Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Transaction isolation with the append-only log concept in the key-value storage

Author: Alina Boshchenko (2732782)

1st supervisor:	Dr. A. Belloum, *University of Amsterdam*
Daily supervisor:	A. Lomakin, *JetBrains*
	E. Naumenko, *JetBrains*
2nd reader:	Dr. F. Regazzoni, *University of Amsterdam*

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 17, 2023

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Context. This master thesis explores the adoption of a new transactional concept in embedded databases, with a focus on improving performance and maintaining database properties. The motivation for this research stems from the need for improved reading operation speed, efficient disk storage utilization, alleviation of transaction competition, the potential for multi-node architecture, and elimination of heavyweight garbage collection. The proposed approach is based on the utilization of an operation log concept for transaction implementation instead of copy-on-write B-trees.

Goal. The research aims to develop and implement a new transaction isolation concept in key-value storage, specifically Xodus DB. The primary objectives of the research are to identify and address concerns related to the current transaction implementation in key-value storage, perform a literature review of new approaches and issue mitigations, modify the selected algorithm if required, design and implement key components and the logic of the new approach, validate the results obtained, and document the modifications while evaluating the trade-offs and outcomes of the new method.

Method. The thesis begins with an examination of existing and proposed algorithms, continues with suggesting modifications and new concepts and subsequent implementations, and concludes with benchmarking and results reporting and discussion. The methodological approach can be summarized into the following points: familiarization with the current transaction implementation, literature review, algorithm selection and modification, component design and implementation, validation and results analysis.

Results. The average latency for read operations was improved by an average 75%, and the average latency for write operations was improved by an average 82%. The overall runtime boost achieved is $\approx$ 10x.

Conclusion. The research's main contribution lies in the modified algorithm and its adoption for key-values stores in embedded databases demonstrated

through the evaluation, results discussion, and benchmarking of the new transactional concept. We managed to develop the algorithm and show the impact of its implementation on the real-life example of key-value storage, providing a significant reduction in read-write latencies and overall run time.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

In the rapidly evolving landscape of enterprise databases, the role of transactions cannot be underestimated. Transactions serve as the backbone of any robust and reliable database system, ensuring the integrity and consistency of data operations. However, as businesses face increasingly complex challenges, the implementation of high-quality transactions becomes paramount. In this context, numerous requirements are placed on the design and execution of transactions within enterprise databases. Among these requirements, performance stands out as a critical factor. Slow response times or system bottlenecks can have severe implications, hindering business operations and diminishing overall productivity. Scalability is another key consideration for enterprise databases. Without proper scalability measures in place, a database system may struggle to handle the demands placed on it, leading to suboptimal performance and resource wastage. This work is focused on the development of the new transactional concept in key-value storage, aimed to improve performance and maintain the required properties of the database.

1.1 Motivation and problem statement

1.1.1 Motivation

In today's fast-paced business world, enterprises are dealing with a growing number of users who demand seamless and rapid operations. However, many organizations struggle to manage their expanding databases effectively, which can lead to financial losses and decreased productivity. In the midst of these challenges, a game-changing research paper (2) has provided inspiration. This work offered innovative solutions for achieving top-notch performance in key-value storage. Motivated by their findings, our goal is to build upon

their ideas and extend them further to rethink our approach. We aim to develop a new concept of transaction management and implement it for a real-world case – Xodus DB.

1.1.2 Problem statement

The current state of the system utilizes B-trees, a self-balancing tree data structure, to store key-value pairs separately from the pages (nodes) of the B-tree itself. While this approach allows for efficient data retrieval, it introduces overhead and limitations when dealing with large amounts of data that cannot fit in the main memory. Our goal is to address these limitations and optimize the database system further.

To better understand the motivation behind our work, let us consider the use of B-trees in scenarios where data exceeds the capacity of the main memory. When the number of keys is high, the data is read from the disk in blocks, and disk access time is significantly slower compared to the main memory access time. The primary idea behind using B-trees is to minimize the number of disk accesses, thus reducing latency. However, the current implementation of Xodus suffers from additional disk access overhead due to the separation of key-value pairs from the B-tree nodes.

In the current implementation, we utilize Copy-on-Write (CoW) B-trees, where updates to the tree are performed by writing in append-only mode. In this approach, every time the data is modified, the entire path from the root to the leaf node is copied and rewritten, ensuring the isolation of transactions. While this method eliminates the need for disk access to retrieve keys, it introduces a significant overhead during write operations.

In our investigation, we analyze the limitations of the existing implementation and propose a novel concept based on the responsibility sharing between transaction and data components and introduction of the operation log (2) to handle transactions more efficiently.

By leveraging the OL concept, we aim to address several shortcomings of the current system. Firstly, by moving transaction data from memory to disk, we eliminate the limitation imposed by the available RAM, allowing for the handling of larger transactions without the risk of running out of memory. OL itself will be stored in disk files (.xd files). Secondly, in the OL transactions will compete only in case of the same key, enabling improved write performance compared to the current global lock approach where all the transactions are competing against each other.

Moreover, the proposed OL concept shares similarities with the operation log in the RAFT* algorithm (3), making it a potential stepping stone toward transforming Xodus into a distributed storage system. Ideally, we would aim while having one node to be able

to replicate from the first node to the second node. If a node falls, it would be possible to get a new one and replicate to it, etc. This change would lead to producing fewer backups, but this part is out of the scope of this thesis.

Furthermore, the introduction of the OL concept eliminates the need for the current heavy garbage collector (GC), which can be resource-intensive and cause delays in data processing. The new GC is discussed in detail in Section 4.4.3. The new lightweight GC will no longer require logarithmic search like the current one or additional data loading, resulting in improved overall performance and efficiency.

1.2 Goal

Design and implement a new transaction isolation concept using the append-only log approach in the embedded database.

1.3 Tasks

- Become familiar with the concept of the current transaction implementation in the embedded database (Xodus) and algorithms behind it, and state the concerns;
- Perform a literature review of the new approaches and performance issues mitigations, select the suitable approach that allows solving existing problems from the performance and usability points of view;
- Modify and enhance the selected algorithm if needed, design and implement key components and the logic of the new approach;
- Perform the validation of the result obtained;
- Document the modifications and evaluate the results and trade-offs of the new method.

1.4 Research Question

RQ1: What strategies can be employed to develop an algorithm based on (2) to effectively address the key-value store issues discussed in Section 1.1.2?
RQ1.1: How can we modify and enhance (2) so that it can handle scalability for the case of Xodus DB?

RQ1.2: How can we perform the validation of the new algorithm and evaluate the improvement?

1.5 Research value

This thesis investigates the limitations of the current Xodus database system and proposes the introduction of the new concept to enhance its performance and efficiency. By addressing issues related to disk access overhead, memory limitations, write performance, and garbage collection, we aim to provide a solution that optimizes the transaction handling process. The contributions of this work include a new approach to transactions that improves the scalability and performance of the database system, and setting the foundation for potential future transformations into a distributed storage system.

2

Background

2.1 Environment

In this thesis, we explore the algorithmic improvements aimed at enhancing the performance and efficiency of key-value storage by introducing a new approach to transactional management. Despite that this algorithm can be applied to various key-value storages we decided to select a case on which we will test and tune the suggested concept. As a demonstration case, we selected an embedded No-SQL database Xodus by JetBrains (4). JetBrains Xodus is a transactional schema-less embedded database that is written in Java and Kotlin. The reasons behind this choice are the following:

1. Xodus is an open-source database and we have full access to its code which makes the process of delivering and testing improvements faster;
2. Xodus is embedded meaning that no installation or administration is required;
3. Xodus is written in Java and Kotlin, all the participants in this research are skilled in these languages and can utilize JVM functionality for the performance goals.

From this point, we will also use the notation of system/software to refer to Xodus.

2.2 Memory

In this section, we would like to briefly describe the types of memory that we will use in the course of the research. It is mostly considered to be common knowledge on Memory management, but I would like to highlight it as we will need some details later.

Disk Memory

Disk memory refers to long-term storage devices such as hard disk drives (HDDs) or solid-state drives (SSDs) (5), (6). It provides non-volatile storage, meaning the data remains persistent even when the power is turned off. Disk memory has a much larger capacity than the primary memory (RAM), but it is slower to access.

Data on disk memory is stored in the form of blocks or sectors. These blocks are addressed by their physical location on the disk using a file system. When data needs to be accessed, the disk's read/write head moves to the specific location, and the data is read from or written to the disk. Disk memory is typically used for storing files, databases, and other long-term data.

RAM (Random Access Memory)

RAM, also known as primary memory or main memory, is a type of volatile memory that stores data and instructions that are actively being used by the computer's processor (5), (7). It provides fast and temporary storage for data that the CPU needs to access quickly.

RAM is a physical hardware component of a computer system and is typically made up of integrated circuits. It is directly accessible by the processor, allowing for rapid data retrieval and manipulation. RAM retains data as long as power is supplied to the computer, but it loses its contents when the power is turned off or the system is restarted.

RAM is organized into memory cells, each capable of storing a certain amount of data (usually 8 bits or 1 byte). These cells are addressed by unique identifiers, allowing the processor to read from or write to specific locations in RAM.

Programs and data that are currently in use by the operating system and running applications are loaded into RAM for faster access. The amount of RAM in a computer determines how much data and how many programs can be stored and accessed simultaneously.

