure can be used anytime an identifier is needed, the structure can guarantee users unique values. The structure can also be used any time, a queue of events is necessary, it will not guarantee that each event will get its value in a first come, first served manner and events happening later may get lower values when using the Next Operation, however it will be able to assign each event a position. In a real scenario, this structure can also be used for example to generate invoice numbers, since it will produce a unique value for each Next Operation.

Assuming there are 2 processes, P_1 and P_2, trying concurrently get the serial value and the MRV* is initially composed of 4 records as shown in Figure 15a. If P_1 receives the random key $rk' = 5$ and P_2 $rk' = 6$, both processes are able to proceed with the transaction concurrently without creating conflicts, since they will be modifying different records $rk = 5$ and $rk = 7$, respectively.

After executing both transactions the MRV* will be as shown in Figure 15b, since the Next Operation only reads the serial value and updates the value column, the only difference between before and after

pk_1	...	pk_n	$value$	$other_1$	...	$other_n$

Figure 16: Original table.

pk_1	...	pk_n	$other_1$	...	$other_n$

(a) Original table without the value column.

pk_1	...	pk_n	rk	$value$	$valid$

(b) MRV* table.

Figure 17: After changing to MRV* Serial structure.

the transactions will be seen in the $valid$ column.

In order to update the values, we make use of a worker. After running the worker, the $valid$ column is changed to $true$ as shown in Figure 15c and the serial values of those columns are updated.

3.2.3 Implementation

To implement this structure a new table is created containing the primary keys from the original, a random key, a new boolean column and a value column. Each serial record requires an integer and a boolean column. The integer column represents the serial value, while the boolean column is used to check if the value has been used or not. For each record in the original table, k records are created in the new table. As such, the original table contains the primary key records, the serial value and any number of other columns, as can be seen in Figure 16.

After applying the MRV* Serial structure to the database, two tables are created. The first one, as shown in Figure 17a, is similar to the original table, but the serial value is removed. The second one contains the primary keys, the new random key column, the serial value and the valid column as shown in Figure 17b.

In order to implement the Next operation efficiently, we had to relax the gap component of this operation, which means that it is very likely that gaps may occur under high contention. However, if two transactions get serial values that are close to each other, we know that either both transactions are concurrent or they ran close to each other timewise. When contention is higher, the structure will have more nodes in order to deal with more transactions, however this leads to more values in the structure which can result in two transaction receiving values that are further apart than desired. Under high contention, records are more likely to be picked, which means that even if gaps appear, values should not be too far apart. Under low contention, the number of nodes will be lowered, and older values will get picked since there will not be many records to choose from.

As such, we suggest two ways of implementing the Next operation. The first one, shown in Listing 6, considers all records in the MRV* table and uses the ring property of **MRVs** to lookup records (lines 11 to 22 and 27 to 32), while the second one, shown in Listing 7, only considers records with the $valid = true$ (lines 11 to 22 and 24 to 40) and only considers the first record it finds. In the first one (Next A), first we select a record, if the selected record's valid column is already false, the operation will proceed to the next record (lines 27 to 32) until it founds one with $valid = true$ or until there are no more records. If no record is found with $valid = true$ the operation will return an empty table.

The other way of implementing Next (Next B) is not considering records with valid as false (lines 11 to 22). This version selects a random record, but only if valid is true. If no record is found with $valid = true$ the operation will still return an empty table.

To implement the read operation as shown in Listing 8, we need to find the lowest serial value. To do so, we can just select the record containing the lowest value with $valid = true$. If there are no records with $valid = true$ we select the record with the highest value and add 1, since that will be the new lowest value as soon as the worker runs (line 6).

For updating the serials values, we use a background worker. Refreshes are made through multiple transactions, each transaction only affects one record, thus others are still available. A refresh consists of collecting all the primary keys from the original table and for each one, read the serial value, increment it and update the values in all records with $valid = false$. The worker runs in fixed intervals that can be changed in a configuration file. At the moment, only 1 refresh worker is available at a time. Listing 9 shows an example of how the code can be implemented in Java. The worker runs the following instructions in a cycle:

1) Read all primary keys from a view containing all the keys (lines 4 and 5).

2) Read the number of columns that are part of the primary key (lines 6 and 7).

3) Append all primary key columns separated by commas (lines 9 to 13).

4) Execute the refresh operation for the primary key and commit (lines 14 to 16).

5) Repeat **3)** and **4)** for all primary keys.

Once again, for the serial structure, indexes were not used, but they may result in performance gains. As in the previous structure, existent indexes in the original table are also converted to work in the MRV* tables.

```
IF size > 0 THEN
    cur_value := arr_topk[loop_i];
    WHILE loop_i < size AND {mrv.name}_ > cur_value
    LOOP
        loop_i := loop_i + 1;
        cur_value := arr_topk[loop_i];
    END LOOP;
    IF {mrv.name}_ > cur_value THEN
        loop_i := loop_i + 1;
    END IF;
END IF;

IF size < {k} THEN
    FOR loop_j IN REVERSE size..loop_i LOOP
        UPDATE {table}_{mrv.name}
        SET {mrv.name}[loop_j + 1] = {mrv.name}[loop_j]
        WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
        AND rk = rk_v;
    END LOOP;
    UPDATE {table}_{mrv.name}
    SET {mrv.name}[loop_i] = {mrv.name}_
    WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
    AND rk = rk_v;
ELSE
    IF loop_i > 1 THEN
        FOR loop_j IN 1..loop_i-2 LOOP
            UPDATE {table}_{mrv.name}
            SET {mrv.name}[loop_j] = {mrv.name}[loop_j + 1]
            WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
            AND rk = rk_v;
        END LOOP;

        UPDATE {table}_{mrv.name}
        SET {mrv.name}[loop_i-1] = {mrv.name}_
        WHERE {' AND '.join([f'{pk.name} = {pk.name}_' for pk in data['pk']])}
        AND rk = rk_v;
    END IF;
END IF;
```

Listing 4: Top-K operation code.

```
BEGIN
    count := 0;
    OPEN cursor_rk;
    WHILE NOT done AND count < {k}
    LOOP
        FETCH cursor_rk INTO cur_rk, cur_{mrv.name};
        IF NOT FOUND THEN
            done := TRUE;
        ELSEIF cur_{mrv.name} >= {mrv.name}_ THEN
            count := count + 1;
        END IF;
    END LOOP;
    CLOSE cursor_rk;
    IF done THEN
        SELECT rk INTO ins_rk FROM (
            SELECT rk, {mrv.name} FROM {table}_{mrv.name}
            WHERE {' AND '.join([f'{pk.name} = {pk.name}_'
            for pk in data['pk']])}
            ORDER BY {mrv.name} DESC
            OFFSET {k-1} ROWS) AS T
        ORDER BY RANDOM()
        LIMIT 1;

        UPDATE {table}_{mrv.name}
        SET {mrv.name} = {mrv.name}_
        WHERE {' AND '.join([f'{pk.name} = {pk.name}_'
        for pk in data['pk']])}
        AND rk = ins_rk;
    END IF;
END
$$ LANGUAGE plpgsql;
```

Listing 5: NTop-K Add operation code.

```
cursor.execute(f'''
    CREATE OR REPLACE FUNCTION {table}_{mrv.name}(
    {columns_str(data['pk'], name_suffix='_', with_types=True)})
    RETURNS TABLE (
        {','.join([f'{x.name} {x.type}' for x in data['pk']])},
        {','.join([f'{x.name} {x.type}' for x in data['mrv']])}
    )
    AS $$
    DECLARE rk_ int = FLOOR(RANDOM() * ({model['maxNodes']} + 1))::integer;
            node_rk int;
            done bool = False;
            valid_ bool = False;
            cur CURSOR FOR
                (SELECT rk, valid
                FROM {table}_{mrv.name} AS T
                WHERE {' AND '.join([f'T.{pk.name} = {pk.name}_'
                for pk in data['pk']])} AND rk >= rk_
                ORDER BY rk)
                UNION ALL
                (SELECT rk, valid
                FROM {table}_{mrv.name} AS T
                WHERE {' AND '.join([f'T.{pk.name} = {pk.name}_'
                for pk in data['pk']])} AND rk < rk_
                ORDER BY rk);
    BEGIN
        OPEN cur;
        WHILE NOT done AND NOT valid_ LOOP
            FETCH cur INTO node_rk, valid_;
            IF NOT FOUND THEN
                done = TRUE;
            END IF;
        END LOOP;
        CLOSE cur;
        IF valid_ THEN
            UPDATE {table}_{mrv.name} AS T
            SET valid = FALSE
            WHERE {' AND '.join([f'T.{pk.name} = {pk.name}_'
            for pk in data['pk']])} AND rk = node_rk;
            RETURN QUERY
            SELECT {','.join([f'T.{x.name}' for x in data['pk']])},
            {','.join([f'T.{x.name}' for x in data['mrv']])}
            FROM {table}_{mrv.name} AS T
            WHERE {' AND '.join([f'T.{pk.name} = {pk.name}_' for pk in data['pk']])}
            AND rk = node_rk;
        ELSE
            RETURN;
        END IF;
    END
    $$ LANGUAGE plpgsql;
    ''')
```

