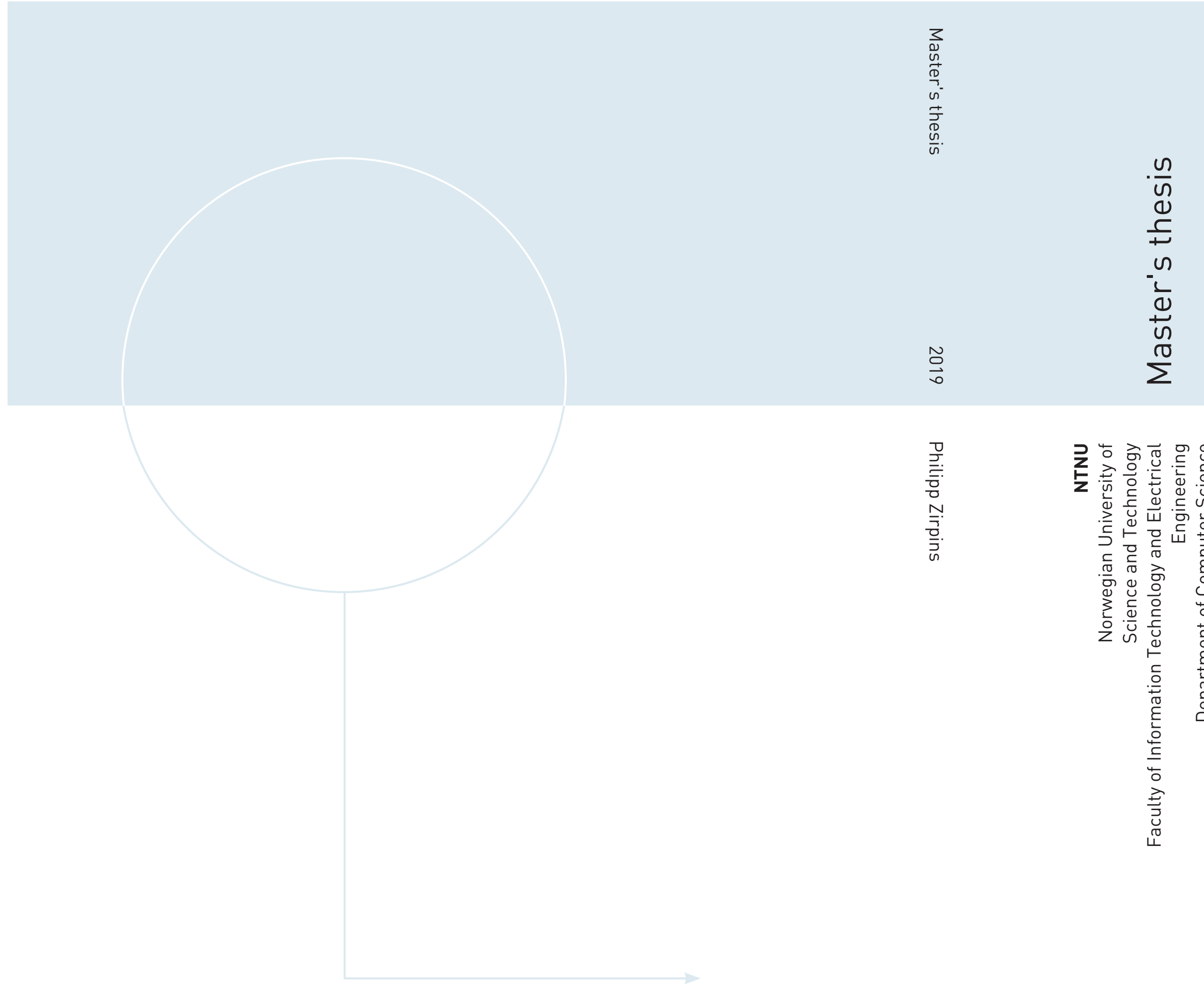


Philipp Zirpins

InnoDB and the Bw-tree

An analysis and comparison of an unlikely friendship in today's times

June 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

InnoDB and the Bw-tree

An analysis and comparison of an unlikely friendship in today's times

Philipp Zirpins

Computer Science
Submission date: June 2019
Supervisor: Rune Humborstad, Jon Olav Hauglid

Norwegian University of Science and Technology
Department of Computer Science

Abstract

In 2013, a team at Microsoft Research published a paper regarding a new type of B-tree, the Bw-tree, made for modern hardware systems that were supposedly far superior to all other modern data structures tested. However, in 2018, another group of researchers had used the 2013 paper's description of the tree to create their own in-memory version, called the OpenBw-tree. The results differed strongly from what Microsoft was claiming in 2013.

Due to the strong variation in results, Oracle had an interest in testing the Bw-tree against their own database. The starting point for this thesis was to split the OpenBw-tree into a client and server side. The main issue, however, was it being a standalone in-memory data structure. As a result, MySQL's storage engine, InnoDB, was directly written to and tested via its Memcached plugin. The plan was to cut the storage engine's cost as much as possible to improve the comparability of both data structures. However, as the OpenBw-tree lacks transaction support, having a potentially large impact on performance, scaling properties of the trees are more important than execution times.

The tests conducted were based on insert-, read-, update- as well as read + update workloads of up to 100 million operations using random-, and monotone-key distributions. Additionally, InnoDB's B+-tree and the OpenBw-tree were tested with up to 64 and 96 concurrent client-server connections respectively. The results show that the OpenBw-tree scales linearly in all test cases. It also has roughly the same performance times for all workload types. The B+-tree, on the other hand, does not scale as good, having most problems with insert and update operations. As a result, the performance times do differ heavily among the workload types and sizes and amount of connections used.

This thesis provides a detailed description of the Bw-tree's core elements and mechanisms which was built upon both research papers' theory parts as well as the OpenBw-tree's code. Even though the Bw-tree performed so well, it should be kept in mind that core storage engine elements, such as transaction support, logging, and persistence, are missing. Implementing these would certainly reduce its performance, however, the scaling potential is quite real. In the end, it is questionable what Oracle would gain by implementing a Bw-tree into InnoDB. After all, the index was built with modern hardware in mind, using CaS operations rather than latches. This either means that substantial changes would have to be made to InnoDB or an entirely new storage engine would be needed instead.

Sammendrag

I 2013 publiserte et team hos Microsoft Research en artikkel om en ny type B-tre, som kalles Bw-tree, er laget for moderne maskinvaresystemer og visste seg til å være langt bedre enn alle andre moderne datastrukturer som ble testet. Men i 2018 hadde en annen gruppe av forskere brukt 2013-papirets beskrivelse av treet for å lage sin egen i-minne versjon, kalt OpenBw-tree. Resultatene var sterkt forskjellig fra hva Microsoft hevdet i 2013.

På grunn av den sterke variasjonen i resultatene hadde Oracle interesse i å teste Bw-treet mot sin egen database. Utgangspunktet for denne oppgaven var å dele OpenBw-treet i en klient og server side. Hovedproblemet var at det var en selvstendig i-minne datastruktur. På grunn av det ble MySQLs lagringsmotor, InnoDB, direkte skrevet til og testet via sin Memcached-plugin. Planen var å kutte lagringsmotorens kostnad så mye som mulig for å forbedre sammenligenbarhet av begge datastrukturene. Men siden OpenBw-treet mangler transaksjonsstøtte, som potensielt har et stort innvirkning på ytelsen, er skaleringsegenskapene til trærne viktigere enn kjøretidene.

Utførte tester ble basert på innsetting, lesing, oppdatering, samt lesing + oppdatering av arbeidsbelastninger på opptil 100 millioner operasjoner ved bruk av tilfeldige og monotone nøkkelfordeler. I tillegg ble InnoDBs B+-tree og OpenBw-tree testet med henholdsvis 64 og 96 samtidige klient-serverforbindelser. Resultatene viser at OpenBw-treet skaleres lineært i alle test tilfeller. Den har også omtrent samme ytelsestider for alle arbeidsbelastningstyper. B+-treet, derimot, skalerer ikke så bra, og har mest problemer med innsetting og oppdatering. Som et resultat av dette varierer ytelsestidene mye mellom arbeidsbelastningstyper og -størrelser og mengden tilkoblinger som brukes.

Denne oppgaven gir en detaljert beskrivelse av Bw-treets kjerneelementer og mekanismer som ble bygd på både forskningspapirets teorideler og OpenBw-treets kode. Selv om Bw-treet gjorde så bra, bør det huskes at kjerne elementer til lagringsmotoren, for eksempel transaksjonsstøtte, logging og vedvarenhet, mangler. Implementasjon av disse vil sikkert redusere ytelsen, men skaleringspotensialet er ganske ekte. Det er tvilsomt hva Oracle ville få ut av ved å implementere et Bw-tree i InnoDB. Tross alt ble indeksen bygget med moderne maskinvare som gjør bruk av CaS-operasjoner i stedet for låser. Dette innebærer at det enten må gjøres betydelige endringer i InnoDB eller det vil være behov for en helt ny lagringsmotor i stedet.

Preface

This thesis marks the end of a two-year master's degree in computer science at the Norwegian University of Science and Technology. This project was conducted at Oracle Norge AS's office in Trondheim between autumn 2018 and spring 2019 under the supervision of Rune Humborstad and Jon Olav Hauglid. However, the outcome that is presented here, was not planned from the beginning. The issue was that the original task was not doable over the course of a year. Hence, this document is mainly the result of the work done since January 2019.

Acknowledgements: I would like to thank Rune Humborstad and Jon Olav Hauglid, that have been my mentors here at Oracle in Trondheim, for their advices and guiding throughout the year. I also would like to give a quick shout out to Svein Erik Bratsberg for his reliability and quick responses regarding thesis and NTNU related questions.

Contents

Abstract . . . . **i**

Summary . . . . **ii**

Preface . . . . **iii**

Table of Contents . . . . **iv**

List of Tables . . . . **vi**

List of Figures . . . . **vii**

1 Introduction **1**
- 1.1 Background 1
- 1.2 Motivation 1
- 1.3 Research goal and questions 1
- 1.4 Caveats 2
- 1.5 Report structure 2

2 Data structures **3**
- 2.1 B-tree 3
- 2.2 B+-tree 4
- 2.3 Bw-tree 7
- 2.4 B+-tree vs Bw-tree 15

3 Evaluation **18**
- 3.1 Implementation 18
- 3.2 Performed tests 26
- 3.3 Results 27

4 Discussion **50**
- 4.1 Overall results 50
- 4.2 Limitations 54

5 Conclusion **56**
- 5.1 Conclusion 56
- 5.2 Further work 56

6 References **58**

Appendices **60**

Appendix A Memcached client code **60**

Appendix B OpenBw-tree client side code **84**

Appendix C OpenBw-tree server side code **111**

Appendix D Python script code **123**

List of Tables

1 Node attributes . . . 8
2 Bw-tree update operation cases . . . 15
3 Overview over the mandatory command line arguments for the Memcached interface. . . . 21
4 Overview over the optional command line arguments for the Memcached interface. . . . 22
5 Random-key results for running the Memcached plugin with 4 and 64 threads. . . . 23
6 Hardware overview for the client and server machines. . . . 26
7 Overview of the tests planed . . . 27
8 Random-key read standard deviation . . . 32

List of Figures

1 B-tree structures 4
2 The nodes of a B+-tree 6
3 Bw-tree architecture overview 9
4 An overview of a logical node 10
5 Pre-allocated Chunk 11
6 Bw-tree node merge process 13
7 Bw-tree node split process 14
8 Memcached interface pseudo code 20
9 OpenBw-tree client side pseudo code 24
10 OpenBw-tree server side pseudo code 25
11 B+-tree (red) and OpenBw-tree (blue) results for **10 million random-key insert operations** done with 8, 16, 32, 64 and 96 connections. . . . 28
12 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million random-key insert operations** done with 8, 16, 32, 64 and 96 connections. 28
13 B+-tree (red) and OpenBw-tree (blue) results for **10 million monotone-key insert operations** done with 8, 16, 32, 64 and 96 connections. . . . 30
14 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million monotone-key insert operations** done with 8, 16, 32, 64 and 96 connections. 30
15 B+-tree (red) and OpenBw-tree (blue) results for **10 million random-key read operations** done with 8, 16, 32, 64 and 96 connections. 31
16 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million random-key read operations** done with 8, 16, 32, 64 and 96 connections. 32
17 B+-tree (red) and OpenBw-tree (blue) results for **10 million monotone-key read operations** done with 8, 16, 32, 64 and 96 connections. 33
18 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million monotone-key read operations** done with 8, 16, 32, 64 and 96 connections. 33
19 B+-tree (red) and OpenBw-tree (blue) results for **10 million random-key update operations** done with 8, 16, 32, 64 and 96 connections. . . 34
20 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million random-key update operations** done with 8, 16, 32, 64 and 96 connections. 34
21 B+-tree (red) and OpenBw-tree (blue) results for **10 million monotone-key update operations** done with 8, 16, 32, 64 and 96 connections. . . 36
22 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million monotone-key update operations** done with 8, 16, 32, 64 and 96 connections. 36
23 B+-tree (red) and OpenBw-tree (blue) results for **10 million random-key read + update operations** done with 8, 16, 32, 64 and 96 connections. 37

24 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million random-key read + update operations** done with 8, 16, 32, 64 and 96 connections. 37
25 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **10 million monotone-key read + update operations** done with 8, 16, 32, 64 and 96 connections. 38
26 B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million monotone-key read + update operations** done with 8, 16, 32, 64 and 96 connections. 39
27 Bw-tree performance scaling for 10, 50 and 100 million random-key insert operations. 40
28 B+-tree performance scaling for 10, 50 and 100 million random-key insert operations. 40
29 Bw-tree performance scaling for 10, 50 and 100 million monotone-key insert operations. 41
30 B+-tree performance scaling for 10, 50 and 100 million monotone-key insert operations. 41
31 Bw-tree performance scaling for 10, 50, and 100 million random-key read operations. 42
32 B+-tree performance scaling for 10, 50, and 100 million random-key read operations. 42
33 Bw-tree performance scaling for 10, 50, and 100 million monotone-key read operations. 43
34 B+-tree performance scaling for 10, 50, 100 million monotone-key read operations. 44
35 Bw-tree performance scaling for 10, 50 and 100 million random-key update operations. 44
36 B+-tree performance scaling for 10, 50 and 100 million random-key update operations. 45
37 Bw-tree performance scaling for 10, 50 and 100 million monotone-key update operations. 45
38 B+-tree performance scaling for 10, 50 and 100 million monotone-key update operations. 46
39 Bw-tree performance scaling for 10, 50 and 100 million random-key rand + update operations. 46
40 B+-tree performance scaling for 10, 50 and 100 million random-key read + update operations. 47
41 Bw-tree performance scaling for 10, 50 and 100 million monotone-key read + update operations. 48
42 B+-tree performance scaling for 10, 50 and 100 million monotone-key read + update operations. 48
43 Bw-tree performances for all **10, 50 and 100 million monotone-, and random-key workloads with 8, 16, 32, 64 and 96 connections**. . . 50
44 B+-tree performances for all **10, 50 and 100 million monotone-, and random-key workloads with 8, 16, 32 and 64 connections**. 51
45 All Bw-tree performance scalings for 10, 50 and 100 million monotone-, and random-key workloads. 51

46 All B+-tree performance scalings for 10, 50 and 100 million monotone-, and random-key workloads. 52

1 Introduction

1.1 Background

In 2013, Microsoft Research published a paper[13] concerning a new type of B-tree, made to utilize today's hardware standards. The so-called Bw-tree is a latch-free index said to exploit the caches of multi-core chips and perform well on flash storage. To evaluate the performance, operations/second were used as measurement while the Bw-tree ran against BerkeleyDB in B-tree mode, as well as a latch free Skip list implementation. The results, based on in-memory performance, claim that the Bw-tree had, depending on the workload, a 5.8 to 18.7 times and 3.7 to 4.4 times higher throughput than the BerkeleyDB B-tree and the Skip list implementation respectively. The team behind the paper believes that these improvements are based on the Bw-tree's latch-freedom as well as CPU cache efficiency as the measurements show an almost 15 percentage points higher L1 + L2 cache hit rate for the Bw-tree.