Heap Memory

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. It is part of the primary memory (RAM) and provides faster access than disk memory (6). Whenever we create an object, it's always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that don't have any reference.

Heap memory is organized in a more flexible manner compared to the fixed-size blocks of disk memory. It allows dynamic allocation of memory blocks based on the program's requirements. When an application requests memory from the heap, the memory manager

finds a suitable block of memory and returns a pointer to that block. The program can then use the allocated memory for data storage.

2.3 Key-value storage

A key-value store is a type of data storage that stores data as a set of unique identifiers, each of which have an associated value. These pairs are called “key-value pairs”. The unique identifier is the “key” for an item of data, and a "value" is either the data being identified or the location of that data.

Key-value stores play a crucial role in numerous systems, offering the fundamental service of storing and retrieving data with different levels of reliability and performance. These stores are employed in databases such as DynamoDB (8), Cassandra (9) and other and file systems such as (10). Much like file systems, the majority of persistent key-value stores are designed to withstand machine failures without compromising data integrity. Building services that can gracefully handle failures while maintaining good performance is a challenging task.

2.4 Copy-on-Write B-trees

B-trees (11), (12) are efficient data structures suitable for storing large amounts of data. They have a high branching factor, with each node having numerous children. The nodes maintain keys in sorted order and ensure that this order is maintained in their sub-trees. For the node N_i with key K all the children of N_i preceding index i store keys that are less than K, while all of the children following index i store keys greater than or equal to K. B-trees provide logarithmic time key-search, insert, and remove. An example of the insert operation is shown in Figure 2.1.

Current implementation of the database uses B+-trees (referred to as "b-trees" hereafter), which denotes the variant that stores all values in the leaf nodes. B+-trees have the advantage of a more compact interior node structure, making them well-suited for disk-based operations.

File systems such as Linux file system (10) employ copy-on-write (COW) B-trees to ensure crash recovery and atomicity. When a modification is made to a node in the tree, a copy of the node is created at a new disk location, and the modification is applied to the copy. Subsequently, the parent node is updated to point to the new location. Changes propagate up the tree to the root. Upon successful completion of the operation, we can

discard the old root. One of the main benefits of this approach is that in the event of a system crash, copy-on-write preserves the original tree structure.

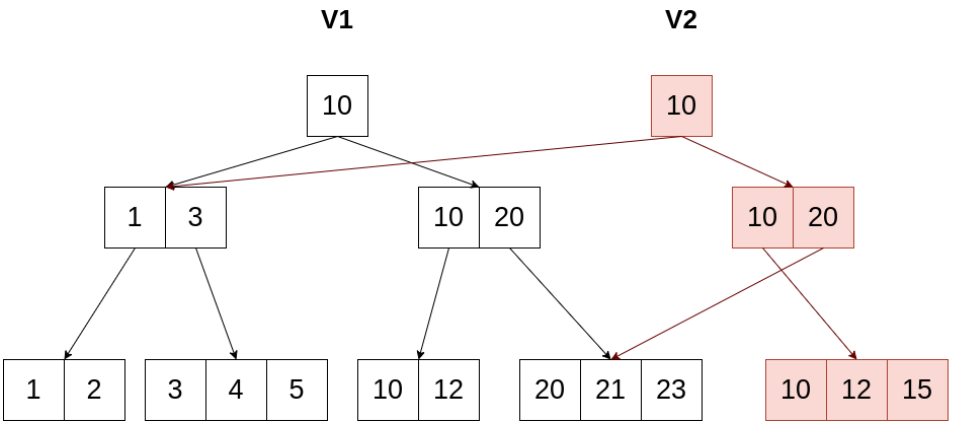

Figure 2.1: Inserting new node 15 in B-tree, the whole branch leading to the target leaf in copied. Red nodes denote newly created version, white – old version.

The number of disk accesses required for reading from or writing to the tree in minimized as B-trees are designed to fit precisely within a single disk block, typically 4KB.

2.5 Operation log

In key-value storages, an operation log, also known as a write-ahead log (WAL) or commit log (13), is a data structure used to record all modifications or updates made to the database. It serves as a persistent record of write operations before they are applied to the underlying data store.

The operation log works by appending each write operation, along with its associated key and value, to a sequential log file. The log file is typically stored on disk or in a durable storage medium. The log entries are written sequentially and sequentially read during recovery or when replaying the log.

The purpose of the operation log is to ensure durability and recoverability of the data in the event of failures or crashes. By writing the modifications to the log before applying them to the main data store, key-value storages can guarantee that data modifications are not lost. In case of a system failure or crash, the log can be replayed during the recovery process to restore the database to a consistent state.

The operation log also helps in achieving atomicity and consistency. By maintaining a sequential log of write operations, it becomes possible to replay or roll back the log entries to ensure that the database remains in a consistent state.

Overall, the operation log is a critical component in key-value storages as it provides durability, recoverability, and helps maintain data consistency in the face of failures or crashes.

2.6 Garbage collector

In key-value storage systems, a garbage collector (GC) is a component responsible for reclaiming disk space that is no longer in use or is occupied by deleted or expired data. It helps to optimize storage efficiency and maintain the overall performance of the system.

In Xodus GC moves useful data to the end of the storage, more specifically to free files – the delta of the whole path from the root to leaf with the minimal key which is not yet in a new file is copied to this free file. It is not possible to perform any write transactions during GC working time, because there is a stage in GC execution when even read transactions can't be performed.

The reason behind write transactions being blocked is the fact that GC does such a huge amount of work so countless repetitions of this work will lead to blocking any useful data processing. Read transactions are blocked due to the following: GC is aimed to delete files, therefore it is essential to make sure that there are no read transactions for those files and durability is valid. So on file deletion, the synchronization of the new files is essential. Therefore, the process becomes the following (Figure 2.2):

1. Stop all read transactions;
2. Do force synchronization of new files;
3. Remove old files;
4. Resume read transactions;

This workflow makes the GC extremely heavy which also supports the motivation for the improvements.

2.7 Queries

Range query and point query are two commonly used types of queries in database systems.

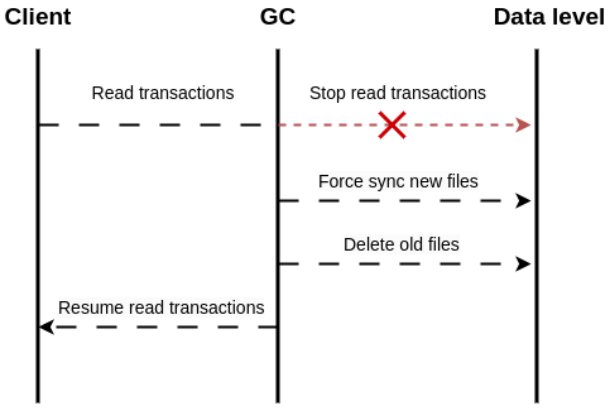

Figure 2.2: Sequence diagram for the garbage collector flow

2.7.1 Range Query

A range query is a type of database query that retrieves all the records or data points within a specified range of values. For example, a range query in a sales database might retrieve all the sales transactions that occurred between a specific start and end date, or it could retrieve all the products with prices within a certain price range. Range queries are useful for extracting data that falls within a specified interval or criteria.

2.7.2 Point Query

A point query is a database query that retrieves a single record or data point based on specific criteria. It seeks to locate and retrieve a particular entry in the database that matches the provided query conditions exactly. Point queries are typically used to fetch specific information about a particular entity or data point.

In summary, range queries retrieve data within a specified range of values, while point queries retrieve a single record or data point based on specific criteria.

The background section has provided an overview of the key concepts and components related to the research topic of this thesis. We discussed the selection of the JetBrains Xodus as a case system and highlighted components impacting performance-related challenges. The section has covered essential aspects such as different types of memory, key-value storage, copy-on-write B-trees, operation logs, garbage collectors, and queries. This foundational understanding sets the stage for exploring the related work in the field, where we will examine existing research and algorithms as well as potential evaluation techniques.

3

Related Work

In this section, we describe the scientific papers related to our research and discuss the advantages and disadvantages of the alternative approaches.

3.1 DGCC: A New Dependency Graph-based Concurrency

The authors of the (1) present a novel concurrency control protocol called DGCC (Dependency Graph-based Concurrency Control). The main idea behind DGCC is to separate concurrency control from execution. To achieve this, DGCC constructs dependency graphs for batches of transactions before executing them. By resolving contentions within the same batch prior to execution, the transactions are able to execute without dealing with contention, while still maintaining the same level of equivalence as serialized execution. This approach effectively utilizes multicore hardware and enables a higher degree of parallelism. In the paper the authors have demonstrated that DGCC outperforms state-of-the-art concurrency control protocols for high contention workloads, achieving up to four times higher throughput.

After the consideration of this approach, we decided not to utilize it in our concept due to the reasons related to the range query execution discussed below.