Listing 6: Next code (alternative A).

```
cursor.execute(f'''
    CREATE OR REPLACE FUNCTION {table}_{mrv.name}(
    {columns_str(data['pk'], name_suffix='_', with_types=True)})
    RETURNS TABLE (
        {','.join([f'{x.name} {x.type}' for x in data['pk']])},
        {','.join([f'{x.name} {x.type}' for x in data['mrv']])}
    )
    AS $$
    DECLARE rk_ int = FLOOR(RANDOM() * ({model['maxNodes']} + 1))::integer;
            node_rk int;
            cur CURSOR FOR
                (SELECT rk
                FROM {table}_{mrv.name} AS T
                WHERE {' AND '.join([f'T.{pk.name} = {pk.name}_'
                for pk in data['pk']])} AND rk >= rk_ AND valid = true
                ORDER BY rk)
                UNION ALL
                (SELECT rk
                FROM {table}_{mrv.name} AS T
                WHERE {' AND '.join([f'T.{pk.name} = {pk.name}_'
                for pk in data['pk']])} AND rk < rk_ AND valid = true
                ORDER BY rk);
    BEGIN
        OPEN cur;
        FETCH cur INTO node_rk;
        IF NOT FOUND THEN
            RETURN;
        ELSE
            UPDATE {table}_{mrv.name} AS T
            SET valid = FALSE
            WHERE {' AND '.join([f'T.{pk.name} = {pk.name}_'
            for pk in data['pk']])} AND rk = node_rk;
            RETURN QUERY
            SELECT {','.join([f'T.{x.name}' for x in data['pk']])},
            {','.join([f'T.{x.name}' for x in data['mrv']])}
            FROM {table}_{mrv.name} AS T
            WHERE {' AND '.join([f'T.{pk.name} = {pk.name}_' for pk in data['pk']])}
            AND rk = node_rk;
        END IF;
        CLOSE cur;
    END
    $$ LANGUAGE plpgsql;
    ''')
```

Listing 7: Next code (alternative B).

```
serial_value = f'''(
    SELECT {','.join([f'K.{pk.name}' for pk in data['pk']])},
    MIN(K.max) as {mrv.name}
    FROM
        (SELECT {','.join([f'{pk.name}' for pk in data['pk']])}, valid,
        CASE WHEN valid THEN MIN({mrv.name}) ELSE MAX({mrv.name}) + 1 END
        FROM {table}_{mrv.name}
        GROUP BY {','.join([f'{pk.name}' for pk in data['pk']])}, valid) K
    GROUP BY {','.join([f'K.{pk.name}' for pk in data['pk']])})
'''
```

Listing 8: Serial read code.

```
while (true) {
    for (String table_name: config.refreshTables){
        try {
            getPk = connection.prepareStatement(
                "SELECT * FROM " + table_name + "_pk"
            );
            pkSet = getPk.executeQuery();
            pkSetMetaData = pkSet.getMetaData();
            pkSetLength = pkSetMetaData.getColumnCount();
            while (pkSet.next()) {
                sb = new StringBuilder();
                for (int i = 1; i <= pkSetLength; i++) {
                    if (i > 1) sb.append(",  ");
                    sb.append(pkSet.getString(i));
                }
                refresh = connection.prepareCall(
                    "SELECT refresh_" + table_name + "(" + sb.toString() + ")"
                );
                refresh.execute();
                connection.commit();
            }
        }
        catch (Exception e) {
            connection.rollback();
            e.printStackTrace();
        }
    }
    Thread.sleep(config.refreshDelta);
}
```

Listing 9: Refresh Worker base code.

Chapter 4

Evaluation

This chapter evaluates the performance of the implemented structures using different benchmarks. Benchmarks are tools that apply workloads to a database and provide metrics that can be used in order to evaluate a **Database Management System** (**DBMS**) performance.

As such, we verify if the NTop-K structure can achieve the same performance gains as the Phase Reconciliation operations while using the MRV technique and also evaluate the performance of the Serial structure when compared to using value increments.

4.1 Test Environment

All tests in this section were conducted on virtual machines in Google Cloud Compute Engine, with PostgreSQL as the database management system where the MRVs* were implemented. We used the *REPEATABLE_READ* isolation level, which in PostgreSQL results in Snapshot Isolation. With this isolation level, write operations first acquire an exclusive lock on the row to be modified. In concurrent writes to the same row, the first write to acquire the lock may succeed, while the others will need to be repeated. Reads take a snapshot of the database when the transaction starts, as such they will only find target rows that were committed as of the transaction start time and can execute concurrently without causing conflicts. Details of the hardware and software used can be found in Table 3. We considered PostgreSQL 15.1 as the default version with no modifications.

The NTop-K and Serial structures were evaluated using different benchmarks. For both structures, simple microbenchmarks were created to have an indication of how the structures behaved in a controlled environment. For the NTop-K structure, a benchmark consisting of a bet transaction was also used to evaluate performance in a more lifelike scenario. This benchmark was inspired on the AuctionMark [2023] benchmark's $newbid$ transaction but has some differences, since the way it was implemented did not allow the structure to be used directly. For the Serial structure, the remaining tests used the TPC-C [TPC]

CPU	RAM	Storage	OS	Database
8 vCPU (E2)	16GB	125GB SSD	Ubuntu 22.04 LTS	PostgreSQL 15.1

Table 3: Test machine specifications.

benchmark with the Benchbase framework [Difallah et al., 2013] to evaluate performance.

4.2 NTop-K

4.2.1 Microbenchmark

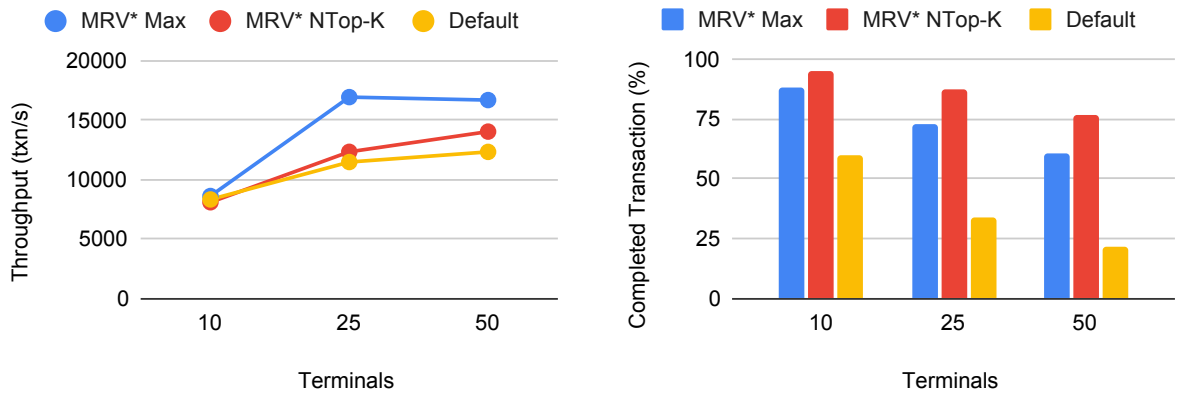
The tests performed using the microbenchmark ran 5 times with 10, 25, and 50 terminals executing operations on 100 records. In cases where the composition of the workload is not explicitly indicated, 65% corresponds to write operations and 35% to read operations, in order to reproduce the target scenario with contention. Each test ran for 90 seconds.

Max

With 50 terminals and a workload of 65% write and 35% read operations, the MRV* Max dedicated structure obtained the best performance, with a throughput 1.35× higher than the standard version and 1.19× higher than the NTop-K structure. This increase in throughput compared to the default operation is a consequence of the higher collision probability in writes, shown in Figure 18b. Specifically, in the standard version with 10 terminals, only 60% of transactions complete successfully, compared to about 90% when MRV* are used. As concurrency increases, the number of rejections also increases. Therefore, with 50 terminals in the standard version, only 21% of transactions complete. On the other hand, in the MRV* version, more than 60% of transactions are executed.

When comparing the MRV* Max and NTop-K structures, we see in Figure 18a that the dedicated structure is better with all levels of concurrency tested. This is the result of the Max structure having a more straight forward approach, since it is designed specifically for this operation. The NTop-K due to being more generalist is designed to support multiple structures, which results in an added performance cost.

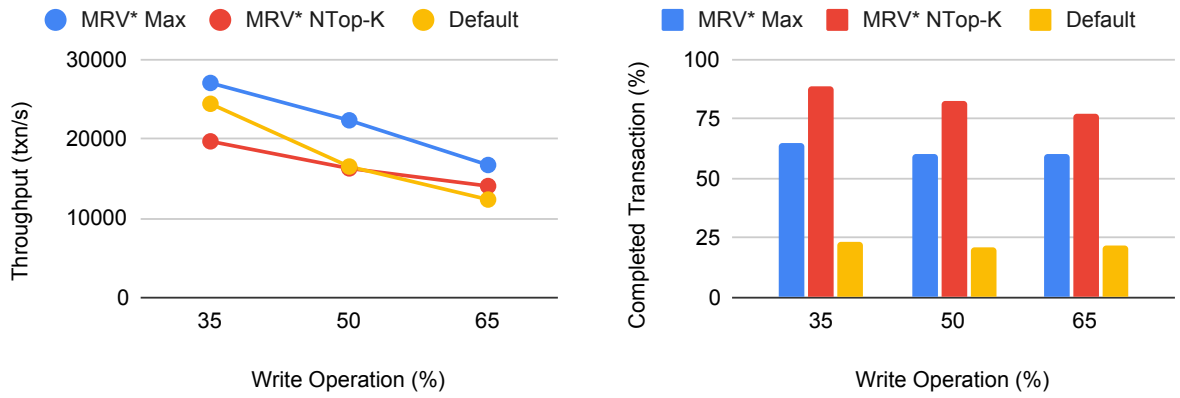
For workloads with more reads, one would expect the performance of the structure to be worse. However, the structures continue to perform well in these scenarios. In fact, the performance with the MRV* Max structure is 1.1× better than the standard version in scenarios with 35% writes and 65% reads



(a) Throughput Comparison.