However, in 2018, a group of researchers published a paper in response to the claims made by the team at Microsoft Research [17]. Their contribution is supposed to fill the gaps, left by the 2013 paper, on how to build a Bw-tree and how to optimize it as well as demonstrating that there are other concurrent data structures outperforming the Bw-tree by a factor of 1.1 to 2.5 while using locks. The Bw-tree implementation used for conducting the tests is called OpenBw-Tree and is part of the index benchmarking framework (IBF), which was used to run this Bw-tree against other data structures[16]. Both the IBF and OpenBw-tree were developed by the researchers.

1.2 Motivation

Based on the results mentioned above, Oracle has an interest in finding the capabilities of the Bw-tree compared to its own B+-tree implementation in MySQL's InnoDB storage engine. Since both papers have vastly different opinions about the Bw-tree, tests in a more realistic setting relative to the use of MySQL are of interest. This refers to workloads run against InnoDB and the OpenBw-tree to determine their differences.

Additional questions can be raised when it comes to the testing of update and deletion of records. Updates always "success" regardless of the actual outcome of the operation. This will be further discussed in section 3.1.

1.3 Research goal and questions

$Q1$: The creation of a solid introductory documentation of the elements, properties, and functions of the Bw-tree.

$Q2$: What are the scaling capabilities of both data structures inside the IBF and InnoDB in terms of manageable workload?

$Q3$: What could Oracle gain by implementing the Bw-tree in MySQL's InnoDB?

1.4 Caveats

Direct comparisons between the OpenBw-Tree and B+-tree inside InnoDB are assumed to be inaccurate as InnoDB is a fully functional storage engine while the OpenBw-Tree is a standalone in-memory Bw-Tree implementation. In theory, this should cause the Bw-Tree to be faster in general as it has not to deal with logging, persistence, and multi-version concurrency control (MVCC).

To handle this situation, rather than mainly compare operations/second, the results in this article will also be based on relative scalability. In that way, it is possible to compare how both data structures perform under different circumstances.

A final concern is that Microsoft never published the code their paper is based on. Thus, it is not guaranteed that the OpenBw-Tree, in fact, is a correct implementation in the sense of how it was designed at Microsoft research. Also, it is a partial implementation as Microsoft's Bw-Tree was integrated into a storage engine, called LLAMA[12], that was specifically built for today's hardware environments.

1.5 Report structure

The remainder of this report is structured into 4 parts. Section 2 gives an overview of the relevant data structures and discusses what effect their properties should have on the tests. Relevant tools, techniques, and functions will be discussed in section 3 combined with an overview of the tests performed as well as their results. Section 4 contains a reflection on the results as well as a summary of how the caveats influenced the tests' outcome. Lastly, section 5 contains a conclusion as well as a description of future work.

2 Data structures

This section starts by giving a brief introduction to B-trees, followed by a description of the similar B+-tree, used in MySQL's InnoDB, and how CRUD-operations, latching and locking work in it. Next, the elements, properties, and functions of the Bw-tree will be described in a bit more detailed fashion. Lastly, thoughts on both data structures will be uttered.

2.1 B-tree

The B-tree is a data structure that guides the search for key-value pairs, so-called records, stored inside it. The search is based on either on the key or values associated with keys. The information for section 2.1 and 2.2 come mainly from one source[9]. However, extra references to it will be made where appropriate.

Architecture and properties

A B-tree is made up of nodes which there are three types of: root, internal and leaf. There can only be one root node. It is the only one to have no parent node, meaning that no other node points to it. Leaf nodes, on the other hand, have no child nodes. All other nodes are internal nodes as they are pointed to by their parent and they themselves point their own child nodes.

Structurally, all nodes are equal. Figure 1 (a) shows the contents of a node. From left to right we have a tree pointer P_1 pointing to a sub-tree containing keys smaller than key K_1, then key K_1 with its respective data pointer Pr_1, followed by a tree pointer P_2 pointing to a sub-tree containing keys larger than key K_1 but smaller than key K_2 and so forth. This results in the B-tree's keys being sorted hierarchically, from left to right. Thus, it allows for sequential traversing and, due to the structure of the tree, minimises the number of disk accesses.

Compared to the binary tree, in the B-tree, each node can have more than two children up to a maximum p, the so-called B-tree order of p. Figure 1 (b) shows a B-tree of order p = 3 and height 2. All nodes are structurally the same except for their contents as the leaf nodes have no children to point to, but the space inside the nodes is reserved for pointers in case the tree has to expand.

The B-tree makes sure that it stays balanced, which means that all leaf nodes are at the same level. This, combined with the fact that there are exponentially more keys per level, ensures logarithmic access times as the algorithms, traversing the tree in search of a record, orient themselves by the sorted keys in the tree.

Structural modifications (SMO): In the B-tree, there are operations causing the data structure to change. Node splits and merges are initiated when a thread tries to

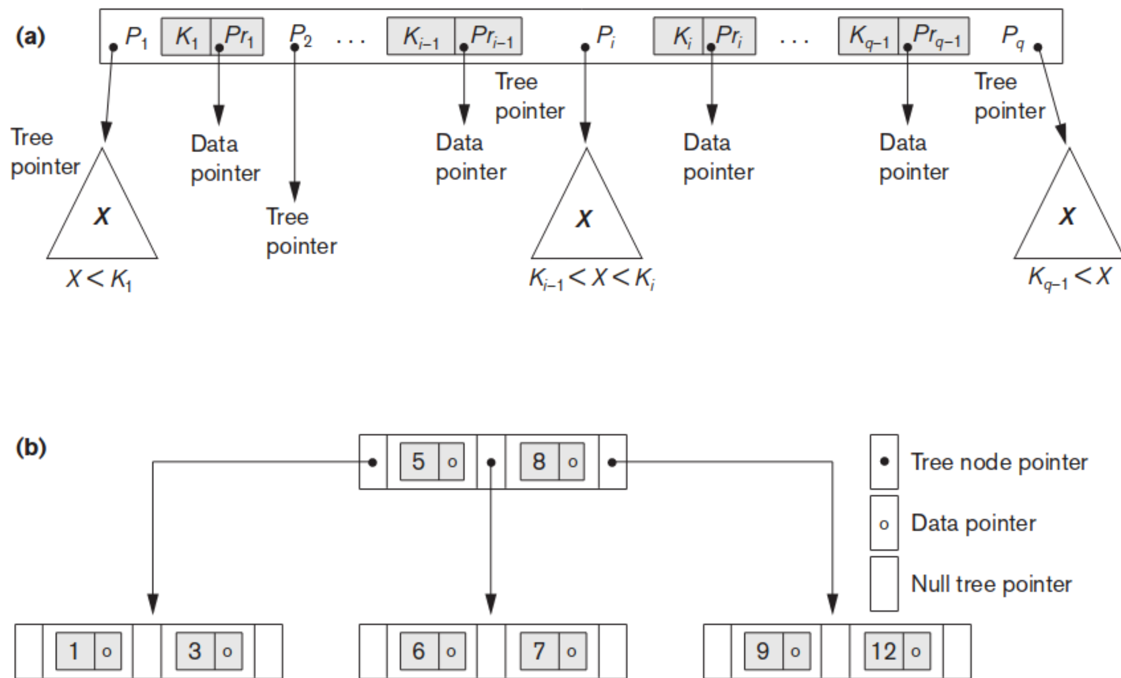


Figure 1: B-tree structures
(a) A node in a B-tree with q-1 search values. (b) A B-tree of order p = 3[9]

insert a record into an already full node or when it removes a record, causing the node to be less than half full respectively.

- **Node split**: Initially, a new node is created and the records in the original node are divided equally among both of them. Since there is now one more node, pointers to and from the original node as well as the new node have to be updated. The result can be that the tree has to be restructured recursively to keep it balanced, which means that the parent nodes of the original and new nodes might also need to split.
- **Node merge**: The remaining values are equally divided among the original node's left and right neighbour. Pointers have to be updated which again might cause recursive restructuring.

Even though node splits and merges can be costly, simulations and analysis have shown that this only occurs rarely as the nodes are on average 69% full[9], meaning there is enough room and record for both insertions and deletions before restructuring becomes necessary.

2.2 B+-tree

The B+-tree is a variation of the B-tree that mainly makes better use of the space given which is enough to greatly improve scalability and hence performance.

Architecture and properties

In a B+tree, the keys with their respective data pointers are stored only at leaf level. Not having to store data pointers in the root and internal nodes mean that there is more space for pointers to additional internal and leaf nodes. This means that a B+-tree with height h has the potential to store many more records than a B-tree of the same height, which gives better I/O performance for the B+-tree as fewer disk-accesses are necessary on average.

Also, each leaf node is connected via a pointer to its right neighbour to improve scan performance. The B+-tree has, just like the B-tree, logarithmic access times.

However, since all data pointers are stored at the leaf level, it also means that all keys must be stored there as well. But in order to be able to traverse the tree in logarithmic time, some keys have to be stored a second time in non-leaf nodes. This applies to a small number of keys and each key is stored redundantly at most once.

While the Bw-tree is mostly both lock and latch free, the B+-tree uses both to implement transaction support. Locks are used to prevent worker threads from accessing certain nodes or records via reads or writes during a transaction. Latches, on the other hand, are used to implement locks in the first place. They are mutexes or semaphores in memory that protect critical sections that are used to set or remove locks. Locks usually stay in place for the duration of the whole transaction while latches are short-lived as they should not block other worker threads from accessing shared data for too long[10].

B+-tree concurrency control: In order to implement transaction support, the B+-tree uses two-phase locking. When a transaction, and thus the first phase, starts, the worker thread will try to acquire locks on the records it had to operate on. This is called row-level locking. During this phase, no locks can be released. When a record has to be read, a read-lock will be established on it, preventing other threads from changing its contents. On the other hand, if a record's contents have to be changed, then a write-lock is required, preventing threads from accessing the record at all. When a worker thread is done, it can start releasing the locks again. During this second phase, now new locks can be acquired[5][9]. However, using locks and latches comes with a high overhead as for every operation, read or write, a system locking request has to be issued first. To avoid these costs as much as possible, multi-version concurrency control (MVCC) is used. The idea is to reduce blocking to a minimum by keeping outdated versions of records, thus *"multi-version"*[9], so that readers do not block writers and vice versa by having each transaction use a database snapshot. These snapshots reflect the indexes state at the point in time before the transaction started. The only case of contention remaining is when two transactions try to update the same record, one of them has to wait until the transaction, that modified the record first, has either committed or rolled back.

The two types of locks used are read and write or shared and exclusive. A read-lock is set by a worker thread that wants to access a record but not change it. Hence, multiple threads can set read-locks on a single record. Write-locks, on the other hand, give exclusive access to a single thread as the contents of the record are going to be changed.

The B+-tree's concurrency control mechanism, implemented using latches and locks, is said to be pessimistic. For instance, even if InnoDB runs with a single worker thread T only, meaning there are no other threads that could try to access records T is working on, T will still acquire read and write locks as needed. The Bw-tree, on the other hand, is said to be optimistic in this regards. Its concurrency control is based on the possibility that a compare-and-swap operation can fail due to another worker threads actions. In this case, the operation has to be restarted from the beginning.

Figure 2 (a) shows an internal or root node of a B+-tree. The difference to the nodes in a B-tree, as depicted in Figure 1 (a) is easy to spot. There are no data pointers associated with the keys. The rest, however, is the same. Part (b) on the other hand visualises a B+-tree's leaf node, which solely stores keys and their respective data pointers as well as a single pointer to the next leaf node to the right.

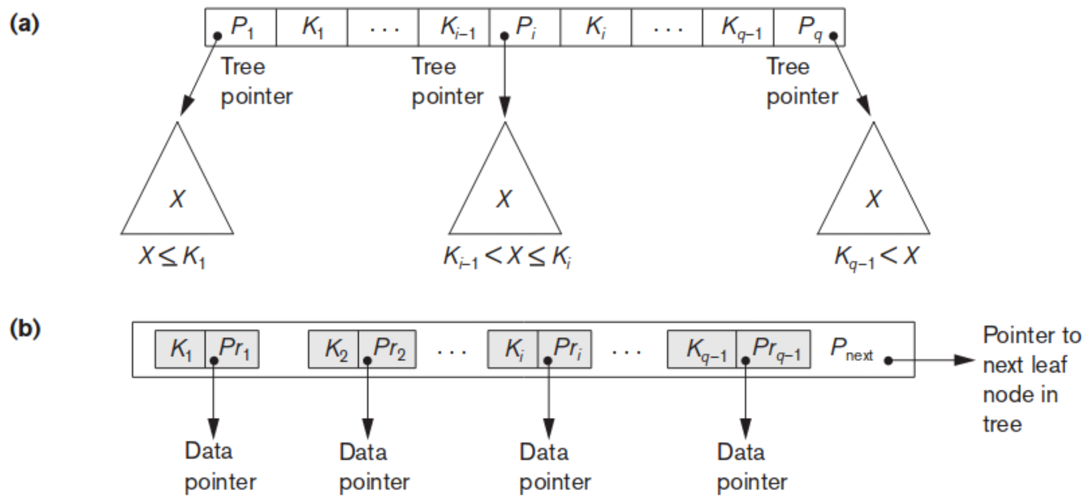


Figure 2: The nodes of a B+-tree
(a) Internal node of a B+-tree with q-1 search values. (b) Leaf node of a B+-tree with q-1 search values and q-1 data pointers[9].

CRUD-operations

All operations have one thing in common. They all start with at least a given key that is used to traverse down the B+-tree's hierarchy to find the position where the operation has to be executed. During traversal, the worker thread acquires a read-lock on its current node. Once a lock is also set on the node below the threads current level, it releases the first read-lock.

Insert: If the key of the given record does not already exist and the node where it has to be inserted is not full, then a write-lock is acquired on the empty record slot, then the new record is inserted into the node and the lock is released again. If the node is full, it has to be split first to make room for the new key-value pair. Node splitting in a B+-tree is essentially the same as in a B-tree with the only difference being that the

leaf nodes' neighbour pointers also have to be updated.

Read: If the key exists in the B+-tree, a read-lock will be acquired on its record first. Next, its associated value is read and the lock is released. If the key does not exist, the operation fails.

Update: If the key exists, acquire a write-lock on its record. Then change its value to the new given value. Lastly, release the lock again. If the key does not exist, this operation might simply fail or instead, based on the implementation, execute an insert-operation.

Delete: If the key exists, acquire a write-lock on its record, then delete the record and release the lock. There will be one free record spot in the node and if it is now less than half full, a node merge-operation has to be executed. A B+-tree's merge-operation is also practically the same as a B-tree's with, again, the only difference being that the leaf nodes' neighbour pointers have to be updated.

2.3 Bw-tree

The Bw-tree is an attempt to create a new data structure build with today's computer properties in mind, including multi-core CPUs, large main memory and flash storage[14]. The tree is mostly[1] latch free compared to the B+-tree that needs both locks and latches to implement transaction support.

Architecture and properties

The Bw-tree has two distinct, performance-related purposes. First, the tree's structure is seldom edited directly. Instead, so-called delta nodes, describing the changes to a node's current state, are appended to either the original node or its chain of deltas. These chains of changes cause the actual node to stay untouched for a while which reduces cache line invalidation a lot. Second, the technique used to append updates is an atomic compare and swap instruction, further explained below, causing threads to have less idle time as there are no latches blocking access to shared items.