Let's take a look at the example with two transactions pictured in Figure 1 in (1). We can observe that operation C is independent while A and B are dependent. In Figure 3.1(a) Lock based protocol is depictured. The locking sequence for the first transaction would be C-A-B, while for the second one – D-B-A. this way we observe a classical deadlock at the stage A-B/B-A. In the databases, it is a common practice to have a deadlock manager to detect and resolve various locking issues. Looking in Figure 3.1(c) we can see that the DGCC approach suggests execution transactions in the same order (here - Txn1, Txn2) in

different threads. Due to the fact that it builds the dependency graph of updates between these transactions, it's possible to traverse this graph in parallel so that there will be no conflicts in transactions. So if we take Txn1 and Txn2, we can observe that 4 threads are created, thread 1 processes Txn1 and Txn2, thread 2 processes Txn1 and Txn2, thread 3 processes Txn1 and thread 4 processes Txn2. This way A is updated twice in thread 1, B is updated twice in thread 2, C and D are updated once. Therefore we receive A-B conflict but as the dependency graph is created, Txn1 is executed in two threads in parallel without any logical conflicts. The dependency graph of transactions is traversed in a way to avoid any conflicts.

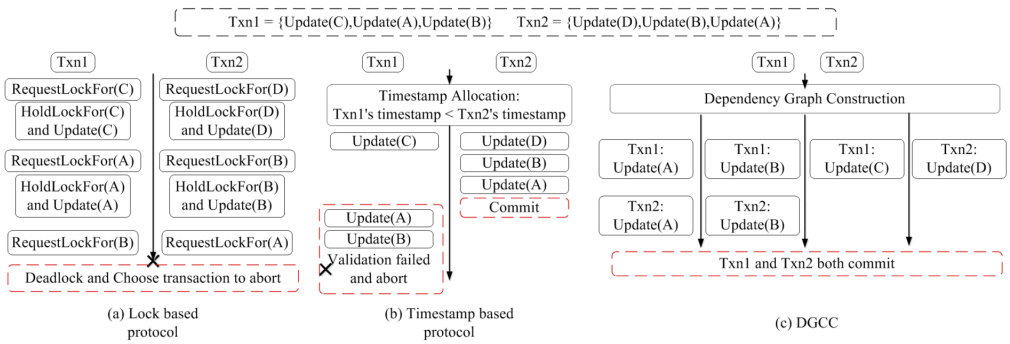

Figure 3.1: Figure 1 from the paper (1). The illustration of the basic idea of Graph-based Concurrency Control and its comparison with the two traditional concurrency control protocols – Lock-based and Timestamp-based.

The main drawback of this algorithm is the following: apart from the update operation presented there is also a possibility for the read operation to be executed. When the point query is executed, we have a clear understanding of how to resolve the conflicts with the help of the dependency graph. The problem arises at the point of range query as there it is not fully clear how to resolve the conflicts. The main disadvantage of this approach is the fact that we need to know in advance, before the execution of the transaction, what entities we will read.

An example to illustrate this problem can be the following: we need to count from 1 to 10, but instead of integer numbers as keys, we have real numbers. We cannot put an infinite number of keys in the graph and we don't know what we will read as we need to build the dependency graph before the execution of the transaction. This poses a big issue when performing a range query, therefore this algorithm is not suitable for our purposes as the range query is an essential functionality of the storage.

As a big advantage of this approach, we can mention the possibility of easily extending it to distributed transactional management.

3.2 Calvin protocol

In this section, we examine the contributions of the second alternative presented – the Calvin protocol (14). Calvin protocol is a protocol developed initially for the in-memory databases but afterward modified to be suitable for the disk databases.

Calvin is a transaction scheduling and data replication layer that addresses the challenges of distributed transactions. Calvin distinguishes itself from previous deterministic database system prototypes by offering support for disk-based storage, achieving near-linear scalability on a cluster of commodity machines, and eliminating single points of failure. Through its approach of replicating transaction inputs rather than effects, Calvin enables the support of multiple consistency levels, including Paxos-based (15) strong consistency across geographically distant replicas, without compromising transactional throughput. This innovative solution effectively reduces contention costs associated with distributed transactions and provides enhanced flexibility and reliability in a practical setting.

Calvin protocol is famous due to its capability to transform any non-transactional storage into a transactional one.

This alternative possesses a similar drawback, but despite it, this algorithm is used by Fauna DB (16) as they managed to modify it in a manner to avoid this issue, however, at a high price in terms of effort and resources (17). Initially, the algorithm had the same disadvantages as the DGCC algorithm described in the previous section, however, the researchers managed to overcome the range query issue in the following years. The main concern that we had when considering this algorithm was the high cost of development. The algorithm has been tuned and re-thought in the course of 3 years by a team of several highly skilled developers, and this seems not worth the trade-off in terms of effort and result.

3.3 Copy-on-write

Current implementation extensively uses Copy-on-write concept. With it we achieved high reading operations speed, which is unfortunately is not sufficient for the growing needs. However, The first idea which can arise when considering different approaches would be that we can still improve current concept further. The limitations of it are described below

and narrows down to the binary search for the key.
One logical page can take one or more file pages, in most cases one. We have B-trees in which key-value pairs are stored separately from the pages (nodes) of the B-tree itself, in different parts of the disk. In B-tree, we first get the address of the key, and after that go to the disk and get the page, from the page get the key and compare. If we had the keys inside the page, we would not have the need to perform this complex appoach to get the key. It would lead to less overhead as we wouldn't have to additionally go to the disk. In this case, we would be able to just directly get the key from the page and compare it with the target one as byte arrays, which turned out to be very fast from Java 11.
Currently, in benchmarks, both pages (nodes) of the trees and keys are stored on the same page of the cache and disk and the overhead is low. However, it is critical to note that in production they can be on very different pages and the performance will be significantly lower than in benchmarks' reports.
It could have been a promising change to keep it on the same page, but the is also an issue with this approach. Transactions are implemented via copy-on-write. Every time we change the data, we copy and re-write the whole path from the root to the leaf of the B-tree to maintain the isolation of transactions. If we store the keys inside as well, it will inevitably lead to the huge overhead on write. Therefore, this approach of storing keys on the page is assessed as not suitable.
The more specific reasons that lead to this overhead are the following. When pages are small, when the copy-on-write is performed we need to copy relatively small amount of data and put it on disk. When the page size increases due to the presence of the key on it, we need to copy way more data and write it, which causes big write overhead. As an example, let us present the following scenario: let's consider putting all entries in one transaction. The performance rate will be high as in benchmark, as there is no copy-on-write, just filling in the data from scratch, before the transaction it was empty. However, if we consider putting the same entries to separate transactions the speed will decrease significantly due to the copy-on-write effect.

Based on the all arguments stated above, the operation log concept is considered to be the most efficient and possessing the best trade-off. The basic concept of it is the following: we write all the transactions to log and on commit phase put it in the B-tree stored on disk. The snapshot isolation is maintained by reading the previous versions both from operation log and B-tree, for committed and removed operations.

To perform the evaluation of the proposed concept we use the Yahoo! Cloud Serving Benchmark (YCSB), described in detail in Section 5.1. YCSB is a widely used benchmarking tool for evaluating the performance of distributed key-value and cloud storage systems. While YCSB is a popular choice, there are several other benchmarking tools available that compete with YCSB in evaluating different aspects of system performance.

3.4 Transaction Processing Performance Council (TPC)

Transaction Processing Performance Council (18) is an organization that develops and maintains a suite of industry-standard benchmarks for evaluating the performance and scalability of transaction processing and database systems. TPC benchmarks provide a standardized framework for measuring and comparing the performance of different hardware and software configurations.
The TPC offers various benchmark specifications, each designed to evaluate different aspects of database system performance. Some of the well-known TPC benchmarks include:

1. **TPC-C:** TPC-C simulates an order-entry workload and measures the performance of transactional processing systems.

2. **TPC-H:** TPC-H is a decision support benchmark that simulates complex ad-hoc queries and measures the performance of database systems in data warehousing scenarios.

3. **TPC-E:** TPC-E simulates an online brokerage workload and measures the performance of transactional systems.

These benchmarks define standardized workloads, data sets, metrics, and reporting guidelines to ensure fair and consistent performance comparisons across different systems. The results are reported in terms of performance metrics such as transactions per second (TPS), response times, and throughput.
TPC benchmarks have their own advantages, such as standardized workloads, established industry recognition and support of a wide range of performance aspects. However, TPC benchmarks are primarily designed for evaluating the performance of traditional SQL-based relational database management systems (RDBMS). The workload models and metrics defined by TPC benchmarks are specifically tailored to measure the transaction processing and decision support capabilities of SQL-based database systems.
However, given that TPC benchmarks are specifically designed for SQL-based systems,

they may not be directly applicable or suitable for evaluating the performance of key-value stores.

Unlike TPC, YCSB provides workload models specifically for key-value stores and NoSQL databases. These benchmarks are designed to assess the performance of non-SQL databases and can be more suitable for evaluating their specific characteristics and capabilities. YCSB is flexible and provides wide extensibility, allowing users to define custom workloads and adapt the benchmark to their specific use cases. Moreover, the YCSB benchmark is easy to use as it focuses on a simple key-value workload model, which can be advantageous for evaluating the basic read-and-write performance of distributed storage systems.