(b) Rejected Transaction Comparison.

Figure 18: Performance comparison of Max operation with and without MRV*.



(a) Throughput Comparison.

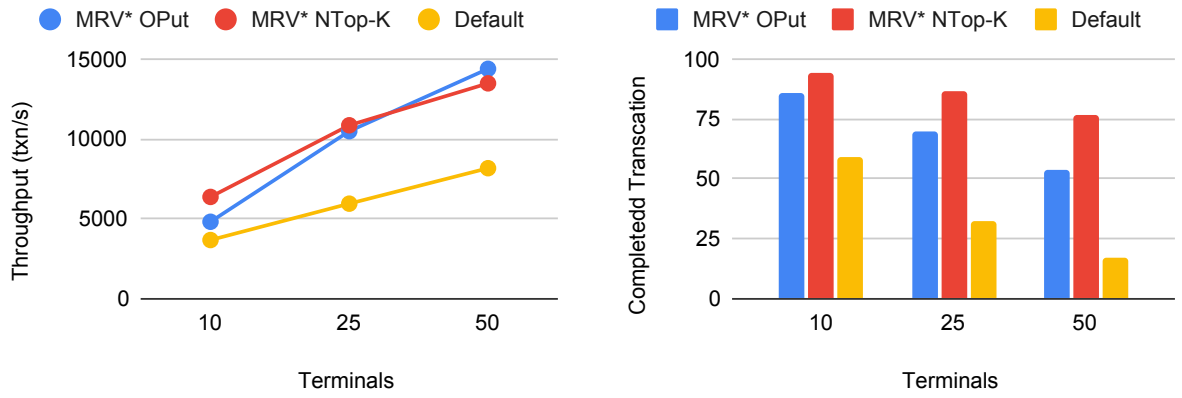
(b) Rejected Transaction Comparison.

Figure 19: Performance comparison of Max operation with and without MRV* for different loads.

with 50 terminals, as we can see in Figure 19a. This increase in throughput is, yet again, the consequence of the decrease in rejected transactions compared to the default version. Once again, when compared to the NTop-K the results are better using the specific structure. As before, this structure has a more costly algorithm in order to support other structures, which results in lower throughput, especially when the amount of write operations is reduced. The NTop-K structure can be used with workloads that have a higher percentage of write operations to improve performance compared to the default version. However, in general, the specific MRV* Max structure is the one showing the best performance across all tests.

OPut

For the OPut operation, whose results are visible in Figure 20a, both MRV* versions exhibit higher throughput than the standard version across all tested scenarios, a result of the reduced collision rate (Figure 20b).



(a) OPut throughput comparison.

(b) OPut rejected transaction comparison.

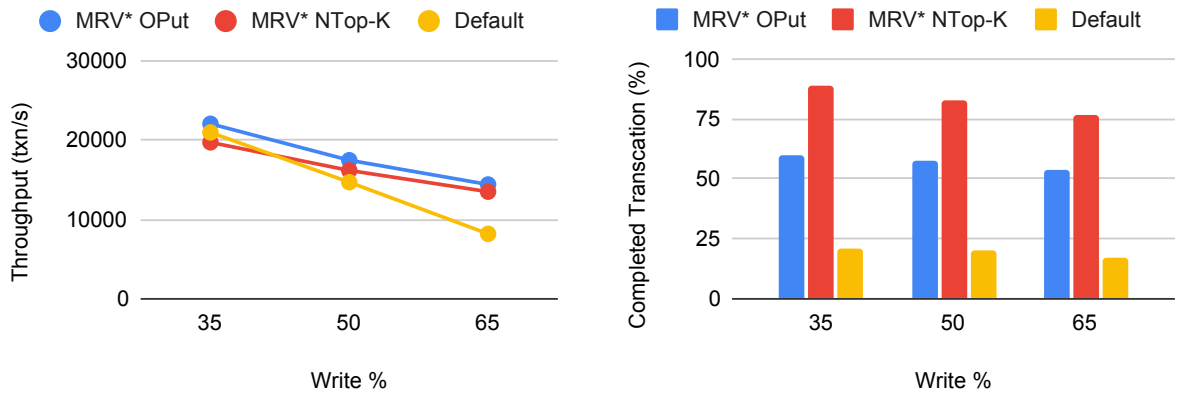
Figure 20: OPut performance comparison for MRV* and non-MRV* with different number of terminals.

When comparing the NTop-K and OPut structures, with a workload of 65% write and 35% read operations, for 10 and 25 terminals the NTop-K structure has better throughput, however once the number of terminals reaches 50 the OPut dedicated structure has higher throughput achieving 14404 transactions per second compared to the 13492 transactions per second obtained with the generalist structure. For 50 terminals, the MRV* OPut version is 1.76× superior to the standard version and 1.07× superior to the NTop-K.

Considering a scenario with a higher volume of read operations, Figure 21b and Figure 21a show that performance is superior to the default version, when using the MRV* NTop-K structure in workloads with a 50% and 65% write percentage. Not only is throughput higher, but we can also notice that the amount of completed transactions is significantly higher with this structure. When compared to the specific OPut structure, the NTop-K generalist structure has lower throughput, but performance is similar to the specific structure. As such, for the OPut operation, both the specific structure and the NTop-K show improvements in performance when compared to the default version and either one of them can be used according to the specific scenario.

Top-K

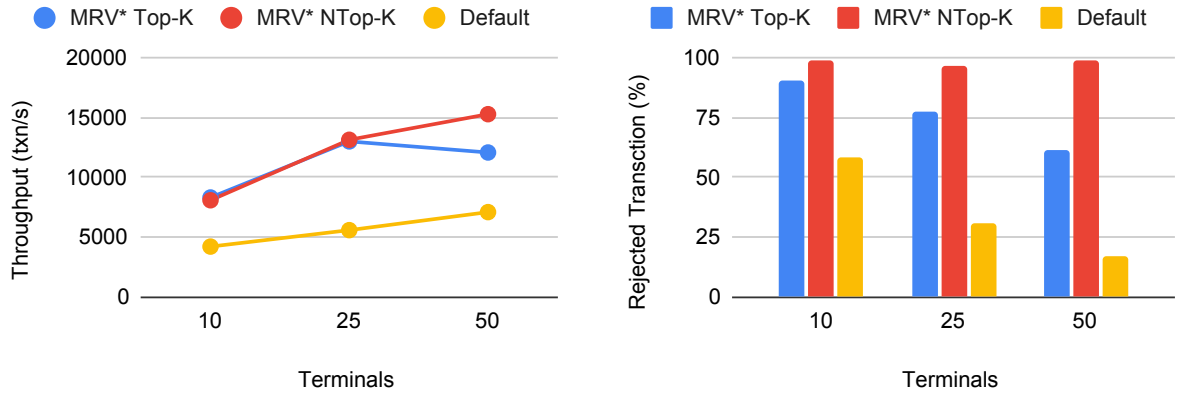
Regarding the Top-K operation, similar to the previous structures, the throughput is higher with the MRV* structures than the standard version in all tests, as observed in Figure 23a. In Figure 23b, it is evident that using the MRV* NTop-K structure results in a lower collision rate than the standard version for the different tested levels of concurrency. With 50 terminals, the throughput is 2.15× higher than the standard version when utilizing the MRV* NTop-K structure. Compared to the specific array implementation of the Top-K structure, results are similar with 10 and 25 terminals, however with 50 the NTop-K structure shows an



(a) OPut throughput comparison.

(b) OPut rejected transaction comparison.

Figure 21: OPut performance comparison for MRV* and non-MRV* for different workloads.



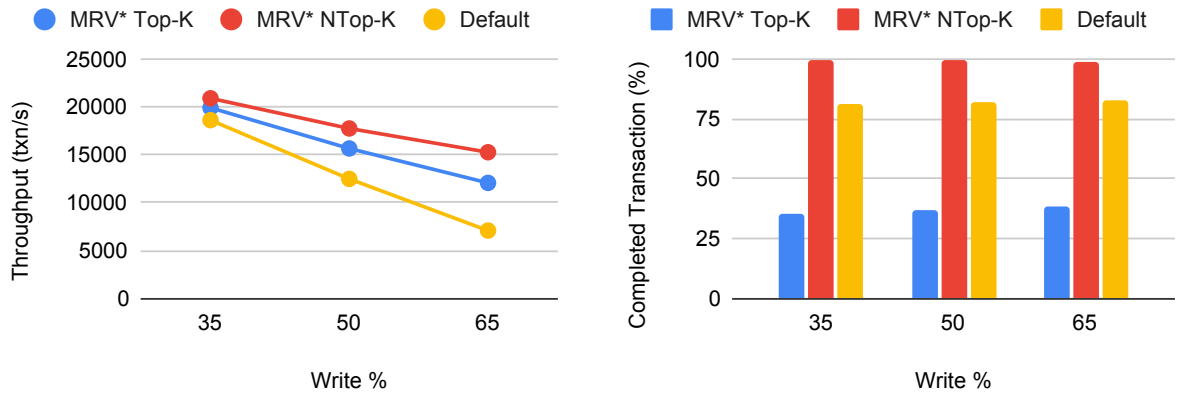
(a) Top-K throughput comparison.