Base nodes: There are two types of base nodes in a Bw-tree, inner base nodes and, leaf base nodes. Both are similar to a B+-tree's root or internal, and leaf nodes respectively and each node has a unique ID.

- **Inner base Node**: It holds a sorted separator item array, where each item is made up of a key, which the array is sorted after, and a node ID. A key, in this case, is also called a separator as it separates keys smaller than itself from the keys larger than itself, just like in a B-, or B+-tree. A key's associated node ID, as Figure 3 shows, is a logical link to one of four destinations. It can point to another

[1] Mostly latch free: the Bw-tree and mapping table data structures can be implemented latch free. But there are improvements, I/O, as well as higher layer components that require either latches or locks[13]

base node of either inner or leaf type, or instead, point to a delta chain associated with one of these base node types. Additionally, the node stores the meta data listed in Table 1.

- **Leaf base Node**: Figure 4 shows a leaf base node in more detail as well as the two delta nodes that are associated with it. The construct of a base node and its deltas is called logical node. The only difference between an inner base node and a leaf base node is the content that is associated with the keys in the item array. Just like in a B+-tree, the keys in the leaf node are associated with values.

Attribute	**Description**
low-key	The smallest key stored at the logical node. In a node split, the low-key of the right sibling is set to the split key. Otherwise, it is inherited from the element's predecessor.
high-key	The smallest key of a logical node's right sibling. **Δsplit** records use the split key for this attribute. **Δmerge** records use the high-key of the right branch. Otherwise, it is inherited from the element's predecessor.
right-neighbour	The ID of the logical node's right sibling.
size	The number of items in the logical node. It is incremented for **Δinsert** records and decremented for **Δdelete** records.
depth	The number of records in the logical node's Delta Chain.
offset	The location of the inserted or deleted item in the base node if they were applied to the base node. Only valid for **Δinsert** and **Δdelete** records.

Table 1: Node attributes
The list of attributes that are stored in the logical node's elements, base nodes and delta records[17].

Mapping table and CaS operations: In a Bw-tree, threads do not traverse the nodes via physical links or pointers like in a B-, or B+-tree, but via a combination of both logical and physical ones. As depicted in the upper part of Figure 3, the root node has two logical pointers to its children, which actually are just their node ID's. Now, the purpose of the mapping table is to map these node ID's to physical pointers that must always point to the current state of their respective logical node, which is either the base node itself or its latest delta node.

In the end, threads traversing the Bw-tree, will continuously consult the mapping table in order to know where to go next. That is why the creators of the OpenBw-tree also calls the mapping table an indirection layer[17].

Updates to the Bw-tree are done in a special way through so-called compare-and-swap (CaS) operations. A CaS operation compares a given, anticipated value to the actual, current value. In this case, the values in mind are memory addresses or offsets based on the node's position in memory or on stable storage respectively. If both match, then a new given, current value is appointed by the executing thread.

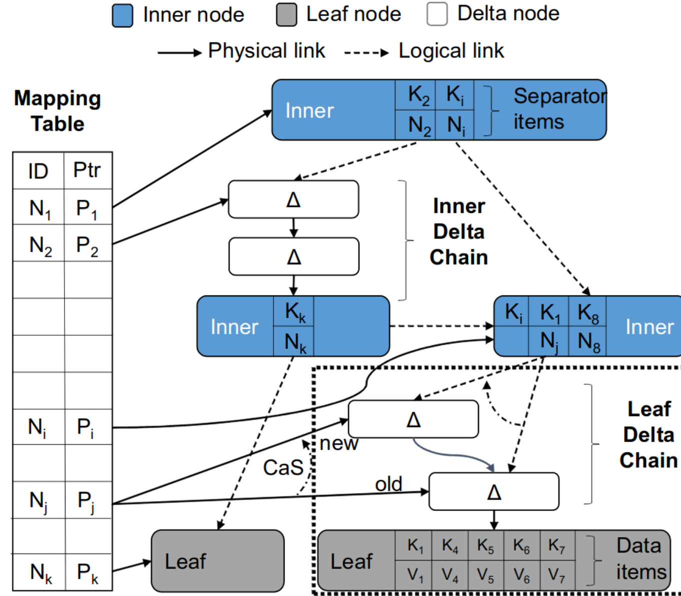


Figure 3: Bw-tree architecture overview
An instance of a Bw-tree with its internal logical links, Mapping Table links, and an ongoing CaS operation on the leaf Delta Chain[17].

This instruction has a property which the Bw-tree is build upon, it is atomic. This means both the compare and the swap stage essentially happen instantly at the same time and cannot be separated. The result is that threads in a concurrent environment cannot interfere with each other as no other thread can execute the CaS at the same time. This behavior is guaranteed by the hardware used.

The only source of interference can be the compare-step where a thread can encounter a different value than expected, in this case, the logical node's current address in the mapping table. Now, the thread would have to restart its operation that usually begins by re-traversing again from the tree's root as this is easiest retry-protocol to implement and already traversed nodes are likely to be in the CPU cache anyway.

In case the Bw-tree changes, for instance through the installation of a new delta node, then the change necessary, in order to install that new node, is to update the pointer in the mapping table as this is the only way the delta can be accessed. This is depicted in the marked part of Figure 3. There is a CaS operation indicated for the physical pointer of Node N_j. It is also shown that the logical pointer of its parent node changes, this, however, is merely a consequence of the physical pointer changing. Hence, updates to the Bw-tree's structure require only a single atomic operation to be performed. This also means that the effects node updates are isolated to that node alone.

If a logical node resides on stable storage but is required by a traversing thread, it

will be loaded into memory and the physical pointer in the mapping table gets updated. No pointer was ever invalid in this case as logical pointers are translated via the mapping table and the only physical pointer pointing at the node is always up to date, regardless where the node is stored.
Logical nodes that end up on disk are there due to the Bw-tree growing. While there is a way to expand the mapping table in a lock-free manner, this does not seem to be the case for the shrinking process[17].

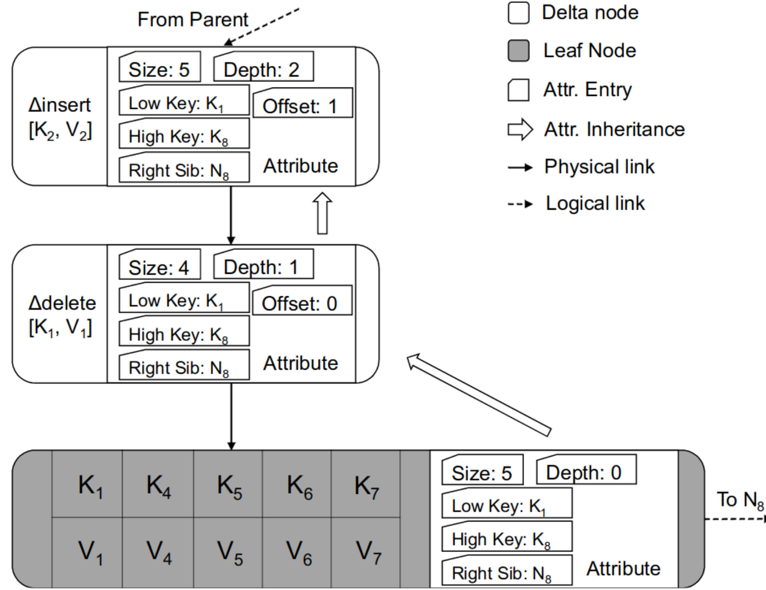


Figure 4: An overview of a logical node
A more detailed illustration of a logical leaf node from Fig. 3 with its base node and two delta nodes.

Deltas and delta chain: Figure 3 shows two delta nodes, also called delta records, in a delta chain for the leaf base node with ID N_j. When a base node is to be modified, then a delta node, describing the change to the node, will be appended to the node's delta chain using a CaS operation. Depending on the kind of the modification, a ΔInsert, ΔUpdate or ΔDelete record will be added. The delta chain is a chronologically-ordered, singly linked list of modifications applied to a node. Links between the deltas and the base node are physical pointers. Due to this technique, the base node stays the same and cache lines are not invalidated.

There are seven delta node types that are usually used in the Bw-tree, however, different articles might have different names for them.

- ΔInsert, ΔUpdate, and ΔDelete deltas are used in update operations.
- ΔRemove, ΔMerge and ΔSplit deltas are special purpose nodes that can change the execution routine of threads by notifying them about an ongoing or past node merge or split operation.
- ΔSeparator is a delta record that is essentially either a ΔInsert or ΔDelete node with extra key-ID values attached to it as part of the finishing step for a node

merge or split.

The delta nodes used for modification hold, just as every base node, the meta data listed in Table 1. This might also be true for the ΔSeparator but it is not clear if the special purpose nodes share the same set of meta data as the other delta node types.

Additionally, the meta data for a delta node D differs from that of the base node N it is associated with. While N's meta data represent its original state before any delta updates, D's meta data represents the logical base node's current state as of when D was added to the logical node. This means that a delta node's meta data is based on either the latest delta node added to the delta chain or if there is no delta chain yet, the base node.

There is an issue that comes with the delta chain as described in Microsoft's original implementation of the Bw-tree. The nodes and deltas are scattered all over the place in memory, causing pointer-chasing which leads to cache and translation lookaside buffer misses. Additionally, exorbitant allocation of small memory blocks creates contention in the allocation manager, especially as the number of CPU cores increases, and will lead to external fragmentation over time. A solution proposed in the OpenBw-tree is to pre-allocate blocks of memory large enough to hold a base node plus a delta chain of a given maximum length. The base node will be stored at the end of the block and each new delta node is stored more and more towards the beginning of the allocated memory block as it is shown in Figure 5. This improves the CPU's cache performance due to spatial and temporal locality. Additionally, a pointer, called marker, is used to keep track of the latest node's position inside the block.

The length of a delta chain can become a performance issue. Hence, once a thread T adds a new delta node to a delta chain that now exceeds a certain threshold, a process called delta chain consolidation is initiated by T. First, T takes a private copy of the base node, then, the delta chain is chronologically applied to it. Lastly, the physical pointer of the old logical node is changed to the consolidated one with a CaS operation. Consolidation, however, comes with a replay and sorting overhead which can be fixed as it was for the OpenBw-tree with the fast consolidation technique[17].

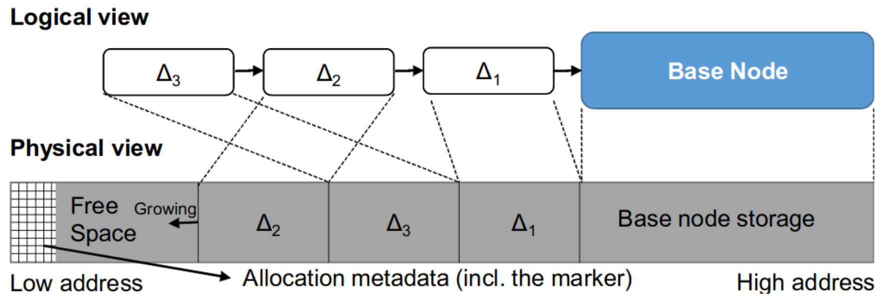


Figure 5: Pre-allocated Chunk
The logical view of a logical Bw-tree base node. Sub-chunks, or slots, are allocated executing a CaS operation on the marker.

Garbage Collection (GC): Since the Bw-tree does not allow for exclusive access to shared data structures, there needs to be a mechanism that makes sure that outdated elements get removed properly. For instance, a thread T, executing a delta consolidation, creates a new, consolidated base node N. The old logical node O still exist, yet it is not accessible anymore due to the CaS that installed N. Still, threads might have managed to access O before N was put in place so T cannot just delete O.

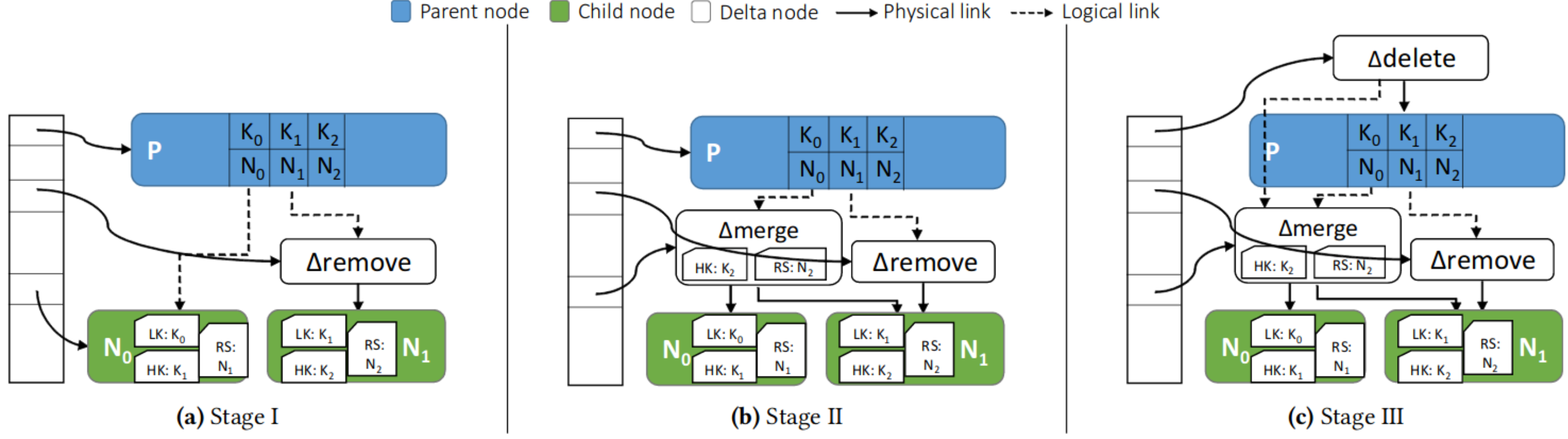
So-called epochs solve this problem. They essentially protect objects from being removed before all threads, being part of an epoch, have finished their operation[13]. The index is constantly maintaining a list of epoch objects of which only one is the current epoch E and threads that want to execute an operation have to join that epoch. Threads can only join the current epoch but can still be inside it while it no longer is the current one. As long as an epoch is not drained, meaning that there is at least a single thread that has not left it, elements marked for deletion are not removed by threads dedicated for garbage collection. Threads on average leave an epoch after the time it takes to execute a single operation. New epochs are added at a constant rate so older ones get drained and outdated elements can be removed and memory as well as node ID's freed can be recycled.

Going back to the example above, but now in the context of epoch E. After T has installed N, it adds O's ID to E's garbage list. This list holds references to all elements that were declared garbage while threads were executing in E. This type of garbage collection does, however, not scale very well as all threads in epoch E add their nodes to a single garbage list. The OpenBw-tree comes thus with an improved, decentralised solution for its garbage collection where each thread has its own garbage list within an epoch[17].

Structural Modifications (SMO): Just like in a B-, and B+-tree, there are two types of operations in a Bw-tree that cause structural modifications, node splits and node merges. They are initiated if a thread notices a full or less than half full node and the resulting workload is split among multiple threads. The Bw-tree does not use latches here as well. As a result, each SMO is divided into three steps and multiple CaS operations are required to execute a node merge or split, one per delta node installed. Also, multiple threads might simultaneously try to execute each separate step. In the end, only one thread will succeed. The other will retry and observe the already executed step.

- **Node merge**: When a thread detects a logical node N_1 being below some threshold size it starts the node merge process, given that the node has a left neighbour. The thread will add a ΔRemove node to N_1, as depicted in Figure 6, which now stops all further use of that node. Other threads that still want to access N_1 will encounter the ΔRemove node and move to N_0, N_1's left neighbour instead. In order to be able to do so, threads have to at least keep track of which parent node P they came from, so they do not have to re-traverse the entire Bw-tree again.