3.5 NoSQLBench

Another possible alternative is the NoSQLBench tool (19). NoSQLBench is a benchmarking tool designed specifically for evaluating the performance of NoSQL databases. It provides a framework for creating and executing realistic workloads against NoSQL database systems, enabling users to measure and compare their performance in various scenarios.
NoSQLBench is specifically designed for benchmarking NoSQL databases. It provides a more specialized approach compared to general-purpose benchmarking tools. NoSQLBench also allows users to define and customize their own workloads to simulate real-world scenarios and access patterns.
NoSQLBench benefits from an active community of users and contributors, providing support, documentation, and ongoing development. This fosters knowledge sharing, collaboration, and the continuous improvement of the tool. The open-source nature promotes transparency, community collaboration, and flexibility in adapting the tool to specific needs.
It possesses a decent alternative to the YCSB. However, there are some reasons behind the final choice being the YCSB. YCSB is a widely adopted and established benchmarking tool for key-value storage systems. It has been used extensively in industry and academia, and its usage has become a de facto standard for benchmarking NoSQL databases, including key-value stores. Moreover, YCSB supports specifically a wide range of key-value storage systems, including popular ones like Apache Cassandra, MongoDB, Redis, Riak, and more. It provides built-in examples of usage, making it easy to create a custom client for the new database using a unified framework.

4

Design

4.1 Overview

As a base approach to building the new concept, we take the paper "High Performance Transactions in Deuteronomy" written by Levandoski, J., Lomet, D., Sengupta, S., Stutsman R. and Wang, R.(2).
The authors introduce a new mechanism called "Deuteronomy" to optimize the transaction processing layer in the database system. Deuteronomy aims to reduce transaction overheads and maximize concurrency without sacrificing data consistency or durability.
The key principles of Deuteronomy involve leveraging a combination of techniques such as optimistic concurrency control, timestamp order, latch-free buffer management and data structures (20). These techniques are designed to minimize the contention between transactions and efficiently handle both read and write operations.
The main feature of the Deuteronomy architecture is the decomposition of the database kernel functionality into two interacting components such that each one provides useful capability by itself. The idea is to enforce a layered separation of duties where a transaction component (TC) provides concurrency control and interacts with one or more data components (DC) providing data storage and management duties (access methods, cache, stability).
The TC consists of three main components: (1) MVCC component to manage the concurrency control logic; (2) version manager to manage the operation log as well as cached records read from the DC; (3) a TC Proxy that lives beside the DC and whose job is to submit committed operations to the DC.
The DC maintains a database state, in the current case presented in the form of a B-tree.
Overall, the paper provides detailed insights and evaluations of the approach, highlighting

its potential benefits for high-performance transaction processing. We aim to adapt and enhance this approach and develop an extended algorithm suitable for our case of key-value storage.

4.2 Concept design

The main idea of the new approach is to use the operation log for the implementation of the transactions instead of copy-on-write B-trees. We write all the transactions to log to put them in the B-tree on the commit phase and maintain the snapshot isolation by reading the previous versions from the operation log both from log and B-tree, for committed and aborted transactions.

The concept revolves around implementing a clear and structured division of responsibilities, ensuring that a transaction component handles concurrency control and recovery while interacting with a data component responsible for data storage and management tasks such as access methods, caching, and stability. The TC has no knowledge of the specifics of data storage, just as the DC lacks an understanding of transactional functionality and primarily functions as a key-value store (2).

The key components of the TC are MVCC (multi-version concurrency control, (21)) component to manage the concurrency control logic and a version manager to manage the operation log. The DC maintains the database state.

The high-level architecture is presented in Figure 4.1, and more detailed architecture is shown in Figure 4.2.

4.3 Components design

In this section, we elaborate more on each component and discuss what fields are required in each component from the design-specific perspective.

4.3.1 MVCC component level

In papers (22) and (21) it was confirmed that using multi-version concurrency control (MVCC) is beneficial in eliminating read-write conflicts by reading a version earlier than an uncommitted writer's version.

MVCC controller is represented with a concurrent hash map with primitive long keys – the hash code of the real key which is stored in the operation log. A single entry contains:

1. Hashcode (key);

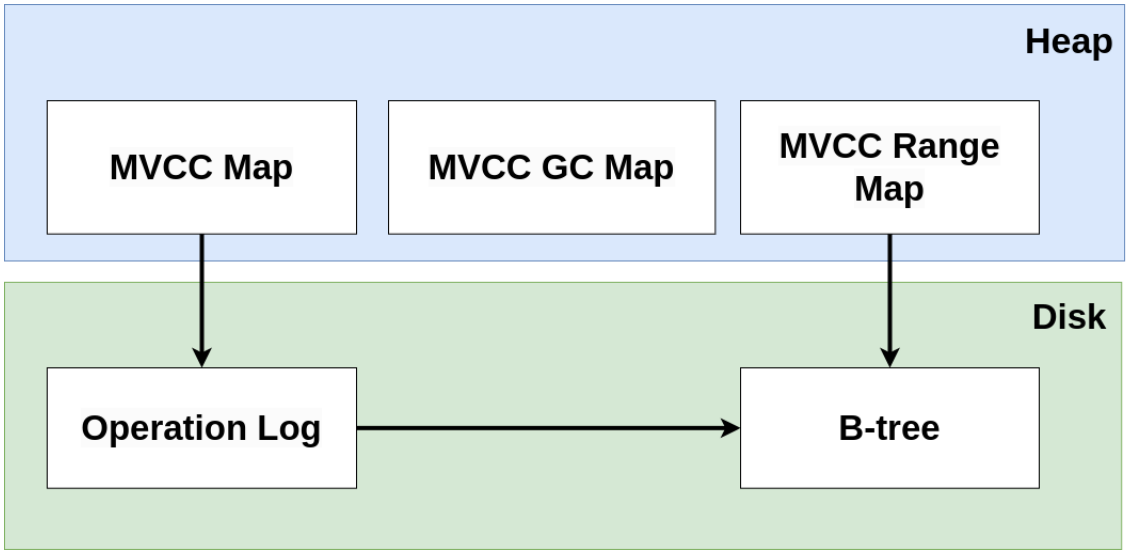

Figure 4.1: High-level architecture of the concept components. MVCC level includes MVCC hash map, MVCC range hash map, and MVCC garbage collection map located in Heap. The operation log level includes the Operation log hash map, and the Data level is represented by B-tree located on Disk.

2. Maximum transaction ID that reads this key;
3. Concurrent linked queue of links to all the operations on this key stored in the operation log;

All those fields are required for successful algorithm utilization. Let's consider the queue of links to operations on this key stored in the operation log (3). Each element in the queue represents an operation reference with the following fields:

1. Operation address in the OL;
2. Snapshot ID;
3. Key hashcode;
4. Transaction state (in progress, reverted, committed), countdown latch;

All those fields are required for successful algorithm utilization. In the implementation, the state is represented with the wrapper containing a countdown latch. As it ensures the correct behavior for the multi-threaded environment this mechanism is required, however, the implementation may vary.

When the interface level of the MVCC component is accessed either a Write or Read

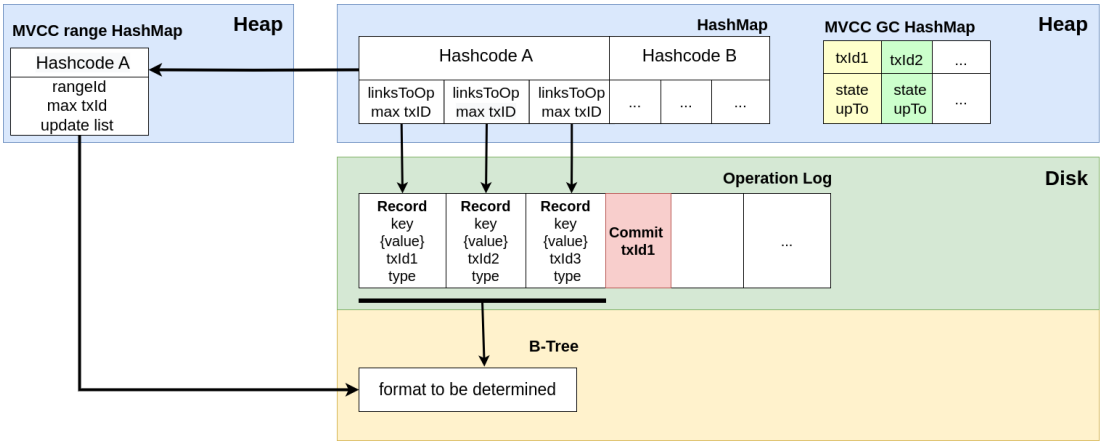

Figure 4.2: Low-level architecture of the concept components. It provides a more detailed overview of the components' structure and interaction.

transaction can be initiated. Transaction object should contain snapshot id and type (read/write) fields. The definition of snapshot id belongs to the MVCC component level, it is global transactions id counter which is incremented on write transactions only. This way we replace the transaction id with the snapshot id as read transactions change nothing in data, meaning that there is no need for unique transaction ids for read transactions for conflict resolutions. Therefore if we are talking about transaction id, we assume the snapshot id.

To perform the garbage collection algorithm described in Section 4.4.3 we need scalable concurrent map supporting navigation methods returning the closest matches for given search target operations.