(b) Top-K rejected transaction comparison.

Figure 22: Top-K performance comparison between MRV* and non-MRV* for different terminals.

increase of 27% in throughput.

Just like in the previous operations, for a higher volume of reads, the performance of the MRV* structures is also higher than the default version. Figure 23 shows that for scenarios where write operations are 35% of the workload, throughput with the NTop-K structure is 12% higher than default. When the write operations are 65% of the workload, there is a 215% increase in throughput. In general, the MRV* NTop-K structure shows better results than its array based counterpart. As such, for this operation, the NTop-K structure is the more adequate.



(a) Top-K throughput comparison.

(b) Top-K rejected transaction comparison.

Figure 23: Top-K performance comparison between MRV* and non-MRV* for different workloads.

4.2.2 Bidding benchmark

The bidding benchmark consists of one transaction that simulates the process of bidding on an item. The database is composed of 5 tables:

- **Item** - Contains general information about an Item.
- **User** - Contains general information about a User.
- **AuctionItem** - Contains an auction information, including the current maximum bid.
- **ItemBid** - Every time a user makes a bid, a record is created in this table with all its information.
- **UserItem** - It works as an inventory, a record is created if a user has an item.

A bid is composed of four steps as follows:

1) Read the current highest bid from the AuctionItem table.

2) Check if the new bidder exists in the User table.

3) Check if the new bid is higher than the current maximum bid and if so update the value.

4) Insert the new bid information on the UserBid table.

Listing 10 shows the non-MRV* SQL code. If multiple transactions try to update the $ai\_current\_price$'s value on the same record concurrently, some of them will have to abort to avoid serialization conflicts. Since tests were made using Snapshot Isolation, transactions always read from a consistent database state taken when the transaction starts, as such concurrent reads do not cause conflicts. However, concurrent updates will not be noticed until the transactions commit, which can lead to expensive rollbacks if multiple transactions update the same records.

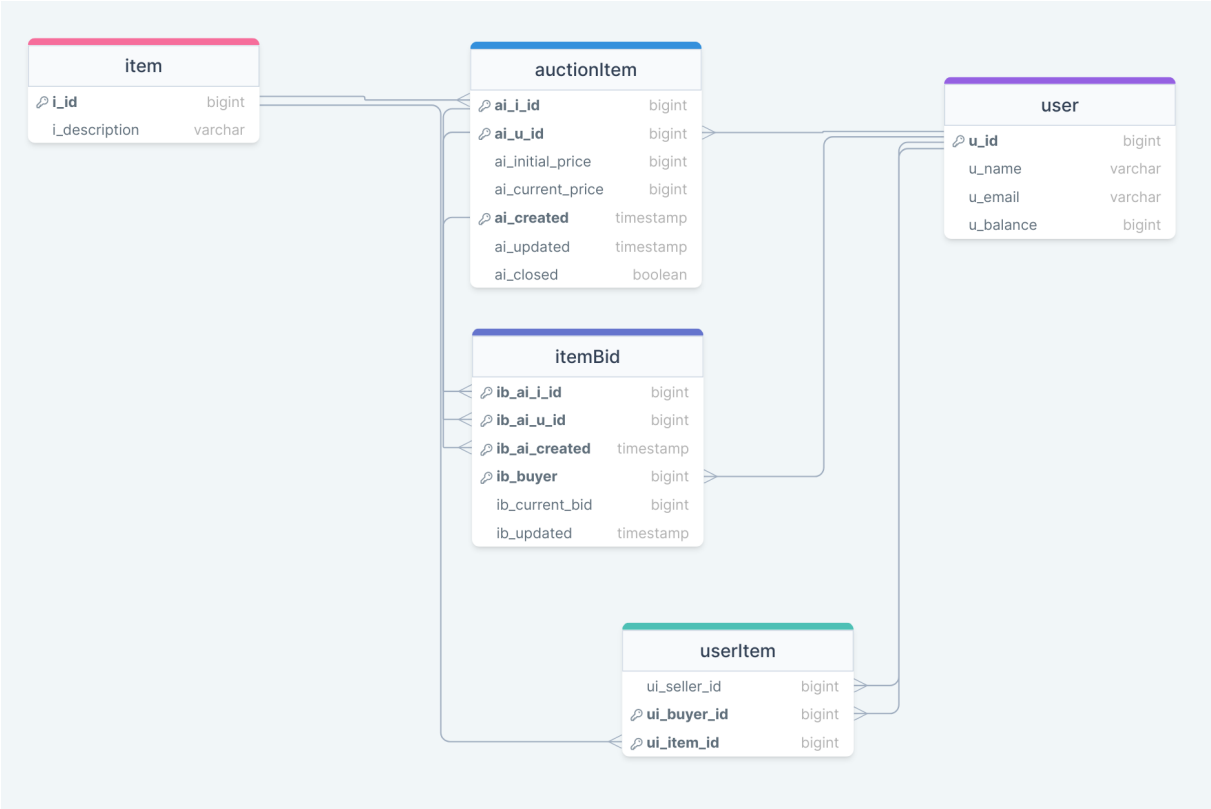


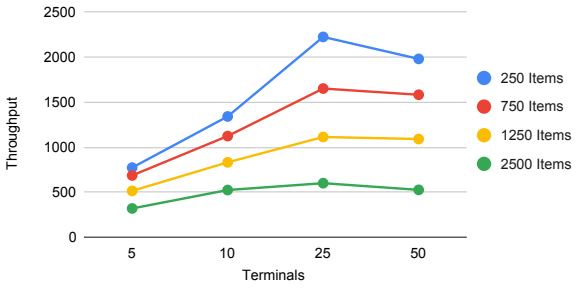
Figure 24: Benchmark SQL tables.

```
UPDATE auctionItem
SET ai_current_price = ?,
    ai_updated = ?
WHERE ai_i_id = ? AND
      ai_u_id = ? AND
      ai_created = ? AND
      ai_current_price < ?;
```

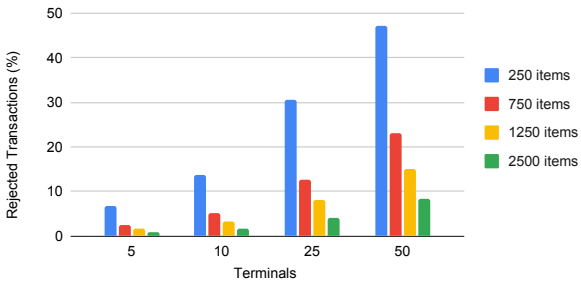
Listing 10: Update AuctionItem bid code.

The MRV* code is similar, but there is no need to explicitly compare the $ai\_current\_price$ with the new bid's value, since the update is intercepted and the comparison will be done with the selected record from the MRV* table.

Figure 25a compares throughput for a varying number of terminals, considering different number of items being auctioned. In general, we see better results when the number of terminals is higher, up to a certain degree. The best results were obtained with 25 terminals and 250 items with a throughput of 2224.37 transactions per second. When the number of terminals increases to 50, there is a decrease in throughput due to the increase in concurrency and an increase in rejected transactions from 30.5% to 47.3%, with throughput going down to 1980.35 transactions per second. Even though the number of

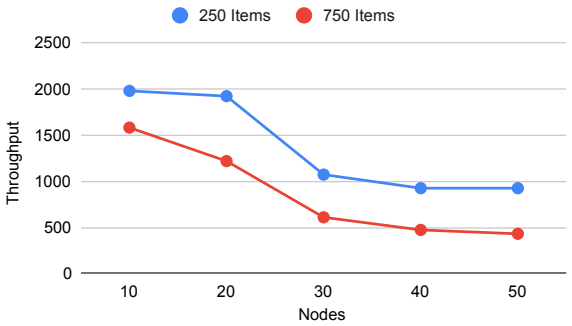


(a) NTop-K throughput comparison.

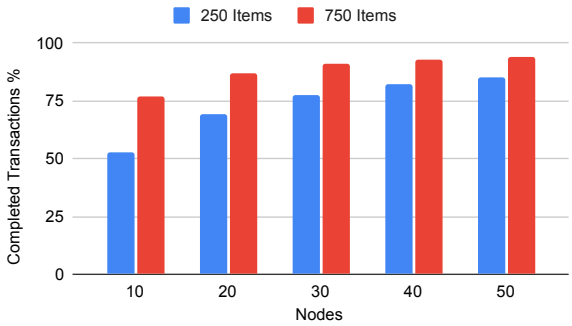


(b) NTop-K rejected transaction comparison.

Figure 25: NTop-K performance comparison for different terminals.



(a) NTop-K throughput comparison.



(b) NTop-K completed transaction comparison.

Figure 26: NTop-K performance comparison for 250 and 750 items.

rejected transactions is lower with more items, throughput is lower, this happens due to the increased cost of reading when there are more items.