  A thread that reaches N_0, coming from N_1, will append a ΔMerge record to N_0, containing N_0's low key, N_1's high key, as well as N_1's logical right neighbour



(a) Stage I (b) Stage II (c) Stage III

Figure 6: Bw-tree node merge process
A depiction of the step-by-step execution of a node merge. Each image (a), (b), (c) shows the result after a delta node installation. Only important details are given for better understandability.

pointer. Additionally, this delta holds N_1's low-key allowing incoming threads to correctly traverse and execute their operation on either N_0 or N_1 via N_0. From now on, N_0 and N_1 are considered to be part of the same logical base node. The remainders of N_1's node are merged and reclaimed when N_0 is consolidated.

The last step is initiated by any thread encountering the ΔMerge node. It will add a ΔSeparator record to P concerning the key-ID pair for N_1 in P. Additionally, this delta contains two key-ID records. The first one being N_0's low key and it's node ID, the second one being N_1's original high-key coupled with its former right neighbour's ID. A thread T that comes across the ΔSeparator node uses those additional key-ID pairs to either directly traverse to N_0 using its ID or, if T's given key lies outside the low-, and high-key's range, it must further traverse the delta chain or base node. Now, only N_0 and the mapping table point to N_1. The difference is that the mapping table pointer is not used anymore as N_1's node ID is practically removed from its parent node.

Lastly, the thread that successfully installed the ΔSeparator appends the ΔRemove node, added to N_1 in the beginning, to the epochs garbage list. This does not free the memory location containing that originally was N_1 as that first happens when the ΔMerge node is removed during consolidation. However, when the ΔRemove node is deleted, the entry in the mapping table pointing to it is removed and N_1's former ID gets recycled for further reuse in the future.

- **Node split**: A split is initiated when a leaf or internal node N_0 has grown in size beyond a certain threshold. The thread T that detects such a situation first tries to execute its own operation and then initiates the splitting process which is described in Figure 7.

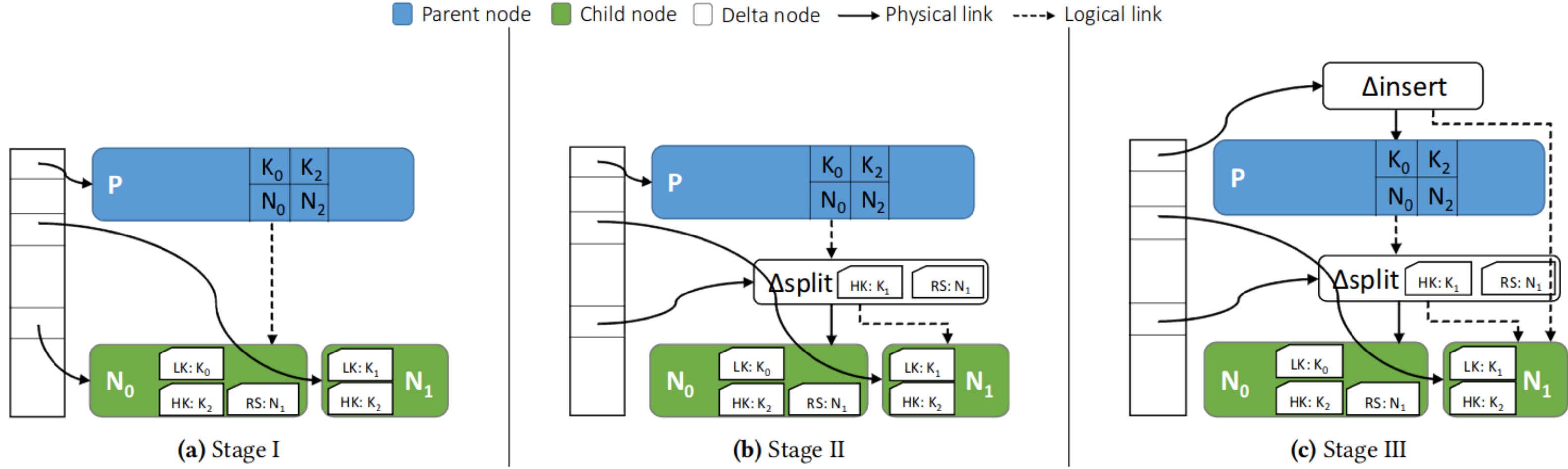


Figure 7: Bw-tree node split process
A series of images showing the results after (a) a new node was created and (b), (c) delta nodes were appended. Important details are given like low-, and high-keys while other data is omitted for simplicity.

First, T creates a new, empty node N_1 which will be the new right neighbour of the original node N_0. Next, it finds the separator key K_1 used to divide N_0's item array entries in half. Records, as well as delta nodes associated to them, having a key larger than K_1 are installed in N_1 and consolidated. Now N_1 gets its meta data, which also includes a logical pointer to N_0's original right neighbour N_2. Next, N_1 is installed in the mapping table. This is done without a CaS as only T knows about N_1 up to this point. Now, a ΔSplit node is added to N_0's delta chain. This delta contains the split key K_1 and a logical pointer to N_1 which makes N_1 N_0's new right neighbour. The split key is used to invalidate records and their associated deltas in N_0 as these now reside in N_1. A thread wanting to access a record with a key larger than K_1 now uses the logical link to N_1.

The last step is about updating N_0's and N_1's parent node P by adding a ΔSeparator to it. This delta contains a logical pointer to N_1, as well as the separator keys K_1 and K_2.

Unfortunately, it is not entirely clear how the invalidated keys are removed from N_0. Having access to the OpenBw-tree's code did not make things clearer in this case. However, one possible option would be that the thread that executes a consolidation on the node sees the ΔSplit node and uses the low-, and high-key in N_0 to not re-install the invalidated records, thus removing them.

- **Concurrent split and merge**: The paper by Microsoft lacks clarification on what happens in case a parent node has to be split when there is also a merge operation being executed on at least one of its children. The problem here is that the parent node will end up in an inconsistent state. The OpenBw-tree solves this

by introducing a ΔAbort record with the downside of introducing the use of a write-lock into the Bw-tree[17].

CRUD-operations

In the context of a Bw-tree, CRUD-operations are divided into page search and leaf-level update operations. The former concerns the read-operation, the latter includes insert-, update- or modify-, and delete-operations.

Page search: A thread T that reaches the leaf base node where the given key K should be located, will first traverse the node's delta chain given that it exists. If T encounters a Δinsert or Δupdate node for K, then it returns the value integrated into these nodes as they represent the current state of K's associated value. However, if T first finds a Δdelete node for K, then it has to abort is operation since the record, chronologically, no longer exists. If neither of these circumstances is the case, then T has to execute a binary search on the base leaf node's item array and return K's record if it exists.

Update operations: There are a handful of cases a thread can run into when traversing a leaf base node's delta chain while trying to execute an operation. Table 2 sums up these cases and depicts the outcome of the operation in each case.
A check mark symbol (✓) shows that the given operation will succeed if the thread encounters a certain delta node, given that both the operation and the delta node concern the same key K. The second case depicts a failure in that situation, represented by an x-symbol (×).
For instance, a key, coupled with whatever value, that already exists in the leaf base node cannot be inserted a second time. Likewise can a thread not update or delete an already deleted key. However, keep in mind that these cases might turn out different for an implementation supporting non-uniq[17].
If the thread succeeds, it will add a delta record to the node or delta chain. Otherwise, it has to search for the key in the leaf base node which is done in a binary search fashion just like in a B-, or B+-tree. Again, on success, a delta node is appended or else the operation fails.

	ΔInsert	**ΔUpdate**	**ΔDelete**
Insert Op	×	×	✓
Update Op	✓	✓	×
Delete Op	✓	✓	×

Table 2: Bw-tree update operation cases

2.4 B+-tree vs Bw-tree

This section expresses thoughts on how the B+-, and Bw-tree will cope with different CRUD operations and workloads. Hence, this part is solely based on expectations that are going to be used later for discussion purposes and reflection.

Thoughts on the B+-tree

Insertions and deletions should take the longest time on average since data will actually be written to or removed from the structure and re-balancing might happen as well. Still, as the tree grows, less and less structural changes need to be done since there is exponentially more space for new records per extra level in the tree. This means that insertions and deletions should scale well, meaning that if the workload grows with a factor of 10, then the time required to execute that workload should grow roughly with the same amount.
Update operations are going to be cheaper than insertions and deletions as they only change the value of a key without actually changing the structure of the tree.
Reads should be by far the cheapest operations, even when the tree grows very large. This is due to the fact that the tree can store exponentially more records per level, hence the path to a record is always relatively short. The number of records inserted into the tree should not have to say very much for read and update operations. Meaning that a tree populated with 10 million records, on which 100 million read or update operations are executed, should give closely the same execution time as a tree populated with 100 million records as the high of both tree's should not be that different. This might also be the case for a single insert as its overall change to the data structure is relative to nonw at all. It is hard to say how the number of threads is going to scale in InnoDB. From an intuitive standpoint, it seems reasonable that some number of threads X will finish a given workload W in roughly twice the time 2X will. However, some point of maximum scaling effect is expected as the machines, which will perform the tests, have their limitations too.

Thoughts on the Bw-tree

Modifications to the Bw-tree are expected to be highly effective as each thread only has to append its own delta to the delta chain. There are, however, some variables whose impact is hard to determine. It seems intuitive that a longer delta chain would slow down a thread's execution but that is only the case if the thread actually has to traverse the entire delta chain without finding a delta related to its own given key. Another issue can be the number of updates to a single node. Usually, the delta chain will grow quickly, meaning that updates should work without large delays. This should be true even if a lot of threads have to retry their operation as the Bw-tree's height should not be the largest cost here. The only thing that actually will cost, in a relative sense, is an SMO. but the larger the tree gets, the more are updates likely to be divided over the whole data structure, hence reducing the amount of SMOs and consolidations.

It is easy to see why reads should be the cheapest type of operation in a B+-tree, but things are a bit different in a Bw-tree. Both update- and read-operations should have, on average, the same chances to find a delta node that stops their traversal. This is only an assumption based on intuition, but if this is true, then, the average performance time difference between a read and an update operation should that between a CaS operation and the time it takes to return the value read.

When it comes to the number of records inserted into the tree and a read or modify workload of X operations, then the result should be the same as for the B+-tree. Meaning that the high should have little to say as such tree structures do not grow all that tall.

It seems reasonable to assume that performance with more and more threads will scale very good in the Bw-tree data structure as no latching is used.

A comparison

If the Bw-tree keeps its promises then it should have better scaling than the B+-tree. Read operations might have the same efficiency in both data structures. Updates, deletes, and inserts, however, are expected to be performed faster in the Bw-tree as only delta nodes are appended most of the time. If these operations turn out to be slower, it might have to do with the length of the delta chain, but as stated above, threads might not even need to traverse the whole chain in order to execute their operation. The amount of threads used should have a bigger impact on the Bw-tree as the operations, in general, are expected to take less time to perform.

3 Evaluation

Starting with a summary of important implementation details in order to ensure reproducibility, this section also lists and describes the tests performed as well as their results.

3.1 Implementation

The four main components to execute the tests are described below. Starting with how the workload files are generated, then summarising the index benchmarking framework (IBF) client execution process, important implementation details and performance-affecting variables. Next, the execution process as well as implementation details for the IBF's client-, and server-side are presented which includes changes to the original IBF code. Lastly, a short summary of the hardware used is given.

Load generation - Yahoo's Yahoo! Cloud Serving Benchmark

The IBF used Yahoo's Yahoo! Cloud Serving Benchmark (YCSB) to generate different types of workloads. *"The goal of the Yahoo Cloud Serving Benchmark (YCSB) project is to develop a framework and common set of workloads for evaluating the performance of different "key-value" and "cloud" serving stores."*[11].

What is practical is, that for the desired amount of records A_R to insert, the sequence of keys is always the same. In other words, two insert-workloads, one with 100 the other with 1000 records, will have the first 100 records' keys in common. This is true for both monotone and random key-distributions but not the case for read and update operations as more inserted records mean more records to work with. Anyway, this ensures that everyone can get the same workloads to run their tests with. While random key-distributions refer to randomly generated, unique keys up to 19 digits long, monotone key-distributions start from 0 and increase with 1 per operation.

The IBF does only use the keys provided by YCSB and discards potential values. The resulting generated workloads are made up of operation-key pairs, where the operation can be of type insert, update, or read. Values, however, are added as the workload is loaded into the client programs later on. Additionally, read and update operation-keys are solely based on the keys used in the insertion workload file. Hence, these operations types should not fail due to a mission record.

Load generation - Python script

The YCSB is able to create monotone-key insert workloads but does not seem to be capable of creating monotone-key read, update and read + update workloads. The problem is that the keys used are not in sorted order. Thus, a simple python script

was written to handle this situation which can be found in Appendix D. While read and update workload files are straight forward to create, writing N lines of *operation and iteration number* to a file, the 50/50 read + update workload is a bit different. A 50/50 read + update file F created by the YCSB was used as a starting point. Now, the python script would take the operation in line N in F and write *operation and iteration number* to a new workload file. This way, the somewhat random sequence of read and write operations were kept but was now in sorted order. This sort of mixture was kept to see how the data structures would cope with executing two types of operations at the same time as it also reflects a typical use case of a database in which read- and update-operations occur rapidly in a non-predefined operation-order.

Memcached Client

The Memcached client is used to run different workloads against InnoDB via its Memcached plugin. As explained in section 1.4, one cannot directly compare the OpenBw-tree against InnoDB, hence, the variables that cause this issue have to be cut down as much as possible. InnoDB's Memcached plugin allows to circumvent SQL overheads and directly execute operations on InnoDB tables[6]. Other measures were taken into consideration as well, listed below in *"Server properties for performance improvement"*. However, most of them had no noticeable effect compared to the cost of latches and locks.

The Memcached client was made to have an efficient way of running workloads against InnoDB via Memcached and gathering execution results. Its implementation can be found in Appendix A. The following summarises important implementation details as well as performance choices made and gives a rough overview of the execution process for the Memcached Client.

Execution process: A general description of how the tests are performed with Memcached is given in Figure 8. However, there are some details that need further mentioning.

- **Command line arguments**: There are five mandatory command line arguments, represented in Table 3, to chose between different workloads and with how many threads to execute them. Additionally, there are four optional arguments, shown in Table 4, that are used to manipulate the clients' execution procedure in certain ways.
- **MySQL setup and reset**: There are a few things that have to be done in order to prepare a MySQL server for the use of the Memcached plugin[6]. The setup for a running server can be either done manually, via a MySQL client, or implemented in code using the Connector C++ library[4]. That way, SQL commands can be automatically sent to the server without user interaction. This technique can also be used for automated server resets so tests can be executed in repetition.
- **Adding values**: As explained above, the workload files the IBF is working with, do not have any values associated with their keys. Instead, the memory address of the vector holding the keys was used as a starting point, for creating *uint64_t*-type values. The reasoning behind this is not quite clear but the YCSB's values seem to consist of randomly generated strings only. Anyway, this technique was

```
Load insert workload and extra workload into vectors, based on the first four
  arguments shown in Table 3.
while repetitions remaining do
    Start timer
    Find the average share-size per thread for the insert workload
    Find start- and end-index I_S and I_E for each thread based on share-size and thread
      ID
    Start threads
        Connect to Memcached plugin
        Iterate over portion of the insert workload using I_S and I_E
        while not done do
            send query to Memcached plugin
            cache the feedback
        end
        Return thread's collected feedback
    Gather all threads' feedback
    Stop timer
    Repeat the procedure for the extra workload
    Start timer
    ...
end
```

Figure 8: Memcached interface pseudo code
The overall execution process used to run tests against InnoDB via Memcached.

adopted by the Memcached client since that is how the OpenBw-tree was tested originally. Thus, when the workload files are loaded into the program, each key gets associated with a value.

- **Threading**: The pthread library was used in order to split the workloads among multiple concurrent workers[15].
- **Using Memcached**: Connecting and sending queries to the Memcached plugin are done using an open source client library called libMemcached which is an API towards InnoDB's Memcached plugin[1].