A single entry contains:

1. Snapshot ID (key);
2. Transaction State (in progress, reverted, committed);
3. Id, up to which starting from snapshot id all the transactions are in the committed state, initiated as -1;

To perform the range query algorithm described in Section 4.4.5 we need scalable concurrent map supporting navigation methods returning the closest matches for given search target operations. A single entry contains:

1. Hashcode (key);
2. Maximum transaction ID that reads this key;

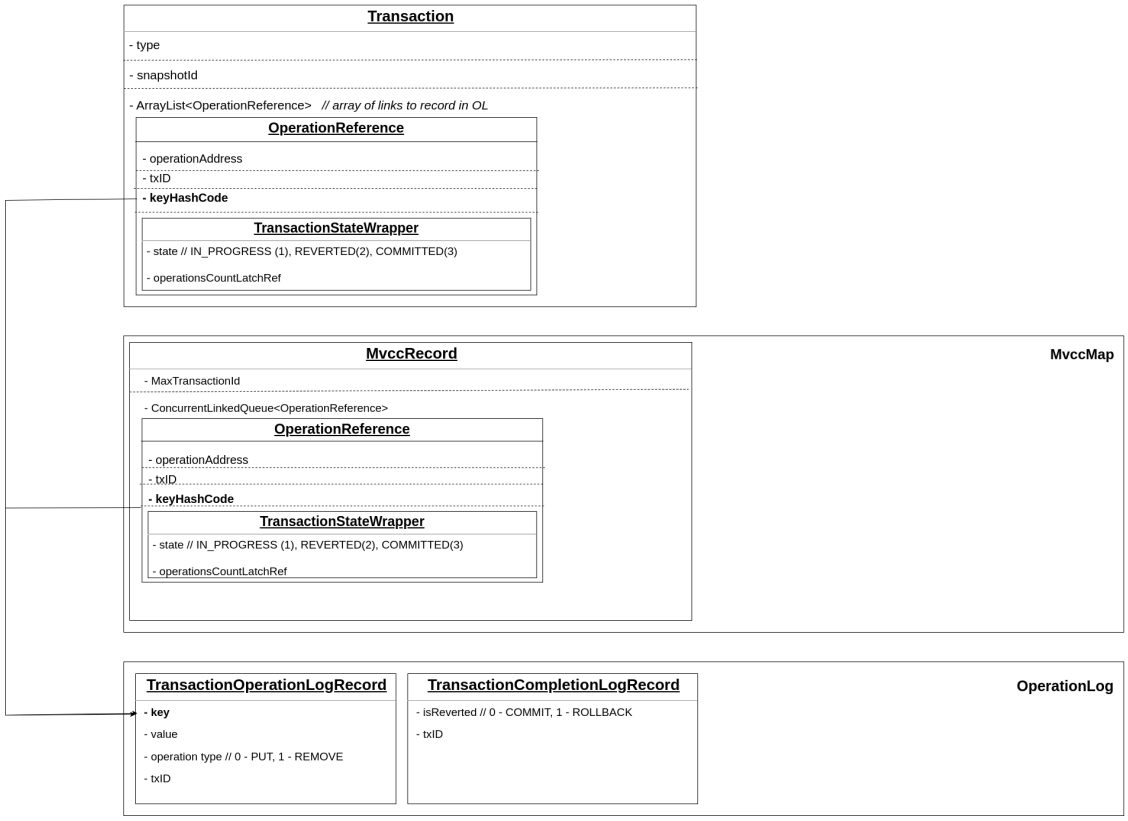

Figure 4.3: OL and MVCC components. This diagram provides a detailed overview of the required fields in each component.

3. Update list;

4. Range ID;

As with MVCC with record versioning, range MVCC maintains a hash table, providing access to the range mapped to the particular key. For each range, we maintain a latch-free list of updates, essentially just the id of the transaction performing the update to the range, plus a last read time that provides a barrier preventing updates to a range being read with an earlier time, thus preventing phantoms.

4.3.2 OL component level

The operation log is represented with a concurrent map supporting navigation methods. The map is sorted according to the natural ordering of its keys. A single entry contains:

1. Record address (key);

2. Log record;

There are two possible types of the log record. The first one is the **operation record,** which contains key, value, operation type (put/delete) and transaction id. This record indicates a particular operation occurred. The second type is the **special record**, which indicates the completion/abortion of the transaction and contains the transaction id field.

The detailed overview of the OL and MVCC components is presented in Figure 4.3.

4.3.3 Tree level

The final representation of the Tree level is yet to be determined. For now to mock the complete algorithm execution the concurrent navigable set was used, and the next step is to determine the final representation of the B-tree.

4.4 Algorithm design

Read and write operations are the core components of any transaction management system. Below we suggest a new concept design for the read and write operations, and these algorithms and the components discussed above are the key takeaways of the research.

4.4.1 Transaction read algorithm

Overview: read-transaction starts, and it aims to find a record corresponding to a certain key. The operation record is read by going to the records list and choosing the maximum ID smaller than the ID of the current transaction. This way the reading transaction always sees the valid data snapshot. If the record is not found in the OL (i.e. it is committed already) it is possible to read it from the B-tree, where it was moved on the commit action.

The pseudo-code for the algorithm for reading the operation associated with the specific key is presented below in Figure 4.4.
The algorithm begins by calculating the hash code of the key and retrieving the corresponding MVCC record. The algorithm then compares the current transaction's snapshot id with the maximum transaction id stored in the MVCC record, updating it if necessary. Next, the algorithm searches for the most recent relevant operation reference for the target key within the MVCC record. If a relevant operation reference is found, it retrieves the corresponding operation from the operation log. If no matching operation is found within

```
function read(currentTransaction,  key):
    hash = calculateHashCode(key)
    mvccRecord = mvccMap.get(hash)

    if mvccRecord is not null:
        if currentTransactionId > mvccRecord.maxTransactionId:
            mvccRecord.maxTransactionId = currentTransactionId

        maxMinValue = 0
        targetEntry = null
        maxTxId = mvccRecord.maxTransactionId

        if mvccRecord.queue.isNotEmpty():
            for ref in mvccRecord.queue:
                if ref.transactionId >= maxMinValue and ref.transactionId < maxTxId:
                    if ref.state != REVERTED:
                        targetEntry = ref
                        maxMinValue = ref.transactionId

        if targetEntry is null:
            return tree.search(key)
        else:
            targetOperation = operationLog.getOperation(targetEntry.operationAddress)

            if targetOperation is null:
                return tree.search(key);

            if targetOperation.type == REMOVE:
                return null

            if targetOperation.key == key:
                return targetOperation.value
            else:
                references = []

                for ref in mvccRecord.queue:
                    if ref.transactionId < maxTxId:
                        references.append(ref)

                references.sort(reverse=True, key=lambda ref: ref.transactionId)

                for ref in references:
                    operation = operationLog.getOperation(ref.operationAddress)
                    if operation.key == key:
                        return operation.value
    else:
        return tree.search(key)
```

Figure 4.4: The pseudo-code for the algorithm for reading the operation associated with the specific key.

```
function addToLog(tx, address, key, value, operationType):
    keyHashCode = xxHash.hash(key)
    operationReference = OperationReference(address, tx.snapshotId, keyHashCode)
    tx.addOperationReferenceEntryToList(operationReference)
    operationLog.put(address, TransactionOperationLogRecord(key, value, operationType,
     snapshot))
```

Figure 4.5: The pseudo-code for adding the record to the log.

the MVCC records, the algorithm falls back to performing a search for the key within the tree data structure.

In the case of a REMOVE operation, it returns null to indicate that the key has been removed. If the target operation's key matches the given key, the associated value is returned. If the key does not match the target operation's key, the algorithm creates a list of operation references with transaction ids lower than the maximum transaction id. It then iterates through the sorted list in order to find a matching operation record.

4.4.2 Transaction write algorithm

The put/delete operations are performed in two phases. The first phase is responsible for putting the target operation to OL record only, the second one – for committing the transaction including the rest of the logic of components interaction, all the necessary checks, and cleaning up the space.

To add the record to the log we use the following simple algorithm, presented in pseudo-code in Figure 4.5. Note that the operation type can be either PUT or REMOVE, and is used when creating a new record. The pseudo-code for committing the transaction is presented below in Figure 4.6.

First, the algorithm checks if the transaction type is WRITE and if there are operations to process. If so, it sets the state to IN_PROGRESS. Depending on the number of operations in the transaction, a latch may be set to control synchronization. For each operation in the transaction, the code associates the operation with the transaction state, creates an MVCC record and adds the operation reference to its queue. It checks the transaction snapshot id against the maximum transaction id in the MVCC record. If the snapshot id is smaller, it marks the transaction as REVERTED and throws an exception to avoid an inconsistent state.

The algorithm updates the snapshot id if necessary, sets the transaction state as COMMITTED, and puts a special record to the operation log to indicate a successful commit.

To maintain system performance, the code performs cleanup operations if certain limits

are exceeded. The MVCC garbage collector is invoked to clean up the transaction map if it exceeds a specified size, and the log garbage collector cleans up the operation log if it exceeds a defined limit.
By following this approach, the code ensures that transactions are correctly committed, maintains data consistency, and performs necessary cleanup operations to optimize system performance.

4.4.3 MVCC garbage collector algorithm

As we aim to create a lightweight and efficient garbage collector for the MVCC controller, we need to develop a new algorithm. After a series of iterations and assessments, the following algorithm was approved.
Let's define ConcurrentSkipListMap as the notation for the map, which should be a scalable concurrent map supporting navigation methods returning the closest matches for given search target operations.

1. In heap create a ConcurrentSkipListMap<Long, TransactionGCEntry> with the key being a snapshot id of objects TransactionGCEntry with the following parameters:

   - transaction id (tx_id)
   - state
   - up to transaction (up_to), initialize as -1

   The map is a part of the MVCC controller and should be integrated into the main algorithm of the MVCC controller, coordinating the read/write operations of the transactions.