Figures 26a and 26b compare throughput for two different scales, considering different number of nodes per MRV* and shows that the best results obtained for the NTop-K structure were with 10 per each record and a table with 250 records and achieved a throughput of 1980.35 transactions per second, resulting in a 40% increase in throughput, as showed in Figure 27, when compared with the default operation which had a throughput of 1413.67.

When scale increases, throughput decreases, this happens due to less concurrency, since the number of terminals is still the same (50) and there are more records to bid on. Reading costs also increase when using the structure, and the increase in records affects performance. Using more nodes per MRV* lessens concurrency problems but brings increased reading costs, as such since every transaction is reading and writing, the costs of reading become more evident because concurrency is not enough for the higher number of nodes to be worth the cost.

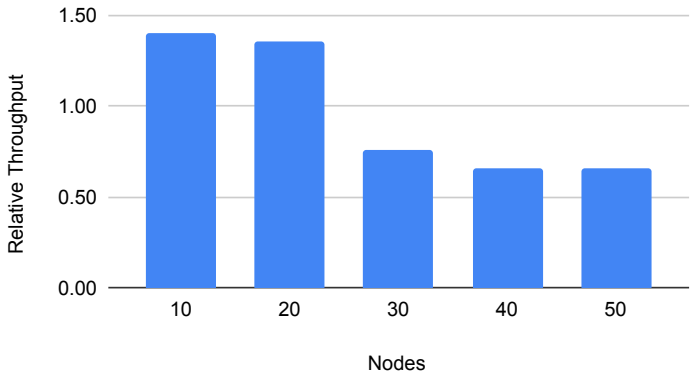


Figure 27: NTop-K throughput comparison between MRV* NTop-K and default with 250 item records.

4.3 Serial

4.3.1 Microbenchmark

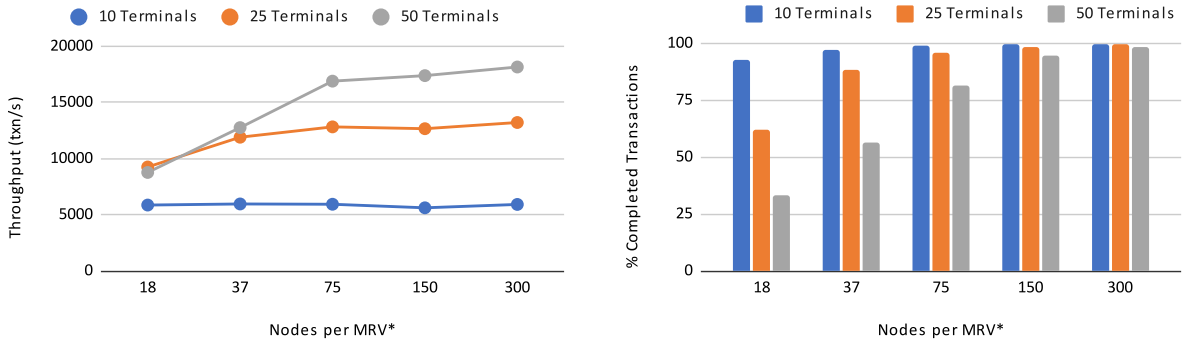
This benchmark consists of a write transaction that increments a record's value and a read transaction that reads it. There is a 65% chance of executing a write transaction and a 35% of executing a read. The tests performed using the microbenchmark ran 3 times with 10, 25, and 50 terminals executing operations on 10 MRVs*. Each test ran for 90 seconds.

Next alternative A

Figure 28a shows that this operation has better results when there are more nodes per MRV* when the number of terminals is 50. With 10 terminals, changing the number of nodes does not result in significant changes in throughput, since the structure can deal with this level of concurrency without much effort. When we increase the number of terminals to 25, we see that the increase in nodes results in higher throughput until 75 nodes. When we reach 75 nodes, the system can already deal with almost all transactions without many conflicts, leading to a transaction completion rate of 95.8% as shown in Figure 28b. With 50 terminals, we achieve a transaction completion rate of 99.3% with 300 nodes per MRV*, which leads to a throughput of 18161 transactions per second, which is a 67% increase when compared to the default operation without the structure.

Next alternative B

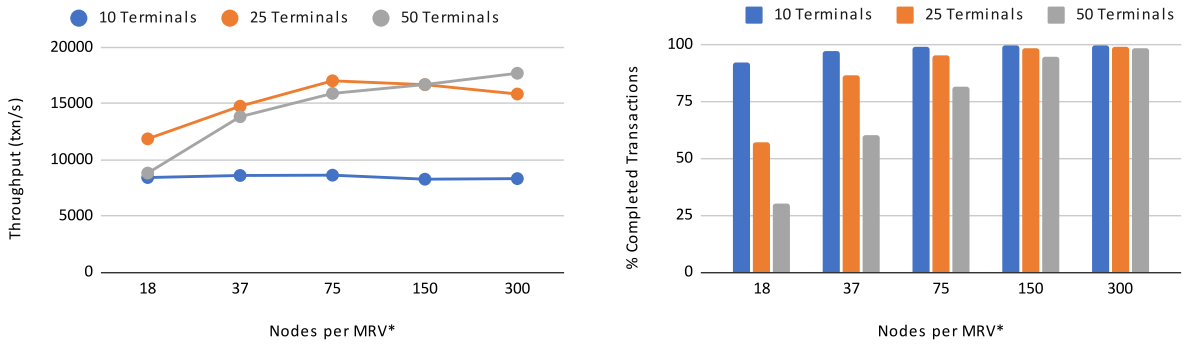
For Next B, Figure 29a shows that the best results were once again obtained when there are 300 nodes per MRV* and 50 terminals. With 10 terminals, changing the number of nodes does not result in significant



(a) Serial throughput comparison.

(b) Serial rejected transaction comparison.

Figure 28: Serial performance comparison with Next alternative A for MRV* for different workloads.



(a) Serial throughput comparison.

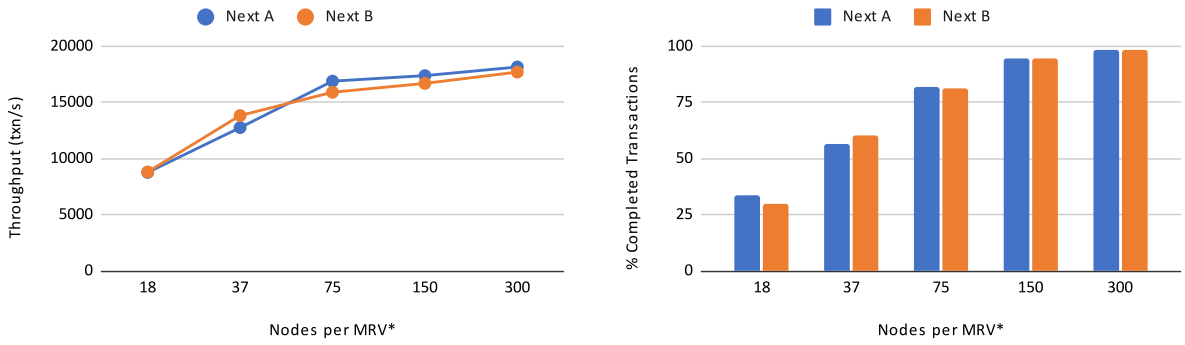
(b) Serial rejected transaction comparison.

Figure 29: Serial performance comparison with Next alternative B for MRV* for different workloads.

changes in throughput due to low concurrency for the amount of nodes. When we increase the number of terminals to 25, we see that the increase in nodes results in higher throughput until 75 nodes. When we reach 75 nodes, the system can already deal with almost all transactions without many conflicts, leading to a transaction completion rate of 95.4% as shown in Figure 29b. Increasing the number of nodes past 75 leads to a slight decrease in throughput due to higher reading costs and load increase for the Refresh worker. With 50 terminals, we achieve a transaction completion rate of 99.3% with 300 nodes per MRV*, which leads to a throughput of 17707 transactions per second, which is a 56% increase when compared to the default operation without the structure.

Comparing Next Versions

When comparing both version of the Next operation, we notice that they show similar throughput in almost all tests, with Next A being better when there are more than 75 nodes per MRV* as shown in 30a. When it comes to transaction completion, both versions also show similar results for all number of nodes. Between



(a) Serial throughput comparison.

(b) Serial rejected transaction comparison.

Figure 30: Serial performance comparison with Next B for MRV* and non-MRV* for different workloads.

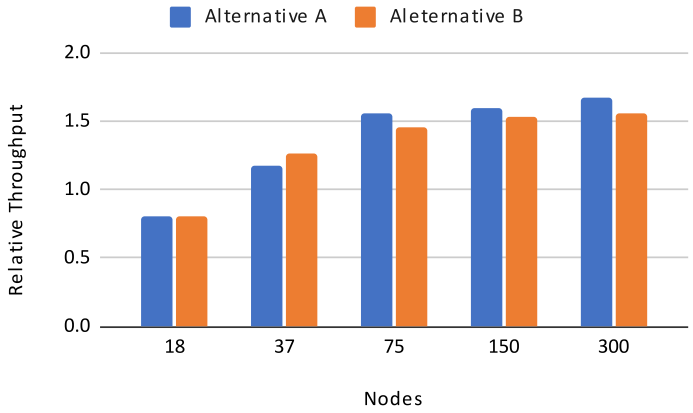


Figure 31: Serial different Next operations throughput comparison with default with 50 terminals.

both versions, the best result was obtained with Next A using 50 terminals and 300 nodes per MRV*.