Server properties for performance improvement

- **Batch sizes**: The default batch size for how many Memcached operations, read or write[3], are performed during a transaction is 1. The issue, however, is that the continuous commit operations have an impact on performance. Even though higher batch sizes, say 1000, improve the execution time of each read or write operation, they can cause a much larger problem. While a single connection to the Memcached plugin executes the workloads without failed operations, multiple do not. Large amounts of operations fail and the more connections are used, the worse the situation gets. Multiple ongoing transactions, that do not commit after each executed operation, can try to access the same resources inside the B+-tree and change them. The transactions to commit first will get their data overwritten by the later transactions' commits. In other cases, two or more transactions can end up in a deadlock.

Mandatory Arguments		
Argument	**Values**	**Purpose**
Key-distribution	rand, mono	Determines if the insert-workload should consists of random or monotone keys.
Insert operations	10, 50, 100	Selects the amount of records to build the B+-, or Bw-tree. The values translate to records in millions.
Extra operations	10, 50, 100, 200	Selects the amount of extra operations to execute on the data structure. The values translate to operations in millions.
Workload	r, u, d	Determines the type of extra operations to be executed. Read, update, or delete.
Number of client threads	8, 16, 32, 64, 96	The amount of threads running on the client chosen for the tests. Each thread creates its own connection towards the server. 96 matching the number of cores on the server and 8 being the lowest amount chosen to run as no new information is gained by choosing even fewer.

Table 3: Overview over the mandatory command line arguments for the Memcached interface.

One option is to split the operations among the threads based on key and thread ID. However, this is hardly a realistic implementation as single threads can become a bottleneck. Another possibility is to wait and retry but that solution performed even worse than running with batch sizes of 1. On top, this solution did not guarantee an execution without substantial amounts of failed operations. Hence, the default batch sizes for Memcached read and write operations were used for the tests.

Another argument for using the default batch size is behavior. In order to compare both data structures, their behavior should be as similar as possible. The OpenBw-tree essentially commits after each CaS operation, meaning that a new delta node immediately becomes visible to all other worker threads. Batch size 1 creates the same behavior in the B+-tree and thus improves comparability.

- **Buffer pool and buffer pool instance**: InnoDB's buffer pool holds table and index data and the larger it is the fewer table contents have to be reloaded from disk over time which increases performance. The buffer pool is split into instances to improve concurrency as the table data is divided among them[3]. It is recommended to allocate 50% to 75% of system memory to the buffer pool[8]. Hence 256 GB of 502 GB were allocated to the buffer pool which was split into the maximum value of 64 buffer pool instance resulting in 4 GB per instance. As the largest insert workload file has a size of 3.5GB, the entire index will fit into the buffer pool which improves overall performance.
- **RAM Disk**: The IBF and OpenBw-tree are solely in-memory. InnoDB, however,

Optional Arguments	
Argument	**Purpose**
–repeat	Tells the program to repeat the workloads five times.
–insert-only	Tells the program to only execute the insertion workload, regardless of the extra operations' value. Can be combined with the repeat option in which case the B+-tree is reset after each iteration.
–extras-only	Tells the program to execute the insertion operation once and not to reset the B+-tree. Can be combined with the repeat option. Good for getting results for extra operations while avoiding constant rebuilding of the B+-tree.
–no-inserts	Can only be used after the program was at least once executed with the extras-only option as no insert operations are executed in this now.

Table 4: Overview over the optional command line arguments for the Memcached interface.

usually is not as it writes logs and data to disk to ensure durability, even in case of a system crash. Performance wise, this causes a big difference as disk I/O is relatively expensive. To deal with this, the MySQL server runs in a RAM disk, which means that everything that usually would be written to disk is kept in and written to memory instead. This makes InnoDB and the IBF much more comparable as both now run fully in-memory.

- **Flush log commit**: Another InnoDB system variable that was considered in order adjust InnoDB's behavior was innodb_flush_log_at_trx_commit[3]. It is used to control the point in time at which the log buffer is written out to the log file and when this file is flushed to disk, in this case, the RAM disk. 10-million-operations-tests were run with the variable set to 2, causing logs to be written after each commit and flushed every second, and the default value, causing log writes and flushes after each commit. However, the results did not differ at all. This is probably caused by one or more bottlenecks that dominate the costs of log writes and flushes, making them tiny in comparison. In the end, the default was kept.
- **Memcached Plugin Threads**: The Memcached plugin runs with 4 threads by default[7]. The idea was to improve performance and set it to 64 instead, in theory, one thread per client connection. However, doing that seemed to actually have a small negative impact on insert and update operations. This might have caused more worker threads to compete for node access, ultimately slowing down the overall performance. Read operations, on the other hand, did improve and now seemed to match the Bw-tree's performance which also makes sense as readers do not block each other as writers do.

  However, this option was found out about during an extended search for performance bottlenecks as CPU utilisation on both server and client turned out to be relatively low, but at that time, a lot of testing was already done. With little time remaining, the default value was kept for the remaining tests. Anyway, table 5 shows the results for some tests with the default 4 and also 64 Memcached

threads.

	10 million operations		**50 million operations**		**100 million operations**	
Memcached threads	4	64	4	64	4	64
Insert	202.16	207.95	1319.67	1422.90	2795.02	2939.70
Read	32.52	24.99	168.70	125.20	329.66	256.08
Update	247.14	240.27	1236.22	1272.18	2462.97	2544.53
Read + Update	153.51	126.00	764.26	689.20	1519.66	1405.74

Table 5: Random-key results for running the Memcached plugin with 4 and 64 threads.

Index benchmarking framework (IBF) - OpenBw-tree

In order to make the OpenBw-tree test results relevant to Oracle, they had to run between a client and a server as this is the typical use case in the real world. In other words, practically no one uses a database locally only.

Now, the IBF is merely a framework used to tests different data structures, such as the OpenBw-tree, on a single machine. Hence, changes were made to split the framework into a server and client side while keeping the testability. For the most part, the only file touched is the workload.cpp file as it essentially is the IBF's core.

IBF - client side The IBF client side heavily influenced the Memcached client's construction to ensure as similar behavior as possible. This was done to avoid getting results impacted by differences in behavior. The client side's implementation can be found in Appendix B.

Execution process: The Memcached and IBF clients' codes ended up being practically the same as they have the same tasks to do. This is why Figures 8 and 9 look so similar. Communication and the server's implementation are the main reasons for differences.

- **Command line arguments**: The IBF client side has in total six mandatory command line arguments. Five of them are shown in Table 3 and are, also for this client, used for workload and thread count selection. The sixth is a remainder of the original IBF code and defines which data structure or index type to use for the tests, in this case "*bwtree*". The client-server version of the IBF only guarantees to function properly with that data structure.

  The optional arguments differ somewhat from the Memcached client's as only the first two listed in Table 4 were used here as well. The others were originally used for benchmarking and testing purposes. There is also an option for hyper threading which actually produced slightly worse results with a given number of connections than without using this option. Hence, it was omitted during tests.

- **NMI Watchdog**: Upon starting the client program, a warning might occur telling

```
Load insert workload and extra workload into vectors, based on the first four
  arguments shown in Table 3.
Send setup data to server; create and store sockets
while repetitions remaining do
    Start timer
    Find the average share-size per thread for the insert workload
    Find start- and end-index I_S and I_E for each thread based on share-size and thread
     ID
    Start threads
        Use sockets[thread ID] for communication with server; send iteration count
        Iterate over portion of the insert workload using I_S and I_E
        while not done do
            send query to Server side
            cache the feedback
        end
        Return thread's collected feedback
    Gather all threads' feedback
    Stop timer
    Repeat the procedure for the extra workload
    Start timer
    ...
end
```

Figure 9: OpenBw-tree client side pseudo code
The overall execution process used to run tests against the IBF server side.

the user to turn off the NMI watchdog as it "*consumes one hw-PMU counter*". The performance monitoring unit (PMU) is used to observe micro architectural events[2]. An NMI watchdog is used for best-effort deadlock detection and, against expectations, tests to determine its effects on performance gave at best equal results, hence it was decided to keep it turned on.

- **Adding values**: The original IBF implementation of adding values to record keys as the workloads are loaded was kept and, as mentioned above, used for the Memcached client as well so that both data structures would have to store the same type of records.
- **Server setup**: The mandatory and optional command line arguments the client receives need to get passed on to the server before query handling can start. These arguments include the index type, the number of threads and the important optional command line arguments "*–insert-only*" and "*–repeat*".
  Setting up socket connections can fail sometimes, rarely but it happens, hence all connections are established, one per thread, before testing starts. These are reused for all iterations for a given set of workloads, meaning inserts and extras.
- **Threading**: Pthreads were also used for this client to split the workloads among multiple concurrent workers.

IBF - Server side: The server side acts as a bridge between a number of client connections and the OpenBw-tree, executing incoming queries and answering with feedback.

Its implementation can be found in Appendix C.

```
Setup listening server
Receive setup data; create and store sockets, amount based on setup data
while repetitions remaining do
|  Start threads
|      Use sockets[thread ID] for communication with client
|      Receive iteration count for thread
|      while iterations left do
|      |    Receive query data
|      |    Call query-specific Bw-tree method to handle data
|      |    send feedback to client
|      end
|  Repeat the procedure for the extra workload
|  Start threads
|      ...
end
```

Figure 10: OpenBw-tree server side pseudo code
The overall execution process used to handle queries coming from the IBF client side.

Execution process: Figure 10 depicts the pseudo code for the server side's overall execution process. Important details are listed below.

- **Server setup**: The server starts by setting up a listener that handles incoming connection requests. Next, it receives general setup data from the client where the thread number is straight up used to set up an equal number of sockets which the client is requesting by now.
- **Threading**: On the server side, pthreads are used to execute received queries on the OpenBw-tree.
- **Feedback**: The feedback is a simple short string to indicate the outcome of the query based on the OpenBw-tree's feedback to the server side. For instance "*BWTREE_STORED*" or "*BWTREE_NOTSTORED*" for insert operations.

Changes to original IBF code: Besides the split into server and client, there were made some changes to the original code that should be mentioned.

- **OpenBw-tree update method**: The IBF's update method for the Bw-tree index, that calls the OpenBw-tree's update method, always returns true, regardless of the actual feedback from the OpenBw-tree. The reason for this might be that, since delete operations were not tested by the OpenBw-tree's creators, a CaS for an update can fail but the operation will be retried until it eventually succeeds due to the fact that no prior inserted records are ever deleted. Still, the return value was changed to whatever the OpenBw-tree actually returns.
- **Supported thread count**: The original implementation of the IBF supported only 40 concurrent threads. Since the test server has more twice as much cores available, this number was increased to 96. The necessary change is mostly to redefine the maximum number of supported cores and thread-to-core mappings in

the util.h file.

Hardware

Table 6 summarises the client's and server's hardware specifications.

	Client	**Server**
CPU	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz; 4 Cores	Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz 96 Cores (4 x 24 cores)
RAM	48 GB	512 GB
Operating System	Ubuntu 18.04 (Bionic Beaver)	Oracle Linux 7.6 x86_64

Table 6: Hardware overview for the client and server machines.

3.2 Performed tests

In order to determine in what way or in which cases the Bw-tree can be of interest for Oracle, a collection of tests was created. Table 7 summarises the tests that were planned in order to be able to compare the OpenBw-tree and the B+-tree inside InnoDB.

All tests were done with 8, 16, 32, 64 threads but there is a general exception. The OpenBw-tree indicated that there was room for improvement as its graphs in section 3.3 show. Since the IBF binds a thread to a CPU core, the maximum number of threads the OpenBw-tree could be tested with was 96. It should be kept in mind that this is not a doubling of threads but only a 50% increase from 64 threads. So in case, 16 threads finished workload W twice as fast as 8 threads, the same relation will not exist for 64 and 96 threads.

Three sizes of insert workloads were used for the tests, 10, 50 and 100 million records. Each of these was used as a base for a range of read and update workloads. These extra workloads consisted of 10 up to 100 million operations as the second half of table 7 shows.

Additionally, all insert workload sizes were tested with a 50/50 read and update workload of the same size in order to see how the data structures would cope with a mix of operations and which operation type would be the dominating factor when it comes to performance. Also, while the workloads containing solely reads or updates were tested in different sizes, on each size of insert workloads, the 50/50 read and update workloads were not. The idea was to find out if the indexes' heights had a noticeable impact on performance. This was barely the case for the pure update or read workloads as other properties such as bottlenecks, workload- and key-type had much more to say for performance. Thus, these extra tests for the 50/50 read and update workloads were not conducted.

Client	Connection count		
IBF	8, 16, 32, 64, 96		
Memcached	8, 16, 32, 64		
Operation types and operation counts in million			
Insert	**Read**	**Update**	**Read/Update**
10	10/50/100	10/50/100	10
50	10/50/100	10/50/100	50
100	10/50/100	10/50/100	100

Table 7: Overview of the tests planed

Deletes: It was originally planned to test deletes as well as to determine how both the OpenBw-tree and the B+-tree inside InnoDB would cope with the internal restructuring. This would have been particularly interesting since deletes were never tested with the OpenBw-tree, even though the respective methods were implemented and its paper put a lot of focus on how structural changes would affect performance. However, during the tests, a bug in the Memcached plugin or InnoDB caused the delete operations to fail if a certain amount of records already had been deleted. Regardless of the initial workload size that would build the tree, the bug occurred roughly at 1.8 million deleted records. Since this was considered a low priority bug, there was no chance that it would get fixed in time. Hence, deletion tests were not done.

3.3 Results

The series of images below shows the test results. These can be split into two groups. The first is a collection of paired figures based on the number of client-server connections and workload execution time. The first figure shows a one-on-one comparison between the data structures' performances running a 10 million operations workload. The second one contains a two-on-two comparison for the 50 and 100 million operations workloads of the same workload type. Having all three pairs of graphs in one image seemed confusing as the lines would overlap too much, hence this split. Keep in mind that the workload sizes used do not scale linearly. The second group is also made up of paired figures, but these show how the OpenBw-tree and InnoDB's B+-tree scale on different workload types.

As it comes to the use of terminology, when mentioning to the OpenBw-, or B+-tree, then this refers to the data structures inside the IBF and InnoDB respectively. Elements that could have had an impact on the results in either of the tree types will be used for explanation. The number of threads running on the client, that are used to simultaneously send data to the server is referred to as connections. This is done to avoid confusion regarding Memcached and server threads. Anyway, the OpenBw-tree's server's number of worker threads will always equal that of the client's. InnoDB, on the other hand, is called by the Memcached plugin's threads.

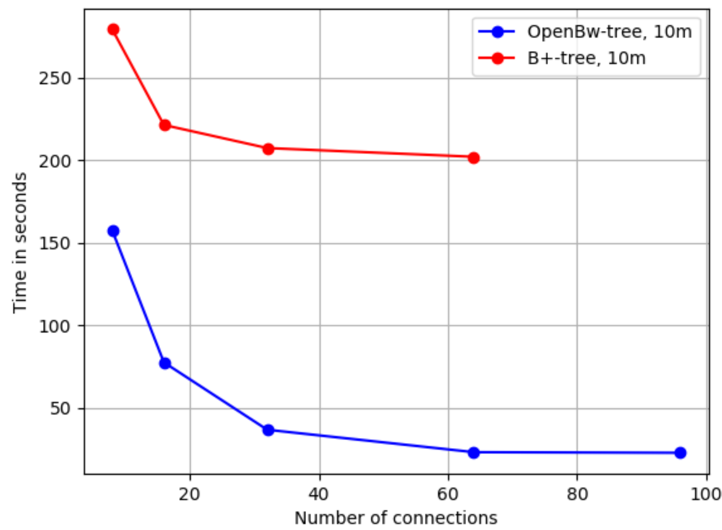


Figure 11: B+-tree (red) and OpenBw-tree (blue) results for **10 million random-key insert operations** done with 8, 16, 32, 64 and 96 connections.