2. When a transaction starts, it is added to the map with the status being IN_PROGRESS

3. After the transaction is committed, depending on its status, its status in the map should be changed to COMMITTED/ABORTED. The flow to be performed on the transaction commit is the following:

   (a) For the current snapshot id find the maximum transaction id in the map, for which all the previous transaction ids are present and have status COMMITTED/ABORTED (max_id) and remove operations from records with txId up to this maximum transaction id, but without it.

```
function commitTransaction(transaction):
    if transaction.type == WRITE:
        wrapper = TransactionStateWrapper(IN_PROGRESS)

        # wrapper and latch logic is not strict and could be changed, but is required for multi-threaded environments
        if transaction.operationLinkList.size() > 10:
            wrapper.initLatch(1)

        for operation in transaction.operationLinkList:
            operation.wrapper = wrapper
            transactionSnapId = transaction.snapshotId
            mvccRecord = mvccRecordCreateAndPut(operation, transactionSnapId)

            if transactionSnapId < mvccRecord.maxTransactionId:
                wrapper.state = REVERTED
                transactionsGCMap.get(transactionSnapId).stateWrapper.state = REVERTED
                recordAddress = address.getAndIncrement()
                operationLog.put(recordAddress, TransactionCompletionRecord(ROLLBACK, snapshotId))
                latchRef = wrapper.operationsCountLatchRef

                if latchRef is not null:
                    latchRef.countDown()
                    wrapper.operationsCountLatchRef = null

                mvccRecord.linksToOperationsQueue.remove(operation)
                throw Exception()
            while true:
                txSnapId = transaction.snapshotId
                currentSnapIdFixed = snapshotId

                if currentSnapIdFixed < txSnapId:
                    snapshotId = txSnapId
                    break
                else:
                    break

        wrapper.state = COMMITTED
        transactionsGCMap.get(transaction.snapshotId).stateWrapper.state = COMMITTED
        latchRef = wrapper.operationsCountLatchRef

        if latchRef is not null:
            latchRef.countDown()
            wrapper.operationsCountLatchRef = null

        recordAddress = address.getAndIncrement()
        operationLog.put(recordAddress, TransactionCompletionRecord(COMPLETED, snapshotId))

        if transactionsGCMap.size() > transactionsLimit:
            lock = ReentrantLock()
            if lock.tryLock():
                try:
                    mvccGarbageCollector.clean()
                finally:
                    lock.unlock()

        maxMinId = mvccGarbageCollector.findMaxMinId()

        if operationLog.size() > operationsLimit and maxMinId is not null:
            service = Executors.newCachedThreadPool()
            logGCThread = service.submit():
                logGarbageCollector.clean()
            logGCThread.get()

            handleMinMaxThread = service.submit():
                mvccGarbageCollector.handleMaxMin()
            handleMinMaxThread.get()
```

Figure 4.6: The pseudo-code for committing the transaction.

(b) If the interval where some ids are NOT sequential but present and COMMITTED/ABORTED is found, remove a subsequence from starting id to max_id - 1. Also, "merge" together sequential COMMITTED/ABORTED operations, leaving the last COMMITTED unmerged. Remove deleted transactions' ids from GC MVCC hashmap.
Example: 1-7 are COMMITTED, 8 is missing, 9-11 are COMMITTED, 12 is the last IN_PROGRESS, max_id = 7 -> remove 1-6 (up to 7-1=6) and 10, and put up_to field in 9 to 10 ("merge").

(c) If we find an interval where ids are sequential but some are IN_PROGRESS, remove a subsequence from start to max_id-1. "Merge" together sequential COMMITTED/ABORTED operations, leaving the last COMMITTED unmerged. Also, remove deleted transactions' ids from GC MVCC hashmap.
Example: 1-7 are COMMITTED, 8-9 are IN_PROGRESS, 10-13 are COMMITTED, 14 is the last one IN_PROGRESS, max_id = 7 → remove 1-6 (up to 7-1=6), and put the up_to field in 10 to 12 (as 14 has a reference to 13).

4. When the links to the operations queue of the MVCC record is empty, this record can also be deleted.

The pseudocode is presented in Figure 4.7.

```
void clean(snapshotId, mvccHashMap,transactionsGCMap) {
   maxMinId = findMaxMinId();
   if (maxMinId != null)
     if (maxMinId >= snapshotId)
            throw new ExodusException();

     removeUpToMaxMinId(); // remove references with txId < maxMinID

     activeOrEmptyTransactionsIds =
        findMissingOrActiveTransactionsIds(); // for missing or IN_PROGRESS txIds

     removeBetweenActiveTransactions(); // remove references with txId between
IN_PROGRESS or missing txIds and update corresponding up_to fields
}
```

Figure 4.7: The pseudocode for the MVCC GC algorithm

Assumptions:

1. Max_id should always be smaller or equal to the current snapshot id. If it is bigger, an exception should be thrown.
2. It is assumed that in transactions GC map ids are not sorted.
3. Max_id should never be removed.

4.4.4 OL garbage collector algorithm

Operation log garbage collection is an essential part of the algorithm and should be performed after MVCC garbage collection.
Approach: the iteration of the OL is performed, on the current step we are reading the OperationReference A with key K. We take this key K and access the MVCC map to find the MVCC record by key's hashcode. After that we search for the latest operation in the COMMITTED state, corresponding to this key (def. X). After that, we check if the snapshot id of X (def. S) is equal to the current or not.

Algorithm (*):

1. For the record R in OL:
   (a) Current transaction id = T;
   (b) Access the MVCC map:
      i. If there is nothing for this record R in map throw exception and break;
      ii. The record in the MVCC map is found.
      is $T == S$ (snapshot id of the last committed operation)?
      no: delete operations references from the queue of this record;
      yes: copy operations references to the tree and delete them from the queue of record;
      iii. if $S > T$: delete operations references from the queue of this record;
      iv. if $T > S$: throw exception and break;

Assumption: for the maximal minimal snapshot id of this record – if there is no link to it in the OL, we can safely remove it.
The garbage collection process will be performed in two phases of iteration:

1. Find all special records in OL;
2. Due to special records found in step 1 we have the knowledge of which transactions are committed or aborted, so we can perform algorithm (*).

Note: from the MVCC map we remove only operations references from the queue, not the entire record. The reasoning for it is the following: for each key, there can be multiple transactions, each of which might need pieces of these operations. If we delete the entire MVCC record, IN_PROGRESS transactions won't be able to see the data.

The introduction of the OL GC will involve some updates to the MVCC GC. Only if the queue of operations references of a particular record is empty, we remove the record from MVCC map. Deletion should be performed in a thread-safe manner as multiple threads can add, and only one can delete the reference but while the deletion is happening some other thread can write. However, when we delete operation reference from the queue (not MVCC record) we don't need multi-threading, as we work within the try-lock.

4.4.5 Range query algorithm

The **point query** algorithm is highly straightforward at this point as if we need to obtain the value corresponding to the certain target key, it can be solved by calling the read() method. However, the range query is more complicated and involves some additional steps.

Overview: DC uses key distribution to provide partitioning information. The protocol for defining logical key ranges defined in (2) involves the following steps:

1. TC requests the B-tree (DC) to partition the keys it manages, specifying the desired number of disjoint partitions.
2. The DC responds to the TC, providing the partitioning information. The TC's record manager, which utilizes the MVCC component, uses this partitioning to assign resource ids to the ranges.
3. Subsequent concurrency control requests use these resource ids to identify the specific range resource that contains a given key.

In the reference paper the ranges are disjoint, however it's not a strict requirement, and for our algorithm we will use joint ranges. Ranges are joint if the same key can occur in two ranges. When constructing the MVCC Range HashMap, txId is added in either

range $R_1...R_n$ or ,$W_1...W_m$ if they are joint on this key, there is no preference which one. Following this logic, there will be one range per key in the MVCC Range HashMap. This way we have a pair key-range, therefore, when we do the update, the txId from the update List for this key can occur only in this range. However, we need to take into account the fact that ranges are joint when we perform the merge of range queries with the changes.

Range merge with joint ranges.
R_1, R_n, W_1, W_m - keys for corresponding ranges, $R_1 < R_n, R_1 \leq W_1, W_1 < W_m$. If two ranges $R_1...R_n$ and $W_1...W_m$ overlap, we use actually tree ranges at the merge step (Figure 4.8):

$$R' = R_1...W_1 \text{ - put txId of the first range}$$
$$R'' = W_1...R_n \text{ - put txId of both ranges}$$
$$R''' = R_n...W_m \text{ - put txId of the second range}$$

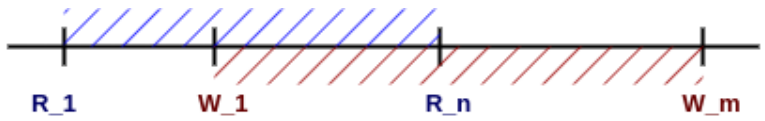

Figure 4.8: Example of ranges intersection

Apart from that the algorithm for the range query remains as described in initial paper.