Comparing both versions with the default operation, Figure 31 shows that for 50 terminals, if there are more than 37 nodes per MRV* the structure shows better results throughout all tests with both versions of the Next operation.

4.3.2 TPC-C

TPC-C [TPC] reproduces an order-entry environment, where operators execute transactions against a database. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock of warehouses.

One of TPC-C's most common abort causes is incrementing the $d\_next\_o\_id$ variable in the district table.This variable works as an incremental ID for the district's next order, so we can use the Serial structure to try to improve performance for this increment operation. TPC-C's $d\_next\_o\_id$ variable is part of

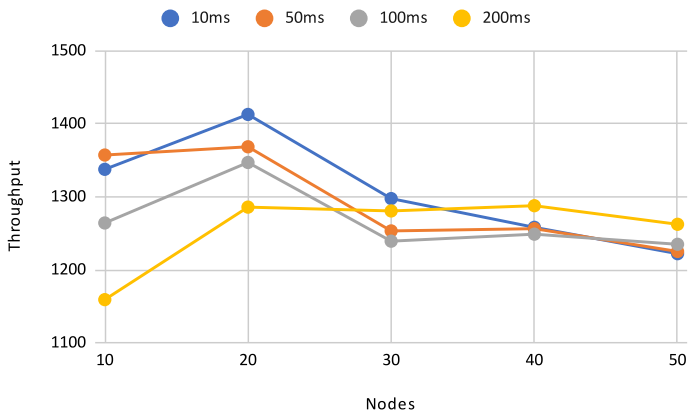


Figure 32: Next alternative A operation throughput comparison for different number of nodes and worker intervals.

some consistency conditions on the TPC-C specification [TPC]. The Serial structure cannot guarantee this conditions, however this is still a good test example for the structure in a general workload.

Tests were made considering both versions of the Next Operation and are divided accordingly in this section. For both versions of the Next Operation tests were divided in two categories, in the first the tests were conducted for different worker intervals and different number of nodes per MRV*. In the second, tests considered the best results from the previous one and applied those settings to different workload scales.

Next alternative A

Figure 32 shows how throughput varies when changing the number of nodes per MRV* with a TPC-C scale of 1 warehouse. This structure is also dependent on how the worker performs, as such these results are shown for different Refresh worker intervals. The best results obtained with 1 warehouse were with 10 nodes per MRV* and the worker running every 10ms with an average throughput in 3 runs of 1349.73 completed transactions per second. Results were generally better using a lower number of nodes. Figure 32 shows that the top 4 tests throughput wise were all with 10 or 20 nodes.

When the number of nodes is higher, lower refresh rates start having better results since there are more nodes available and as such refreshes are not needed as often considering the same load. Refreshes get more expensive when there is an increase in system load and serial records, since more records will have to be refreshed.

Figure 33 shows all tests had an increase in throughput ranging from a 1% to a 20% increase. In general, the best results came when using a lower number of nodes, which can be a result of the higher

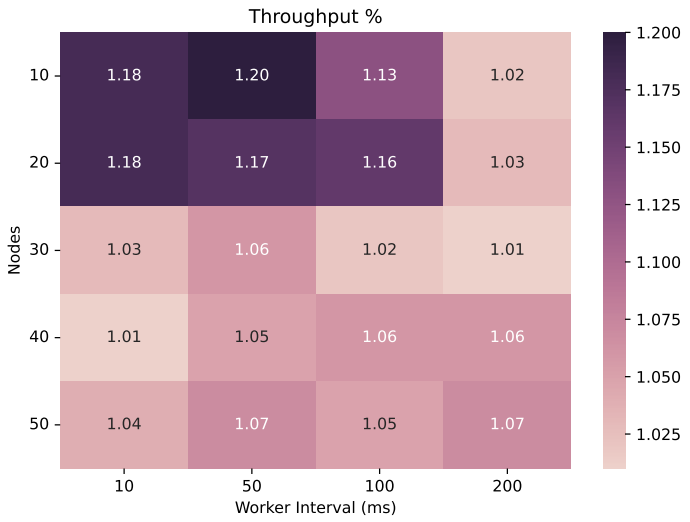
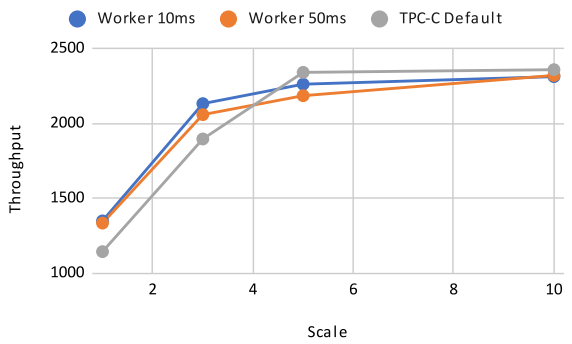


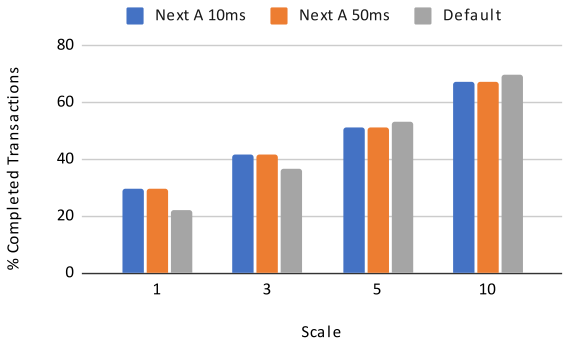
Figure 33: Next alternative A operation throughput comparison with default.

costs of read operations with more nodes. A higher number of nodes also makes refreshes more expensive due to the higher number of records.

Figure 34a and Figure 35 shows how throughput varies using 20 nodes and refresh rates of 10 and 50ms for different scales. Lower scales have higher concurrent accesses to data since there are fewer warehouses and as such fewer options for the transactions to choose. So, as expected, the Serial structure is better than the default for lower values (1 and 3 warehouses). When there are 5 or 10 warehouses, the TPC-C default version is better, since there is not enough data access concurrency for the structure to provide the best results. However, the throughput while using the structure is similar to the throughput when it is not being used for 50 warehouses. In fact, the throughput is 98% of the default version which is only a 2% decrease, which can be justified considering we have an 18% increase in throughput when there are higher degrees of concurrency (1 warehouse). The worst case scenario found during these tests appeared when using a scale of 5 warehouses with a refresh interval of 50ms, where the throughput was 7% lower than the default version. However, this was a very specific scenario. For the same scale, if we just change the refresh interval to 10ms instead of 50, the performance drop compared to the default version is just 3%. Since different worker settings affect performance, it is recommended to benchmark the system in order to obtain the best performance for each use case. Figure 34b shows the percentage of completed transactions using the Next A Operation compared to the default when there is an increase in scale. As expected, when there is a higher number of warehouses for the same number of terminals, concurrency decreases and the Serial structure becomes ineffective.



(a) Next A operation throughput comparison.



(b) Next A operation completed transaction percentage.

Figure 34: Performance comparison of Next A for different scales with 20 nodes.

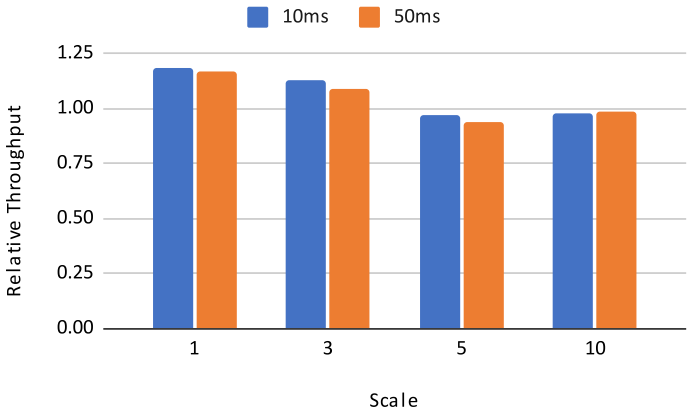


Figure 35: Next alternative A operation throughput comparison with default for different scales with 20 nodes.

Next alternative B

Figure 36 shows that the best results for this version of Next were obtained using 20 nodes per MRV* and a refresh rate of 10ms. This result was the best of all the performed tests for 1 warehouse, obtaining a throughput of 1413.09 transactions per second, which results in an improvement of 24% in throughput compared to the default version of TPC-C. Every time a transaction executes the Next operation in this version, a node is temporarily removed until it is refreshed, so when the number of nodes is lower, a higher refresh rate allows more nodes to be available at once. If the refresh rate is too low, transactions may abort due to the lack of available nodes. This version of Next behaves similar to the previous one, and lower refresh rates work best with a higher number of nodes per MRV*, while higher refresh rates are ideal when the number of nodes is lower.