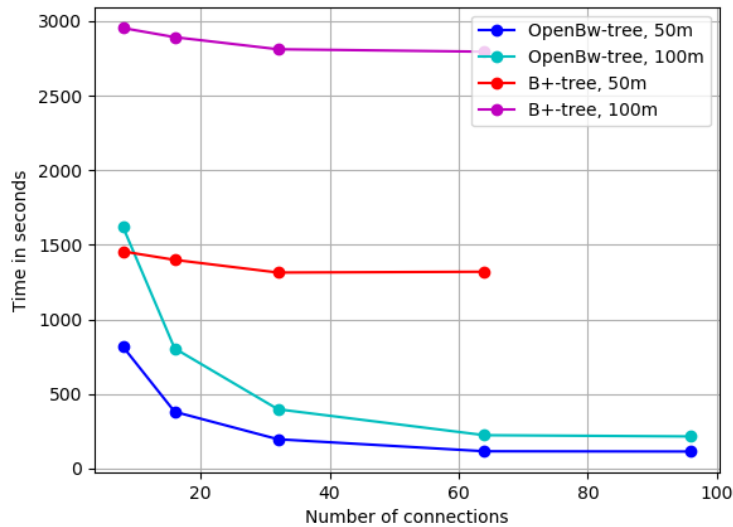


Figure 12: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million random-key insert operations** done with 8, 16, 32, 64 and 96 connections.

Comparison: insert, random-key: Starting with the results for random inserts, Figure 11 and Figure 12 depict the performance graphs for the insert workloads using random keys.

The OpenBw-tree builds its index much faster than the B+-tree. This comes hardly as a

surprise. Installing new records on the server is done quickly using delta nodes. Threads do not block but just use CaS operations. Additionally, the OpenBw-tree's curves show that it scales linearly. This means that having a set of connections C and a workload W, running W again a server with C * 2 connections effectively halves the execution time while doubling the size of W while running with C connections should double the execution time.

The bottleneck the OpenBw-tree encounters from 64 threads and onwards is most certainly caused by the communication between server and client, multiplexing, demultiplexing and data transfer. It is questionable to what degree other factors impact this situation. Threads having to retry their operations, creation and appending of deltas and the height of the index can barely be seen as defining loads compared to network communication. An indication for this is that neither the client's nor the server's CPU's were ever fully occupied, even as the number of connections was increased.

Besides the performance difference, for the B+-tree, there are two things that stand out. Its graphs and own performance. First, even though the 10 million-records graph does not scale linearly, there is some improvement upon using more and more connections. The same cannot be said for the larger workloads as their graphs are almost flat. Since the smallest workload indicates scalability, the cause of this behavior should be the larger workloads themselves. It is possible that a lot of operations' keys are directed to the same leaf nodes inside the data structure which causes continuous restructuring, cache misses and keeps the affected nodes constantly locked by a few worker threads while others have to wait to get access.

Second, the B+-tree hits its bottleneck much earlier than the OpenBw-tree. This time, network communication cannot alone be the cause. This assumption is based on two observations. One, the CPU utilisation is quite low and does barely scale up when the number of connections is increased. Two, the OpenBw-tree's curve ends up much lower, meaning that one or more properties have a slowing effect on the B+-tree as network delay alone should not weight so heavy on performance. Some candidates are the batch size, record distribution, and the locking mechanism but it is hard to determine which of those has the biggest impact. One might think that the Memcached plugin's default 4 threads could be the bottleneck. Thus tests were run with 96 threads to check that theory but the results showed less than 5% performance difference.

Comparison: insert, monotone-key: The results for monotone-key inserts in Figure 13 and Figure 14 are very similar to their random-key counterparts. There are however two interesting observations to be made here.

First, the B+-tree performs much better in this situation. For instance, 100 million inserts with monotone keys on 32 threads take a little more than seventeen minutes while the random-key version takes almost three times as long. This improvement should come from the way the records are distributed among the connections. No connection with ID N_i holds any record-keys smaller N_{i-1}'s largest key and no keys larger than N_{i+1}'s smallest key. In other words, the workload is split onto the connections in a sorted manner. As a result, each worker thread on the server potentially operates on its own

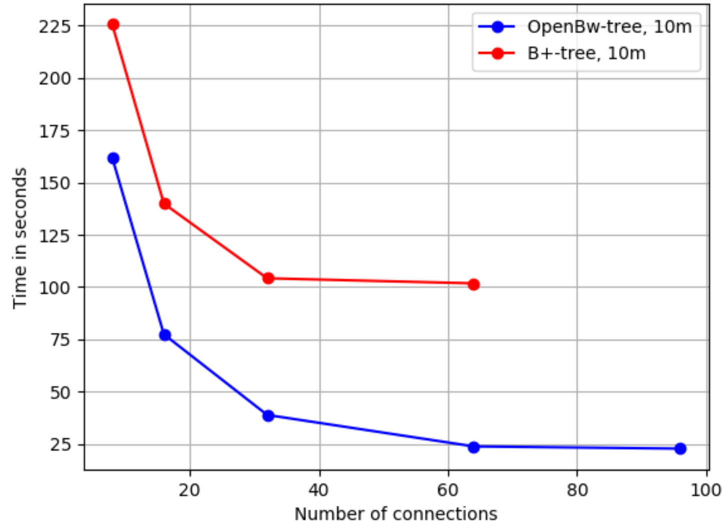


Figure 13: B+-tree (red) and OpenBw-tree (blue) results for **10 million monotone-key insert operations** done with 8, 16, 32, 64 and 96 connections.

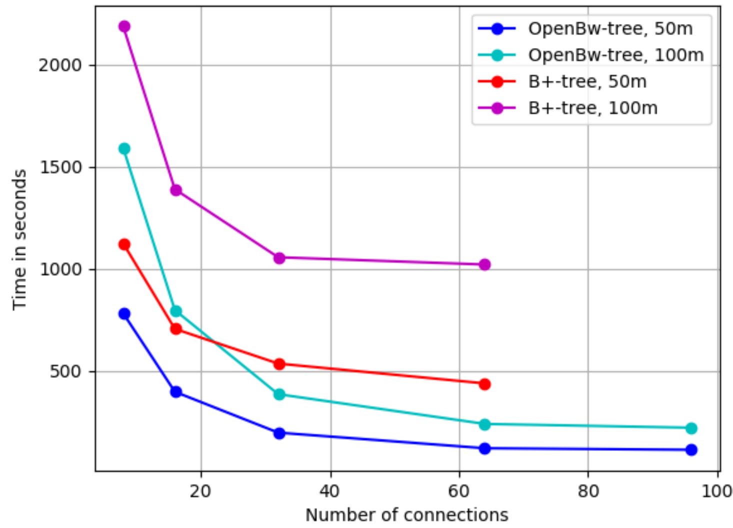


Figure 14: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million monotone-key insert operations** done with 8, 16, 32, 64 and 96 connections.

branch in the B+-tree without interfering with other threads. This will not always be the case due to restructuring activities but it could explain the improved performance. In the end, the B+-tree still cannot outperform the OpenBw-tree due to one or more of the bottleneck-candidates mentioned above.

Second, increasing the number of connections, up to 32, has a significant impact on performance for the B+-tree and does reduce the execution time by up to roughly 64%. The OpenBw-tree's performance, on the other hand, is not topped. However, it does not really improve or change either, even when the keys are split as explained above. It seems like record-key-distribution matters much less as network traffic when it comes to the impact on performance, meaning that insertions, random-, or monotone-key, happen so quickly that worker threads on the server will barely get in each other's way, even less so as the tree grows.

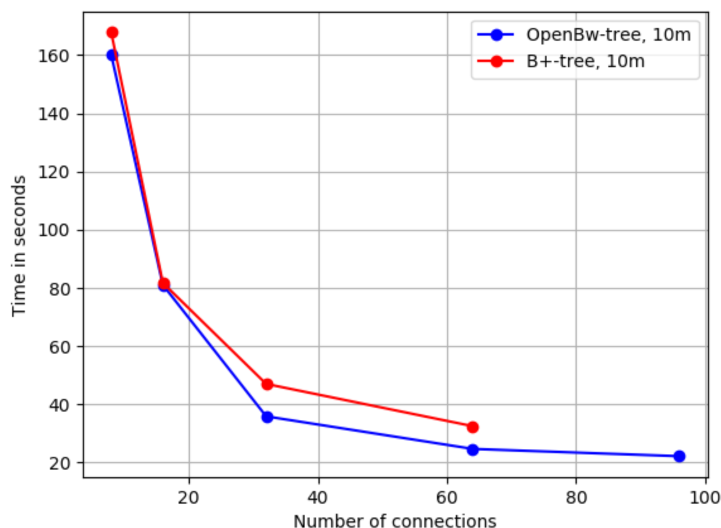


Figure 15: B+-tree (red) and OpenBw-tree (blue) results for **10 million random-key read operations** done with 8, 16, 32, 64 and 96 connections.

Comparison: read, random-key: While the OpenBw-tree dominated insert performances with relative ease, read operations draw a bit of a different picture as Figure 15 and Figure 16 show.

Both data structures are not that far off from each other, especially when using fewer connections. All three pairs of graphs show the same pattern. The B+-tree slightly outperforms the OpenBw-tree using 8 client-server connections, they match at 16 and diverge from thereon in favour of the OpenBw-tree. However, there is an exception. Executing the 10 million operations workload, the B+-tree requires slightly more time to finish using 8 connections and never surpasses the OpenBw-tree. Still, the rest of the graph pair equals the other two. This indicates that the workload has to have a certain size for that the B+-tree becomes the more efficient choice in this situation. There a are couple of things that could cause these results. There is one less bottleneck in the B+-tree. Reading records does not require write-locks. In other words, the tree's worker threads can concurrently retrieve the requested records without interfering with each other. The remaining bottlenecks are the Memcached plugin's threads and the network delay. Tests with 64 threads inside Memcached showed that the B+-tree's random-key read performance would be at least as good as that of the OpenBw-tree. This explains

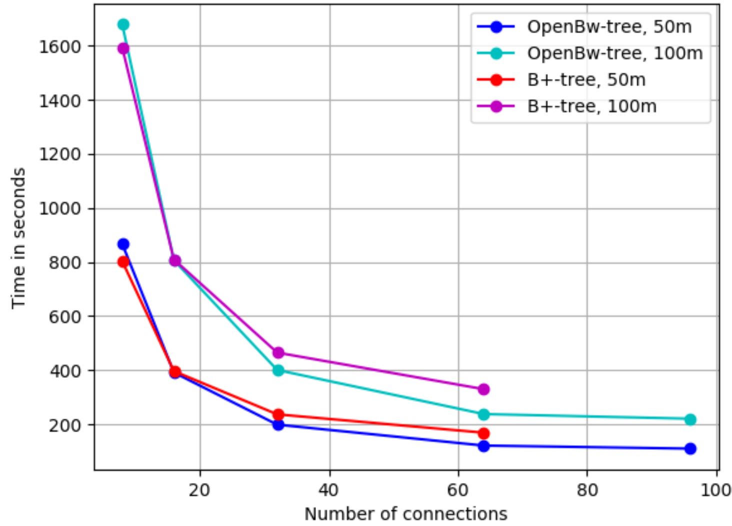


Figure 16: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million random-key read operations** done with 8, 16, 32, 64 and 96 connections.

why the B+-tree performs so well with 8 connections, meaning that Memcached's 4 default threads seem to manage to saturate up to 16 connections before the OpenBw-tree becomes the faster data structure in this case.

However, it is hard to determine what the truth actually is in this case. These results are quite similar so standard deviation might matter here. Table 8 shows the standard deviation values for both data structures running with 8 to 64 and 96 connections. Each value-cell hold the deviations for 10, 50 and 100 million operations. Compared to the total cost of the workloads, these deviations are practically irrelevant. Hence, it seems like the cause for the results lies somewhere else. Maybe, smaller indexes might just be read faster by the Bw-tree. However, this is unlikely since the B+-tree improves, going from 8 to 16 connections to the point where both data structures match, even though Memcached's threads stay constant at 4.

	Data structures	
Connection count	**OpenBw-tree**	**B+-tree**
8	2.13 ; 14.22 ; 35.81	4.26 ; 10.11 ; 11.85
16	3.07 ; 10.27 ; 10.57	2.73 ; 7.51 ; 18.91
32	0.62 ; 1.11 ; 4.53	3.35 ; 13.25 ; 22.17
64	0.58 ; 1.30 ; 1.41	2.37 ; 6.59 ; 12.53
96	0.11 ; 1.79 ; 2.27	-

Table 8: Random-key read standard deviation
A list of the standard deviation for 10, 50, and 100 random-key read operations.

Lastly, it is interesting to see that equally sized read and insert workloads take the same

amount of time to execute for the Bw-tree. It seems like the installation of new delta nodes, occasional node splits and delta node consolidations do not add any noticeable cost at all. On the other hand, network delay might just be a too dominating cost.

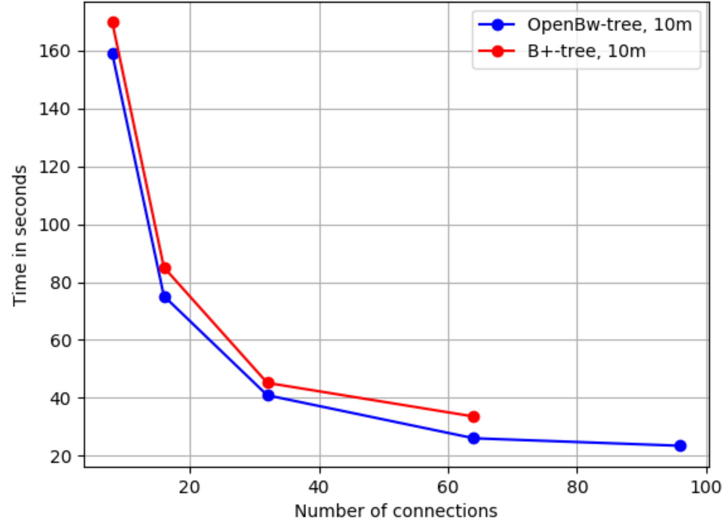


Figure 17: B+-tree (red) and OpenBw-tree (blue) results for **10 million monotone-key read operations** done with 8, 16, 32, 64 and 96 connections.

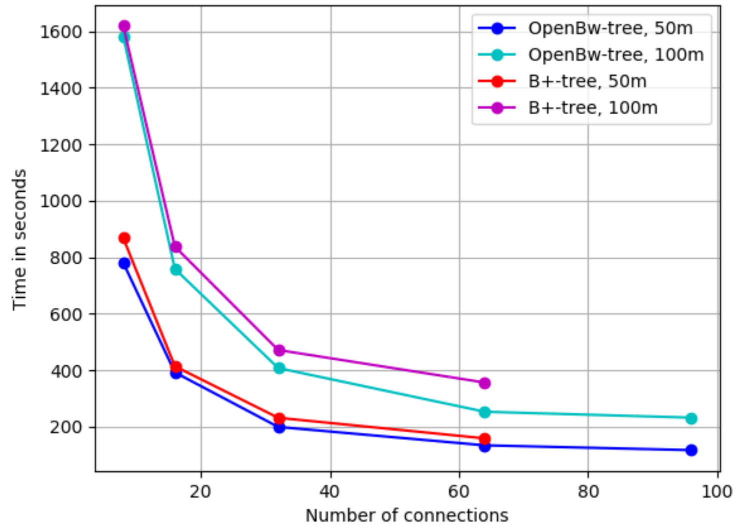


Figure 18: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million monotone-key read operations** done with 8, 16, 32, 64 and 96 connections.