5

Evaluation

5.1 Evaluation approach

To perform the evaluation of the developed algorithm and showcase its performance on a real-world example, we will use the Xodus database as the target system. To perform the evaluation of the algorithm we will use the Yahoo Cloud Serving Benchmark (23).
The Yahoo Cloud Serving Benchmark (YCSB) is a popular open-source benchmarking tool developed by Brian F. Cooper and his team at Yahoo Research. YCSB is designed to measure the performance and evaluate the scalability of different database and storage systems. It provides a standardized workload and a set of client programs for simulating various real-world application scenarios, such as social media, e-commerce, and IoT. The main reason for using YCSB is to assess the performance characteristics of database under realistic workloads. By running YCSB tests, developers and users can compare the performance, scalability, and efficiency of different systems, identify potential bottlenecks, and optimize their configurations accordingly. YCSB's flexibility and extensibility make it a suitable tool for researchers, developers, and system administrators in the field of database and storage systems evaluation.
To perform the evaluation we will run the set of workloads both on the previous implementation of the transactional components and the newly developed ones.
We will use the following core workloads provided by the benchmark:

Workload A: Update heavy workload
This workload has a mix of 50/50 reads and writes. An application example is a session store recording recent actions. Updates in this workload do not presume you read the original record first. The assumption is all update writes contain fields for a record that

already exists; oftentimes writing only a subset of the total fields for that record.

Workload B: Read mostly workload
This workload has a 95/5 reads/write mix. As with Workload A, these writes do not presume you read the original record before writing to it.

Workload C: Read only workload
This workload is 100% read.

Workload D: Read latest workload
In this workload, new records are inserted, and the most recently inserted records are the most popular.

Workload F: Read-modify-write
In this workload, the client will read a record, modify it, and write back the changes. This workload forces a read of the record from the underlying datastore prior to writing an updated set of fields for that record. This effectively forces all data stores to read the underlying record prior to accepting a write for it.

5.2 Results and analysis

To evaluate whether the number of records has a significant impact on the overall performance we also ran workload A on both versions with the number of records [40000, 60000, 80000, 100000]. The results are presented in Figure 5.1 and from it it can be determined that with the increase of the total records number from 40000 to 100000 the latencies and overall runtime remains stable, without strong fluctuations and the new concept clearly over-performs the old one.

To evaluate the suggested new concept performance in different scenarios we executed the workloads A, B, C, D, and F on both versions with the same number of total records (40000). The highlights are presented in Figure 5.2 and discussed in detail below.
Let's analyze the benchmark results for different metrics, focusing on the average latency for reads and writes, overall runtime and throughput and draw conclusions based on the goals of each workload.

Workload A (Update heavy workload)
AverageLatency for reads is 114.27 ms compared to 29.09 ms with ((114.27 - 29.09) / 114.27) * 100 = 74.53% improvement. AverageLatency for writes is 253.66 ms compared to 34.52 ms with ((253.66 - 34.52) / 253.66) * 100 = 86.40% improvement. The old implementation had a runtime of 564 ms with a throughput of 1773.05 ops/sec. The new implementation had a significantly improved runtime of 59 ms with a higher throughput of 16949.15 ops/sec.
Conclusion: The new implementation shows significant improvements in both read and write latencies compared to the old implementation for the update-heavy workload. This suggests that the new implementation handles updates more efficiently.

Workload B (Read mostly workload)
AverageLatency for reads is 86.04 ms compared to 16.99 ms, percentage improvement is ((86.04 - 16.99) / 86.04) * 100 = 80.26%. AverageLatency for writes is 503.84 ms compared to 91.59 ms, Percentage improvement: ((503.84 - 91.59) / 503.84) * 100 = 81.83%.
The old implementation had a runtime of 514 ms with a throughput of 1945.53 ops/sec. The new implementation had a reduced runtime of 43 ms and a higher throughput of 23255.81 ops/sec.
Conclusion: The new implementation exhibits notable improvements in read latency compared to the old implementation for the read-mostly workload. Write latencies are relatively higher in both implementations, which is expected, still demonstrating considerable reduction compared to the old implementation.

Workload C (Read only) AverageLatency for reads in the old implementation: 76.23 ms compared to 19.53 ms in the new one with the percentage of ((76.23 - 19.53) / 76.23) * 100 = 74.38%. As there are no write operations in this workload, we cannot evaluate the average latency for it. The old implementation had a runtime of 493 ms with a throughput of 2028.40 ops/sec. The new implementation had a decreased runtime of 46 ms and higher throughput of 21739.13 ops/sec.
Conclusion: The new implementation demonstrates a significant reduction in read latency compared to the old implementation, emphasizing its improved performance in handling read-intensive workloads.

Workload D (Read latest workload) AverageLatency for reads in the old implementation is 72.95 ms compared to 18.40 ms, percentage improvement: ((72.95 - 18.40) / 72.95)

* 100 = 74.75%. AverageLatency for writes in the old implementation is 1927.98 us in comparison to 162.74 ms, highly significant percentage improvement: ((1927.98 - 162.74) / 1927.98) * 100 = 91.55%. The new implementation demonstrates improved performance in terms of runtime and throughput – a runtime of 543 ms vs 52 ms and a throughput of 1841.62 ops/sec vs 19230.77 ops/sec.
Conclusion: The new implementation shows improvements in both read and write latencies. It indicates better efficiency in handling the most recently inserted records.

Workload F (Read-modify-write)
AverageLatency for reads in the old implementation is 77.86 ms vs 16.81 ms in the new implementation. Percentage improvement: ((77.86 - 16.81) / 77.86) * 100 = 78.40%. AverageLatency for writes in the old implementation is 229.79 ms vs 38.16 ms in the new one. Percentage improvement: ((229.79 - 38.16) / 229.79) * 100 = 83.37%. The old implementation had a runtime of 668 ms with a throughput of 1497.01 ops/sec. The new implementation had a reduced runtime of 65 ms and higher throughput of 15384.62 ops/sec.
Conclusion: The new implementation achieves notable reductions in both read and write latencies compared to the old implementation for the read-modify-write workload. This indicates improved efficiency in handling read and write operations that involve modifying existing records.

To evaluate whether the new concept overperforms the old one in a multi-client environment, we also ran workload A on both versions with the number of threads [1, 2, 4]. Complete results are presented in Table 5.1. We can observe that in the old implementation, the increase of overall runtime was +26.7% (639 * 100 / 504 - 100) and +64.3% (828 * 100 / 504 - 100) for switching from 1 to 2 and 4 threads correspondingly, while for the new implementation, the increase of overall runtime was +2% and +4% for the same switch from 1 to 2 and 4 threads respectively.

Summary
The new implementation outperforms the old implementation in terms of overall run time and latency improvements, especially for read-intensive workloads, resulting in enhanced performance and responsiveness.

	1	2	4
[READ], AverageLatency(ms)_old	89.76923077	186.5838264	308.3878788
[READ], AverageLatency(ms)_new	23.69548134	55.05927342	120.2519841
[UPDATE], AverageLatency(ms)_old	236.3645833	480.9918864	616.6356436
[UPDATE], AverageLatency(ms)_new	30.31568228	50.90775681	303.6149194
[OVERALL] RunTime(ms)_old	504	639	828
[OVERALL] RunTime(ms)_new	49	50	51

Table 5.1: Benchmark results for the workload A, number of threads = [1, 2, 4], number of records = 50000

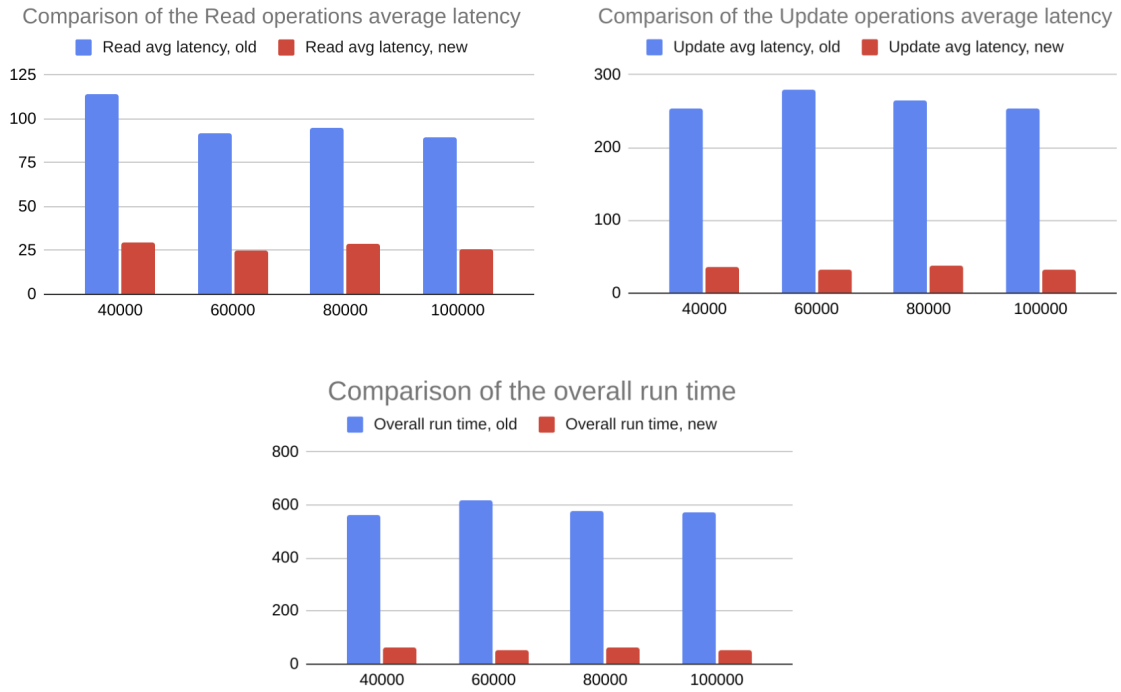

Figure 5.1: Summary of the results for workload A and different number of total records. X-axis denotes the number of records, the y-axis is time in milliseconds; the blue color denotes the old implementation, and the red is the new implementation.