Figure 38a and Figure 39 shows similar results to the previous version, where the Serial structure has

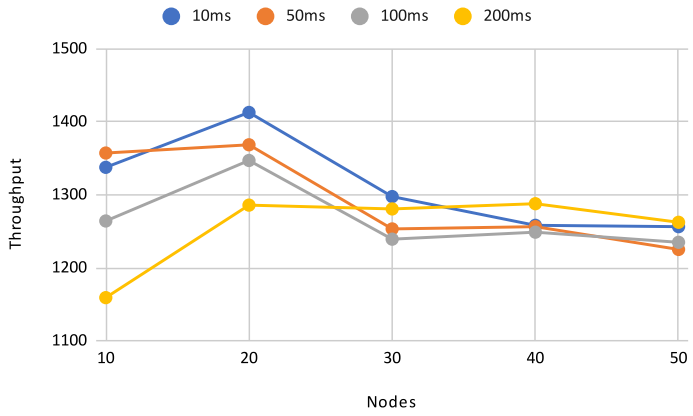


Figure 36: Next alternative B operation throughput comparison for different number of nodes and worker intervals.

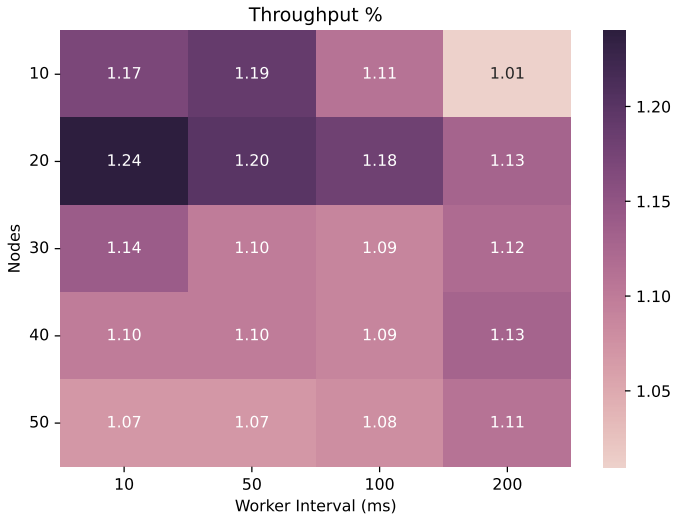
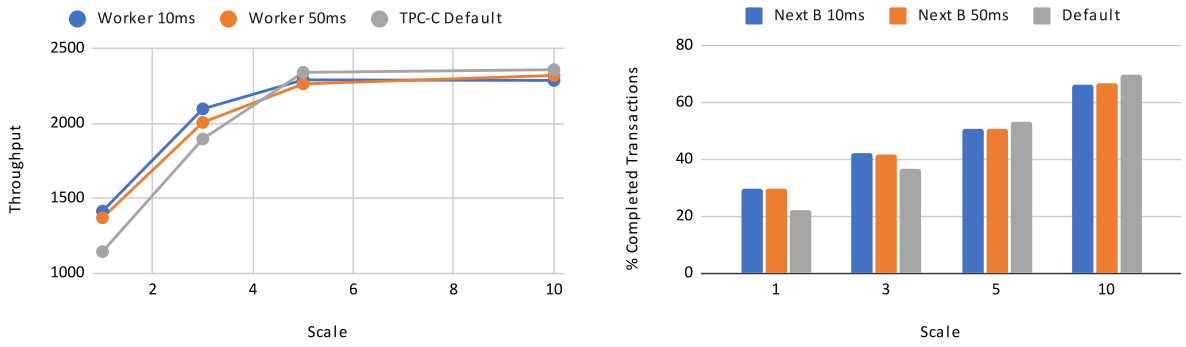


Figure 37: Next alternative B operation throughput comparison with default.

higher throughput for lower scales (1 and 3 warehouses), while the default version is slightly better for higher scales (5 and 10 warehouses). For a scale of 1 with refreshes executing every 10ms, there is a 24% increase in throughput. For 3 warehouses we still see an improvement with a higher refresh rate with an 11% increase in throughput. For a higher scale, there is a decrease in throughput when comparing to the default version of 2% with 10ms refreshes for 5 warehouses and 2% with 50ms refreshes for 10 warehouses. Once again, the decreased concurrency for higher scales makes it so that the structure does not provide ideal results however, it still has a similar throughput to the default version even in these less than ideal scenarios.

Figure 38b shows the percentage of completed transactions using the Next B Operation compared to the default when there is an increase in scale. As in the previous version of the Next Operation, when



(a) Next B operation throughput comparison.

(b) Next B operation completed transaction percentage.

Figure 38: Performance comparison of Next B for different scales with 20 nodes.

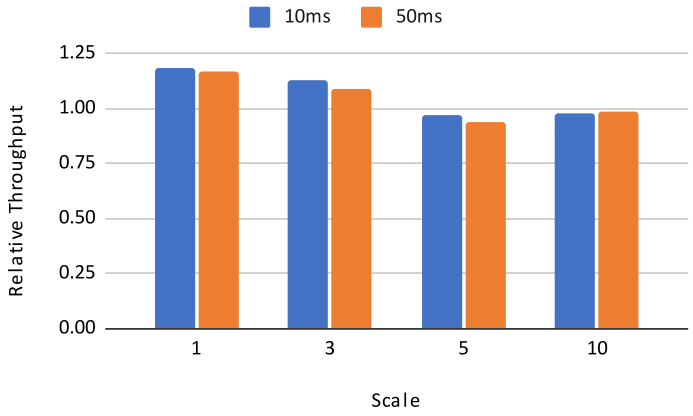
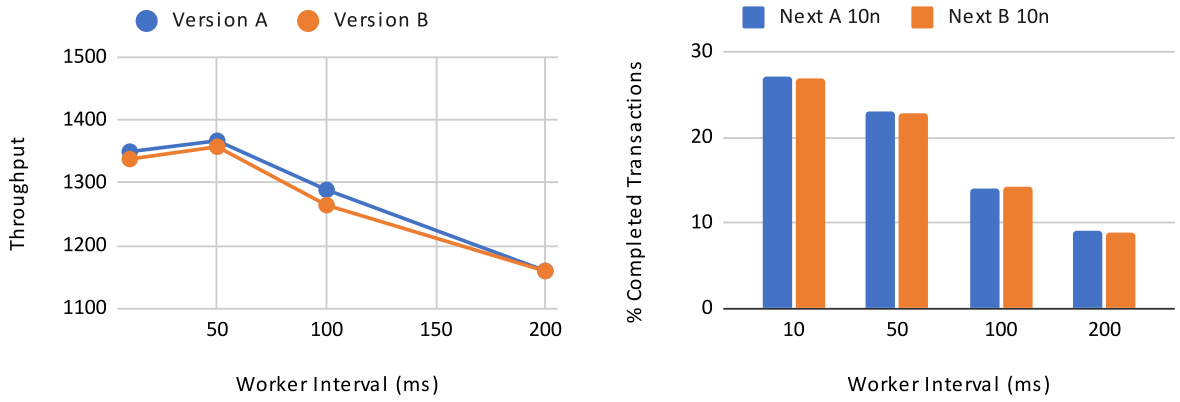


Figure 39: Next alternative B operation throughput comparison with default for different scales with 20 nodes.

there is a higher number of warehouses for the same number of terminals, concurrency decreases and the Serial structure becomes yet again ineffective.

Comparing Next Versions

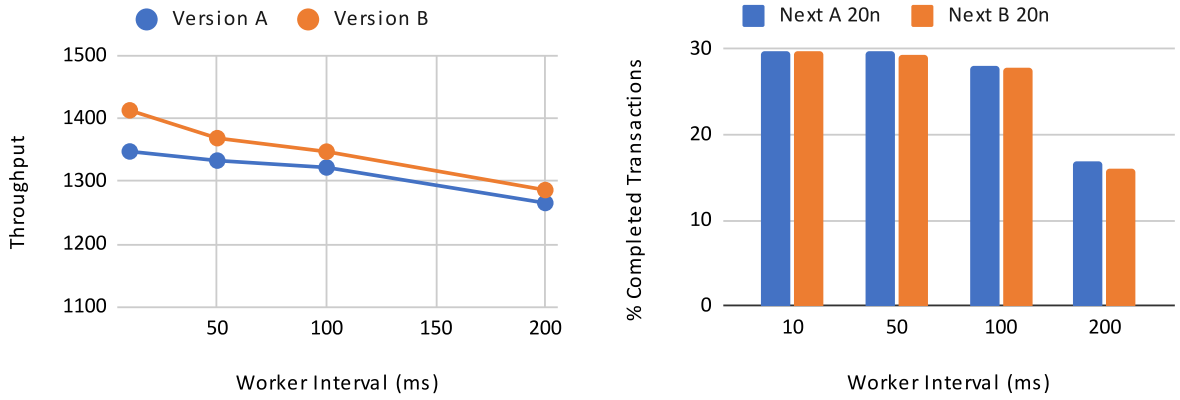
Figure 40a to Figure 44a show how the both Next Operations compare throughput wise for 10 to 50 nodes respectively. Figure 40b to Figure 44b show the respective percentage of completed transactions for each test. In general, Next alternative B has a higher throughput than Next alternative A for the different number of nodes. When there are only 10 nodes per MRV* Next A has better results, however for all other tests Next B has better performance.



(a) Next throughput.

(b) Next completed transaction percentage.