Comparison: read, monotone-key: Figure 17 and Figure 18 depict the read monotone-key workloads' results. Performance wise, there are barely any differences to the random-

key read version, even though caching for monotone-key reads should be much better due to temporal and spatial locality. The small differences in the results could be caused by standard deviation which might explain why the OpenBw-tree surpasses the B+-tree in every case contrary to B+-tree's performance during random-key reads. However, just like in table above for the random-key reads, the deviation was minimal again.

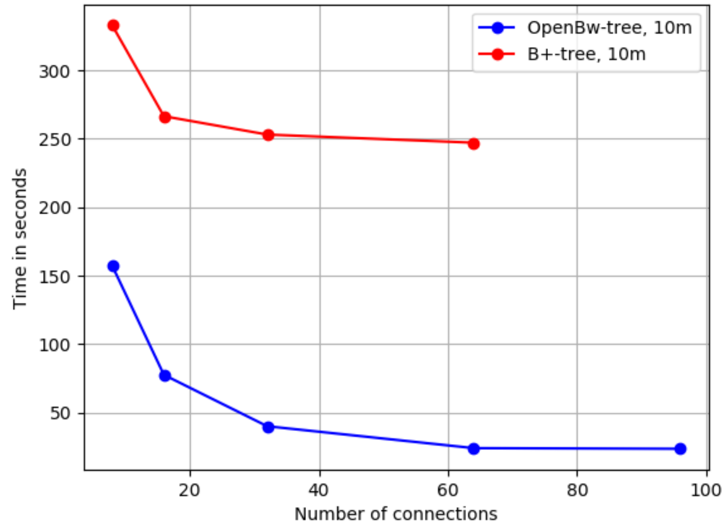


Figure 19: B+-tree (red) and OpenBw-tree (blue) results for **10 million random-key update operations** done with 8, 16, 32, 64 and 96 connections.

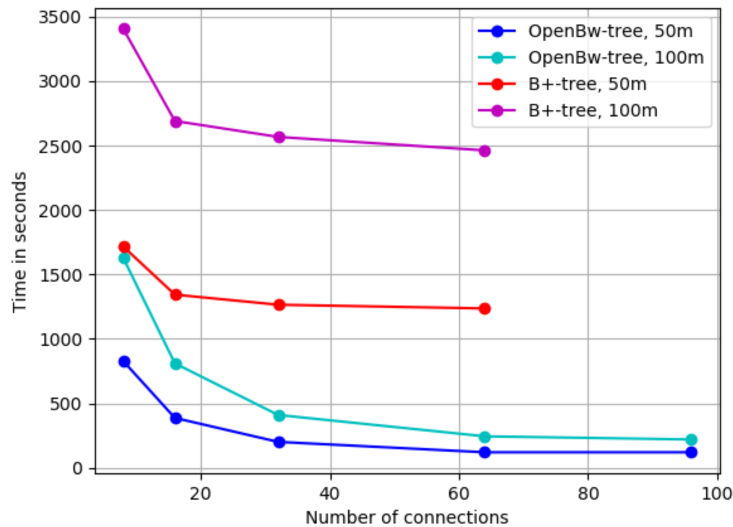


Figure 20: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million random-key update operations** done with 8, 16, 32, 64 and 96 connections.

Comparison: update, random-key: The results for update operations have a strong similarity to the graphs for the random-key insert operations but, as Figure 19 and Figure 20 show but workload size and connection count cause a bit different behavior.

Update workloads, run against the B+-tree, of ten million operations, take longer time to execute than insert workloads of the same size. This difference gets bigger with the number of connections used, being roughly 25% when using 64. However, the trend switches as the workload size increases, where 50 million updates are performed faster than equally many insert operations using at least 16 connections. The situation becomes clearer with workloads of size 100 million. 8 connections perform the inserts 16.6% faster than the update workload, but 16 or more finish the updates up to 20% earlier. This might be caused by the height of the tree during the operations. While the update operations should have a constant execution time since the index does not grow during this type of workload, the inserts do not. An insert operation is cheapest at the beginning of the workload but gets more and more costly as the index and traversal length grows.

There are two things both workload types, inserts and updates, have in common. The B+-tree has a steep performance increase going from 8 to 16 client-server connections before it hits one of the bottlenecks mentioned above. For the update workloads, however, does this boost also exist for the larger ones. This could indicate a high standard deviation for the 50 and 100 million insert workloads but in actuality this is, again, not the case. Anyway, the potential bottlenecks here are the same as for random-key and monotone-key inserts and an increase of Memcached's threads did not change performance too.

The second commonality is the OpenBw-tree's performance as it copes equally well with both workload types and there are hardly any performance divergences. That is because both insert and update operations are executed by appending delta records. The only difference between both of them is that the index changes structurally during insertions and that it grows over time, changing per-operation execution time. But inserts and updates have even more in common execution-wise as inserts and reads. Since the latter pair results in equal performance on the OpenBw-tree, it is no surprise that the former does as well. In the end, the OpenBw-tree outperforms the B+-tree again by quite a margin.

Comparison: update, monotone-key: The OpenBw-tree has, yet again, the exact same performance as with all other workload-key-type combinations. The B+-tree, on the other hand, does present somewhat different results this time. For clarification, it does not perform as good as the OpenBw-tree but there are other things that stand out once these results are compared to other workload-key-type combinations.

As with monotone-key insert operations, monotone-key updates take less time to execute than their random-key counterpart. However, monotone-key inserts are always executed faster than the monotone-key update workload of the same size. This relation did not exist for the random-key version as only the update workload with 10 million operations took longer than the insert workload of the same size. This behaviour might be caused

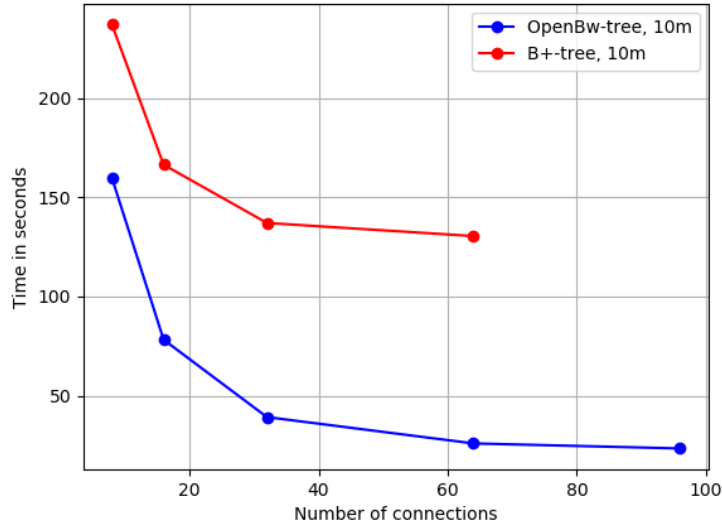


Figure 21: B+-tree (red) and OpenBw-tree (blue) results for **10 million monotone-key update operations** done with 8, 16, 32, 64 and 96 connections.

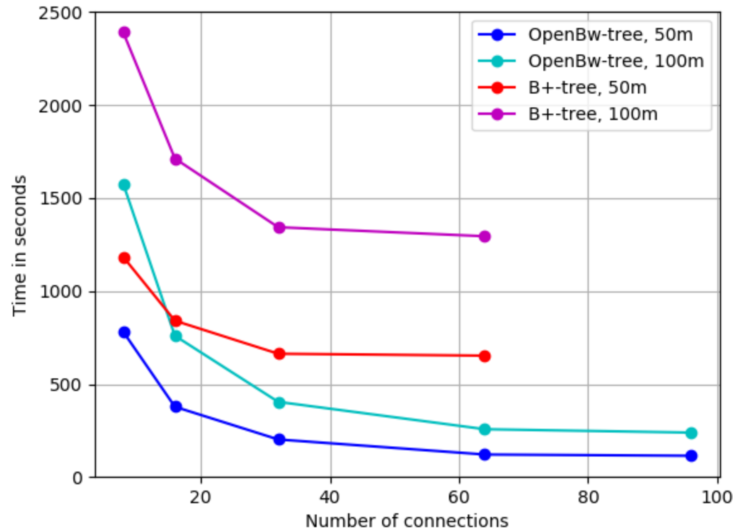


Figure 22: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million monotone-key update operations** done with 8, 16, 32, 64 and 96 connections.

by improved caching due to the use of monotone keys. Each thread can potentially reuse cache lines due to locality in space in time and as a result, execute a given monotone-key insert workload overall faster than an update workload using monotone keys and being of the same size. However, it seems strange that the update workload would not benefit as much from cache usage, but, in the end, a tiny per-operation performance difference

would be enough.

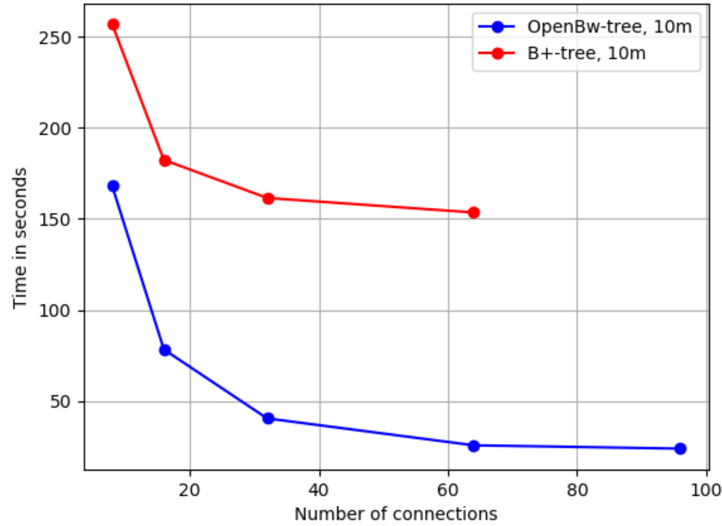


Figure 23: B+-tree (red) and OpenBw-tree (blue) results for **10 million random-key read + update operations** done with 8, 16, 32, 64 and 96 connections.

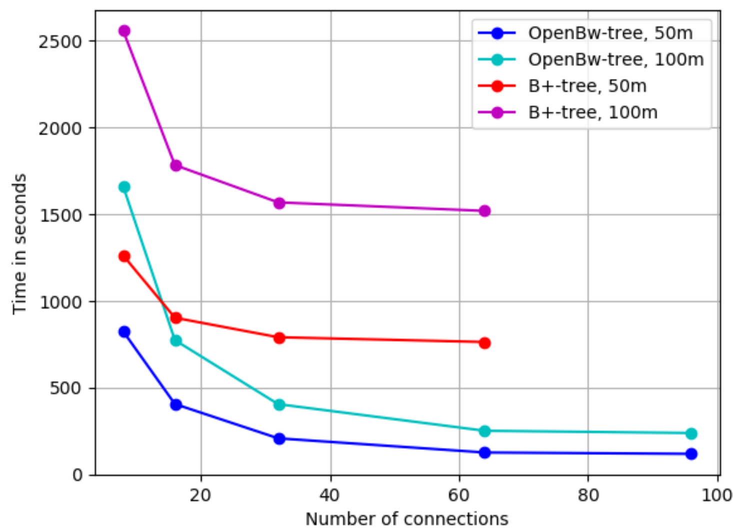


Figure 24: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million random-key read + update operations** done with 8, 16, 32, 64 and 96 connections.

Comparison: read + update, random-key: Figure 23 and Figure 24 depict the results for the 50/50 read and update workloads using random keys.

The OpenBw-tree has practically the same execution time for a combination of reads and

updates as for separate read and update workloads. Reads do not get affected in their execution by updates and simply return a records current state, regardless if a worker thread is updating it or not. Updates did hardly seem to interfere with each other during the pure update-based workloads as they gave the same results as for purely read-based workloads. Now, in this case, there is only 50% of operations that are updates, meaning even less conflict and retry potential. Hence, these results are hardly a surprise.

Even though the B+-tree's results are different, they produce data the matches the 50/50 distribution of the workload very well. For a given amount of connections C_N, the result for a 50/50 workload of size W_S will lie slightly above combined performances, divided by 2, for the separated read and update workloads, of size W_S and connection count C_N. In simpler terms, the 50/50 workload takes 50% of the time it takes to execute both a read and update workload of the same size with a given number of connections. The result lies slightly above the average A of these two values but it comes closer to A the larger the workload size gets.

For 64 connections and the 10 million operations workload, it is about 11% above average while for the 100 million version its is only 7%. A higher-than-average result indicates that updates, as they take longer to execute on the B+-tree, do have a stronger impact on results. This makes sense since update operations require read- and write-locks, blocking all kinds of access to a record. Now, these percentages indicate that the updates' blocking becomes less of a deal the larger the index is. This is only natural as the chance for two operations, of which at least one is an update, to access the same node becomes smaller the larger the B+-tree is. In the end, it is thus no surprise that the Bw-tree outperforms the B+-tree again.

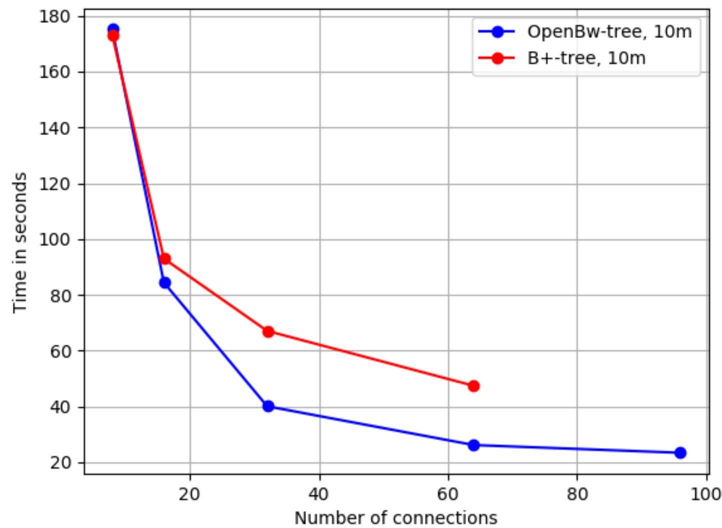


Figure 25: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **10 million monotone-key read + update operations** done with 8, 16, 32, 64 and 96 connections.

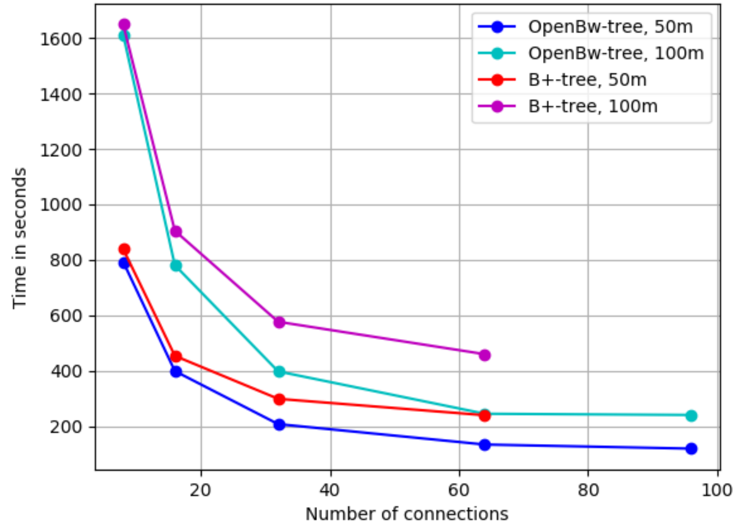


Figure 26: B+-tree (red, magenta) and OpenBw-tree (blue, cyan) results for **50 and 100 million monotone-key read + update operations** done with 8, 16, 32, 64 and 96 connections.