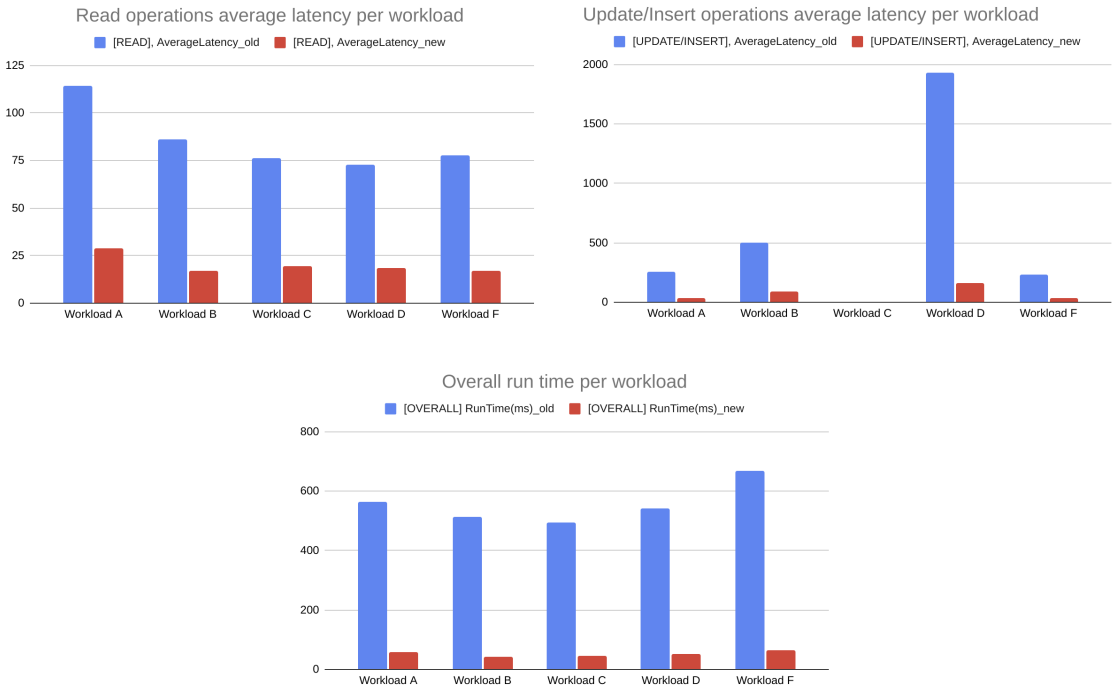

Figure 5.2: Summary of the results per workload, number of records = 40000, number of threads = 1. X-axis denotes workload, the y-axis is time in milliseconds; the blue color denotes the old implementation, and the red is the new implementation.

6

Threats To Validity

6.1 Internal Validity

Implementation Issues. The successful implementation of the new transaction management algorithm is crucial for internal validity. Any errors, bugs, or limitations in the implementation may affect the validity of the results. It is important to ensure that the implementation accurately reflects the proposed concept and the proper process of QA verification should be initiated.

6.2 External Validity

The external validity of the findings depends on the representativeness of the sample used in the study. The sample chosen should reflect real-world scenarios and make the generalizability of the results possible. The implementation and evaluation of the results were performed on the basis of the Xodus database to showcase the real-world example case, therefore, if the same concept will be implemented on the basis of another key-value storage, the results might vary due to different implementation details, algorithms utilized, and starting points. However, the key takeaway of the paper is the algorithm, not the implementation, and it is generalized for the case of key-value storage and can be evaluated against other algorithmic approaches producing the same dynamics.

6.3 Construct Validity

The reliability and consistency of the measures used to assess transaction management are crucial for construct validity. So far we performed the benchmark both in single-thread and multi-thread environments to evaluate the performance. There is a possibility to also

perform an assessment of the data consistency with the same benchmark, it is planned as the first point for future work and awaits only the required equipment.

6.4 Conclusion Validity

The sample size of the total number of records may not be sufficient to detect small but meaningful effects or differences. We were limited by the execution time of the benchmark, however, there are also future plans to perform the same benchmarking procedure on a database size of 100 million records. This sample size should be more than sufficient to prove the validity of the drawn conclusions.

7

Conclusion and discussion

7.1 Conclusion

The results obtained from the benchmarking process (Section 5.2) allowed us to evaluate the performance characteristics of the database under various workloads. The new transactional concept demonstrated significant improvements in terms of average latency, overall runtime, and throughput compared to the previous implementation across different workloads. In a single-client environment, the average latency for read operations was improved by an average 75%, and the average latency for write operations was improved by an average 82%. The overall runtime boost achieved is $\approx$ 10x.
Overall, the evaluation results demonstrate that the new transactional concept enhances the performance and responsiveness of the database, particularly in read-intensive scenarios, providing a solid foundation for future efficient transaction management.

7.2 Discussion

The evaluation of the new transactional concept has provided insights into its performance and potential benefits for enterprise databases. However, there are several aspects that require further consideration and future work to ensure the success and practicality of the new concept.
The first one is the determination of the final version of the tree component. As a temporary solution, we created a mock component, however, to receive the complete picture of the algorithm implementation we need to agree on the final version of the tree, and the next step would be to transform the current representation into the representation of the B-tree.

Also, a good point for potential future work would be the execution of the benchmark with a larger number of records, such as 100 million records. This will help assess the scalability and performance of the new concept under more realistic and demanding conditions. It will provide insights into how the database system handles larger volumes of data and transactions, enabling better preparation for real-world scenarios.

Moreover, as the new transactional concept will handle customer data, it is crucial to establish a robust quality assurance process. This process should encompass comprehensive testing, including stress testing, edge case analysis, and verification of consistency across different database operations. Rigorous QA measures will provide confidence in the reliability and accuracy of the new concept.

As a point of long-term future work, we can highlight the exploration of the fault tolerance and scalability of the new transactional concept in distributed environments. Distributed systems introduce additional challenges, such as data replication, consistency across multiple nodes, and fault recovery. Investigating how the new concept performs in these scenarios will help determine its suitability for deployment in large-scale distributed systems.

By addressing these points in future work, the algorithm can be further refined, ensuring its readiness for real-world deployments. Continued research and development will enable the new concept to overcome challenges and meet the evolving needs of enterprise databases.

References

[1] C. Yao, D. Agrawal, P. Chang, G. Chen, B. C. Ooi, W. Wong, and M. Zhang. **DGCC:A New Dependency Graph based Concurrency Control Protocol for Multicore Database Systems**. 2015. iii, 11, 12

[2] J. Levandoski, D. Lomet, S. Sengupta, Stutsman R., and R. Wang. **High Performance Transactions in Deuteronomy**. 2015. 1, 2, 3, 17, 18, 29

[3] D. Ongaro and J. Ousterhout. **In Search of an Understandable Consensus Algorithm**. 2014. 2

[4] JetBrains. **JetBrains/Xodus: Transactional schema-less embedded database.** 5

[5] MiniTool. **Memory vs. Storage: Differences and How Much Do You Need?**, 2021. 6

[6] DigitalOcean. **Java Heap Space vs Stack - Memory Allocation in Java**, 2022. 6

[7] D.Baker. **RAM vs. Hard Drive: What's the Difference? Which One is Better?**, 2022. 6

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. **Dynamo: Amazon's highly available key-value store**. 2007. 7

[9] A. Lakshman and P. Malik. **Cassandra — A Decentralized Structured Storage System**. *Operating Systems Review*, **44**:35–40, 04 2010. 7

[10] O. Rodeh, J. Bacik, and C. Mason. **BTRFS: The Linux B-tree filesystem**. 2013. 7

REFERENCES

[11] O. Rodeh. **B-trees, shadowing, and clones**. page 1–27, 2008. 7

[12] D. Comer. **Ubiquitous b-tree**. pages 121—137, 1979. 7

[13] C. Mohan, D. Haderle, B. Lindsay, and P. Pirahesh, H. amd Schwarz. **ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging**. 1992. 8

[14] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. **Calvin: Fast Distributed Transactions for Partitioned Database Systems**. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, page 1–12, 2012. 13

[15] L. Lamport. **The part-time parliament**. 1998. 13

[16] Fauna. **Fauna**, 2020. 13

[17] M. Freels. **Fauna: Lessons Learned Building a Real World, Calvin-based System**, 2020. 13

[18] The Transaction Processing Performance Council. **The Transaction Processing Performance Council**. 15

[19] NoSqlBench. **NoSqlBench**. 16

[20] M. Michael. **Scalable Lock-Free Dynamic Memory Allocation**. page 35–46, 2004. 17

[21] D. Lomet, A. Fekete, R. Wang, and P. Ward. **Multi-Version Concurrency via Timestamp Range Conflict Management**. *Proceedings - International Conference on Data Engineering*, pages 714–725, 2012. 18

[22] P-A. Larson, Blanas. S., C. Diaconu, C. Freedman, J. Patel, and M. Zwilling. **High-Performance Concurrency Control Mechanisms for Main-Memory Databases**. 2011. 18

[23] Yahoo! **brianfrankcooper/YCSB**. 31