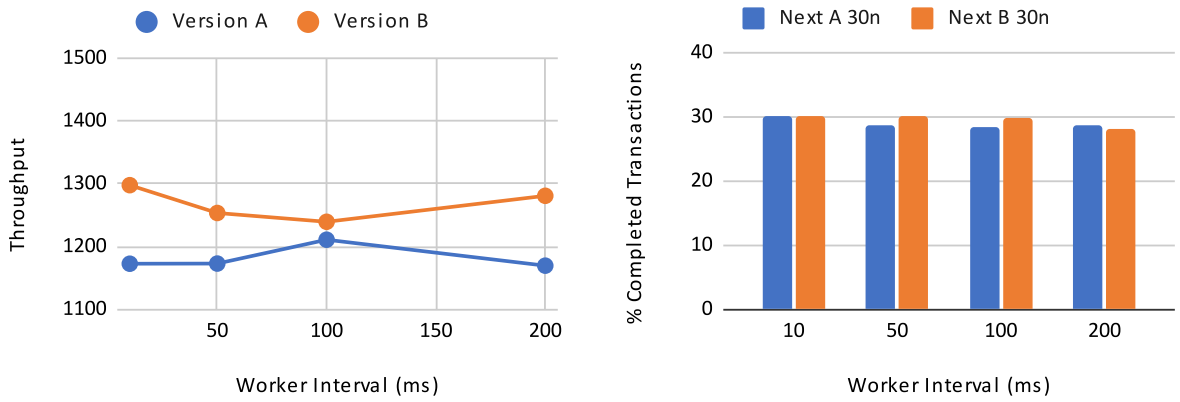
Figure 40: Comparison between both versions of the Next operation with 10 nodes per MRV*.



(a) Next throughput.

(b) Next completed transaction percentage.

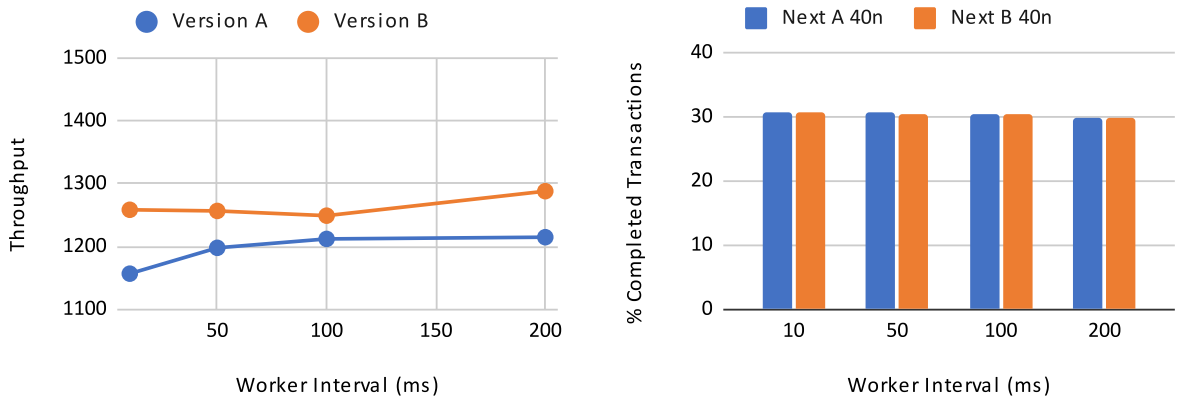
Figure 41: Comparison between both versions of the Next operation with 20 nodes per MRV*.



(a) Next throughput.

(b) Next completed transaction percentage.

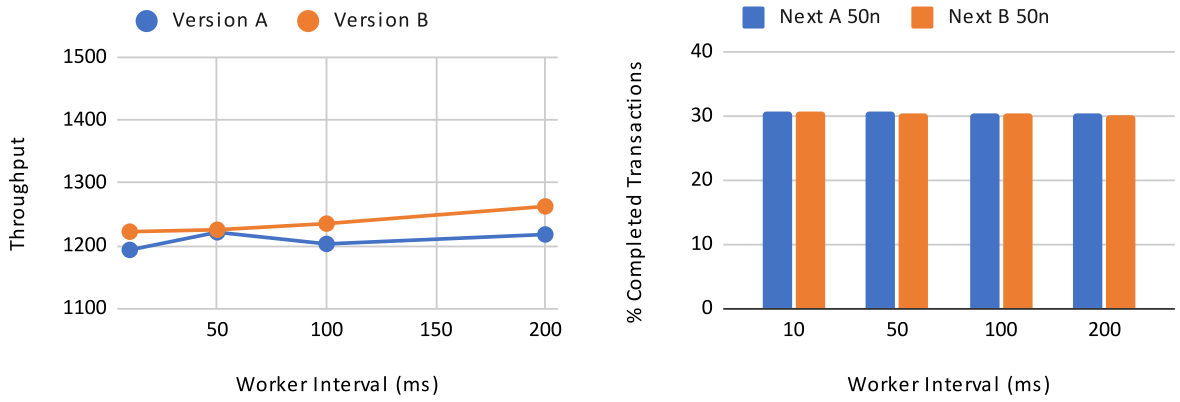
Figure 42: Comparison between both versions of the Next operation with 30 nodes per MRV*.



(a) Next throughput.

(b) Next completed transaction percentage.

Figure 43: Comparison between both versions of the Next operation with 40 nodes per MRV*.



(a) Next throughput.

(b) Next completed transaction percentage.

Figure 44: Comparison between both versions of the Next operation with 50 nodes per MRV*.

Chapter 5

Conclusions and future work

5.1 Conclusions

In this work, we have extended the applications of Multi-Record Values (MRVs) by implementing new operations for data structures that had not yet been addressed using this method. We proved that the MRV technique can also be used with different structures. First, we found that it is possible to obtain performance gains similar to Phase Reconciliation but without the impact of reconciling and having a fixed number of cores. Second, we found that it is possible to obtain a serial structure using MRVs without gaps and without locking mechanisms.

The Serial structure results show that there is an increase in throughput of up to 20% in most of the tests. If used properly, selecting appropriate parameters for each scenario, the structure can be used to increase performance any time a serial number is needed. The NTop-K structure has also showed an increase in performance in many tests when compared to the default operations, and also showed a similar throughput to most of the specialized custom structures developed. This shows that this generalization can be used, especially in scenarios where the OPut or Top-K operations are used with good performance. When compared with the Max dedicated structure, the NTop-K has a slight lower throughput, however, it is still an improvement when compared with the default operation.

In general, this implies that the using this structures in distributed systems could potentially lead to enhanced performance while maintaining coherence guarantees across multiple nodes. Given that these structures are built upon the principles of Multi-Record Values, they can be applied to existing transactional database systems without modifications to their engine code, which is particularly valuable in systems where the source code is not accessible.

5.2 Prospect for future work

For the NTop-K structure, improving the Read operation should lead to a more scalable structure when there is an increase in node number and workload scale. Results were positive when the number of nodes and workload scale were lower, however, performance deteriorates when these parameters increase.

At the moment, for the Serial, only one Refresh worker is available at a time, as such if the system load increases significantly, it becomes necessary to have multiple workers refreshing registers simultaneously. Implementing multithreaded Refresh workers, so that it lessens the load on each worker when there is a higher number of registers per MRV* might be able to bring more performance. The Refresh worker algorithm could also be improved in order to consider if refreshing a register is necessary according to the current load. At the moment, results using both the Refresh and the Adjust workers result in a performance drop, as such, another topic that can be improved in the future is finding the ideal Refresh rate and Adjust worker work rate, so that the system can be dynamic and adjust itself to the current load without causing performance drops.

At the current time, the structures were only tested using a single database machine. As such, it would be important to test them in distributed systems so we can get new insight on their performance in these scenarios. Since this structures rely on randomness and do not require synchronization to run, it is expected that their usage will also lead to an increase in performance.

Bibliography

AuctionMark. AuctionMark: An OLTP Benchmark for Shared-Nothing Database Management Systems « H-Store, November 2023. URL https://hstore.cs.brown.edu/projects/auctionmark/.

Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 31–36, September 2015. doi: 10.1109/SRDS.2015.32. ISSN: 1060-9857.

Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986. ISBN 0201107155.

Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013. URL http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

Nuno Faria and José Pereira. MRVs: Enforcing Numeric Invariants in Parallel Updates to Hotspots with Randomized Splitting. *Proc. ACM Manag. Data*, 1(1), may 2023. doi: 10.1145/3588723. URL https://doi.org/10.1145/3588723.

H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, jun 1981. ISSN 0362-5915. doi: 10.1145/319566.319567. URL https://doi.org/10.1145/319566.319567.

Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-Memory Transactions. page 15, 2014.

Patrick E. O'Neil. The Escrow transactional method. *ACM Transactions on Database Systems*, 11(4): 405–430, December 1986. ISSN 0362-5915, 1557-4644. doi: 10.1145/7239.7265. URL https://dl.acm.org/doi/10.1145/7239.7265.

PostgreSQL. PostgreSQL 14.10 Documentation, November 2023. URL https://www.postgresql.org/docs/14/index.html.

Nuno Preguiça. Conflict-free Replicated Data Types: An Overview, June 2018. URL http://arxiv.org/abs/1806.10254. arXiv:1806.10254 [cs].

Max Roser, Hannah Ritchie, and Esteban Ortiz-Ospina. Internet. *Our World in Data*, 2015. https://ourworldindata.org/internet.

Transaction Processing Performance Council (TPC). TPC-C Standard Specification Revision 5.11, 2010. URL https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.