Comparison: read + update, monotone-key: The OpenBw-tree, shown in Figure 25, performs as always, nothing special to see here. However, the B+-tree's results in Figure 26 are unusual. Using few connections, 8 to 16, the graphs practically match the results for the monotone-key read workload, also having almost the same execution time. Additionally, as there is barely a difference to the OpenBw-tree's results, this means that the B+-tree, in this case, seems to execute update operations as fast as read operations. When using more connections, the graphs split up which is likely due to the use of locks. However, there is also a chance that Memcached's 4 threads could have been an issue here as the read operations would have been executed faster using more threads. Another explanation for these results is given at the end of this chapter in *"Scaling properties: read + update, monotone-key"*.

Scaling properties: insert, random-key: Figure 27 and Figure 28 depict the OpenBw-tree's and B+-tree's scaling properties as the number of connection increases on a given set of workloads. The former scales as expected, roughly halving the execution time by doubling the number of client-server connections used. Without significant bottlenecks, as for instance locks, the OpenBw-tree behaves like it theoretically should. The same, however, cannot be said for the B+-tree as it does not scale at all. As explained earlier, bad record distribution combined with a bad node-filling-degree, splits, index restructuring, and cache-misses can be a cause to these results. In the end, the OpenBw-tree and B+-tree improve with 86.7% and 5.4% respectively going from the lowest to their highest amount of connections tested.

Scaling properties: insert, monotone-key: The collected OpenBw-tree results for monotone-key insert operations, shown in Figure 29, barely differ from its random-key

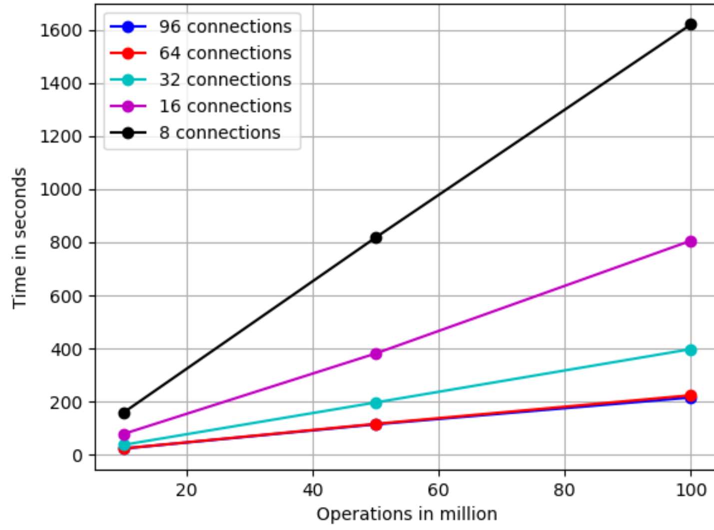


Figure 27: Bw-tree performance scaling for 10, 50 and 100 million random-key insert operations.

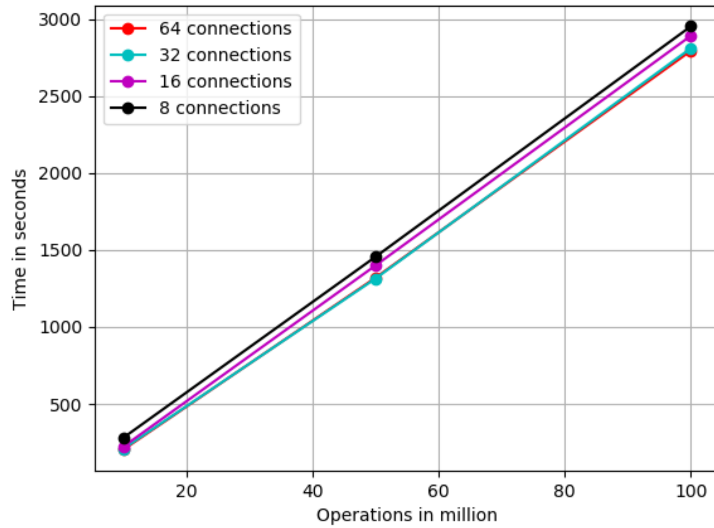


Figure 28: B+-tree performance scaling for 10, 50 and 100 million random-key insert operations.

counterpart. The B+-tree's results, however, collected in Figure 30, do. Starting from 8 connections and moving step-wise up to 64, the performance scales with 36.5%, 24.1% and lastly, 3.3% respectively. This is far from being linear scaling, especially going from 32 to 64 connections which again is hardly any scaling at all but in total a reduction of 53.4% of execution time. The OpenBw-tree improves with a total of 86.2% going from 8 to 96 connections. It should be kept in mind that these values are not definitive due

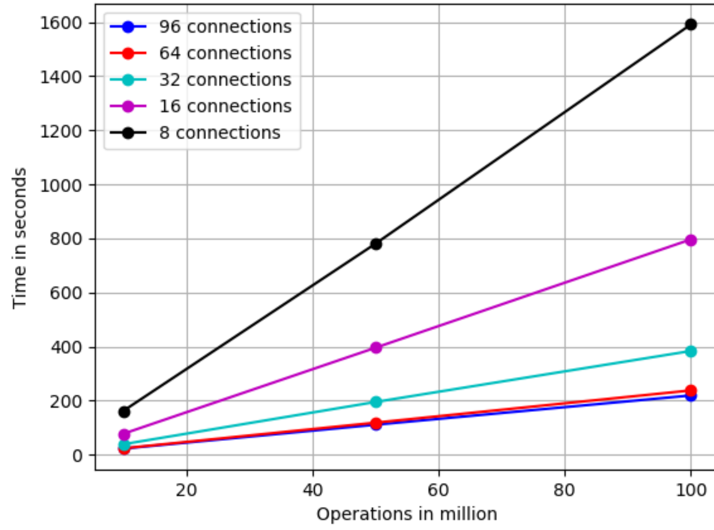


Figure 29: Bw-tree performance scaling for 10, 50 and 100 million monotone-key insert operations.

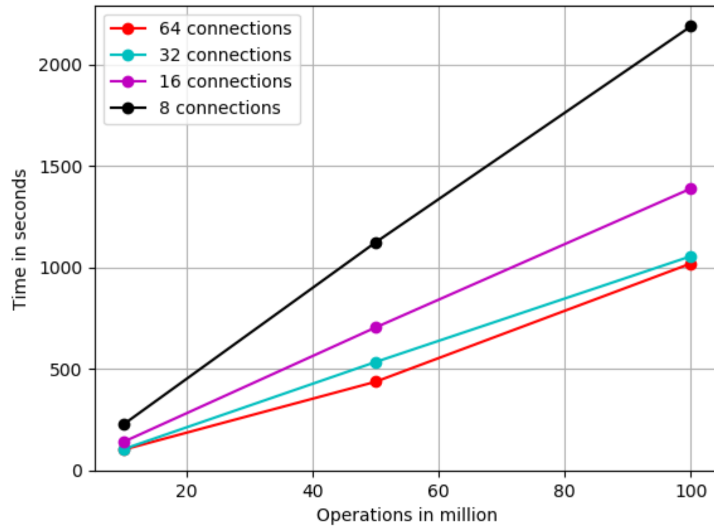


Figure 30: B+-tree performance scaling for 10, 50 and 100 million monotone-key insert operations.

to standard deviation as is easy to see on the 64 connections line in Figure 30 as it is not straight.

Scaling properties: read, random-, and monotone-key: Both random-key, Figure 31 and Figure 32 and monotone-key, Figure 33 and Figure 34, read operations scale relatively equal. The B+-tree scales linearly going from 8 to 16 connections but does

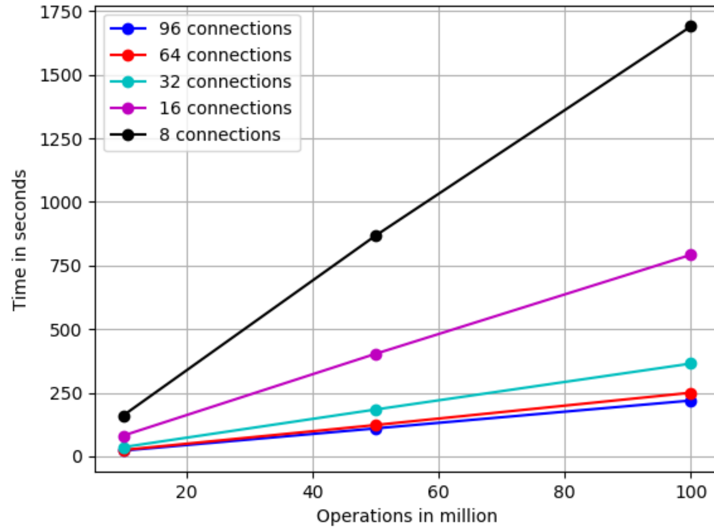


Figure 31: Bw-tree performance scaling for 10, 50, and 100 million random-key read operations.

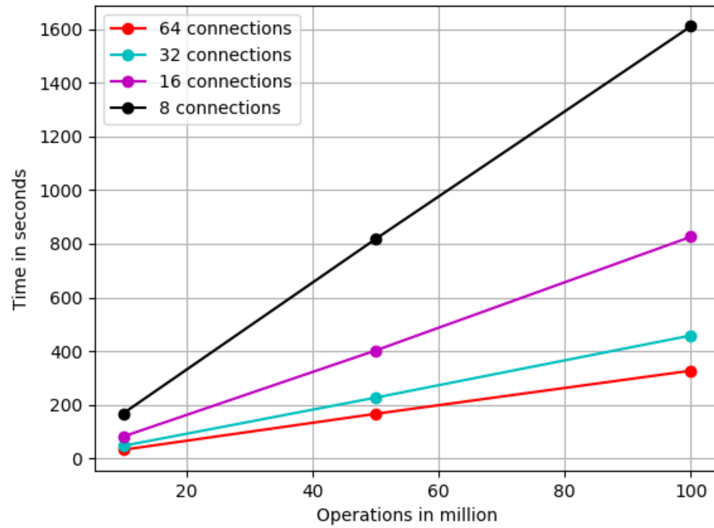


Figure 32: B+-tree performance scaling for 10, 50, and 100 million random-key read operations.

not manage to keep it that way after that as going from 16 to 64 roughly halves the execution time again. In the end, a total time reduction, based on the 100 million operations workload, of 79.9% for random-key and 77.9% for monotone-key respectively.

The OpenBw-tree scales a little bit more than linearly going from 8 to 16 connections running the random-key workload but, just as the B+-tree, is unable to keep it that

way. The monotone-key version, however, roughly scales lineary all the way up to 64 connections. The total improvements on the OpenBw-tree for random- and monotone-keys are 86.8% and 85.2% respectively.

However, as was said before, even though it looks like one or the other data structure scales better in this or that situation, the results are not far off from each other and due to standard deviation, these are not definitive. This includes the fact that the results show partially better and worse than linear scaling.

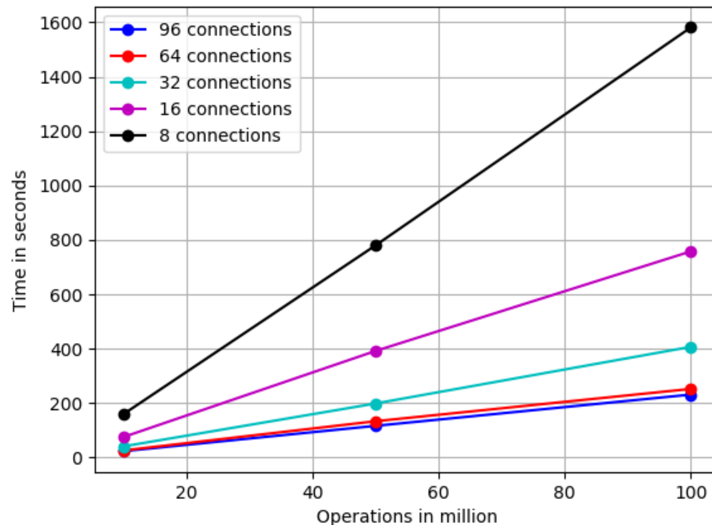


Figure 33: Bw-tree performance scaling for 10, 50, and 100 million monotone-key read operations.

Scaling properties: update, random-key: The results for scaling on random-key updates are shown in Figure 35 for the OpenBw-tree and in Figure 36 for the B+-tree. The former practically scales linearly all the way up to 64 connections. The B+-tree's scaling performance lies somewhere between its insert and read scaling performance. This makes sense since updates, on average, are cheaper than inserts as they do not change the data structure, but are more expensive than reads since they do have to the change records. Thus, it seems like, for the B+-tree, there is a relation between operation cost and the scaling potential which is capped by one or more of the previously mentioned bottlenecks. This might as well be true for the OpenBw-tree as well but it is hard to say as all workload-key-type combinations end in equal results.

Anyway, additional Memcached threads had no effect again indicating that the low scaling is once more caused by the use of locks inside the B+-tree. In the end, the improvement from 8 to 64 threads is merely 28.6% on the 100 million operations workload whereas the OpenBw-tree improves with 87.6%, clearly outperforming the B+-tree once again.

Scaling properties: update, monotone-key: The B+-tree's scaling graphs for

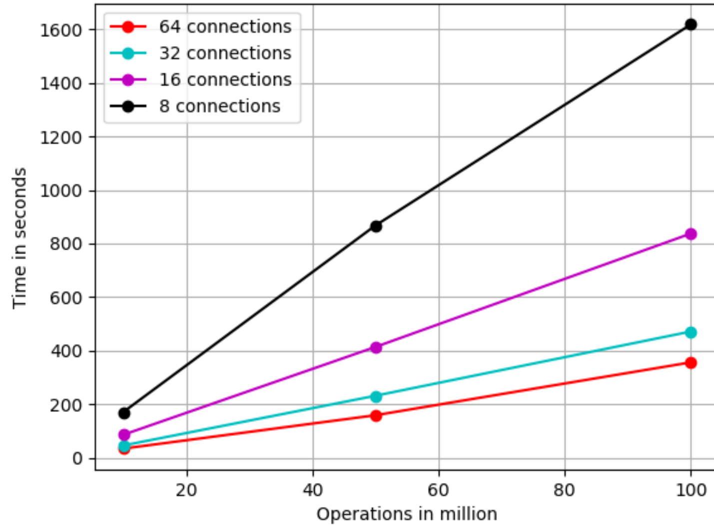


Figure 34: B+-tree performance scaling for 10, 50, 100 million monotone-key read operations.

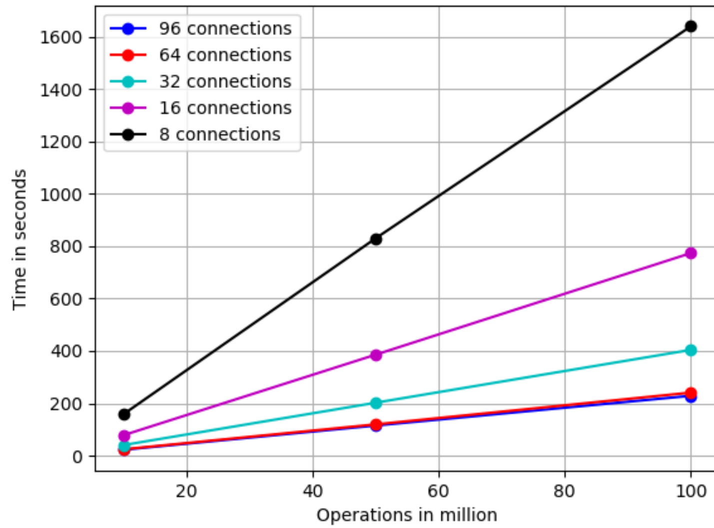


Figure 35: Bw-tree performance scaling for 10, 50 and 100 million random-key update operations.

monotone-key updates, depicted in Figure 38, look a lot like the random-key version except for the time required for execution. The total improvement going from 8 to 64 connections on 100 million operations is 45.8%. This is more scaling than with random-keys which supports the cost-scaling theory mentioned above as monotone-key updates are cheaper due to the key-distribution. Lastly, the OpenBw-tree, shown in Figure 37, scaled with 83.7% between 8 and 96 threads.

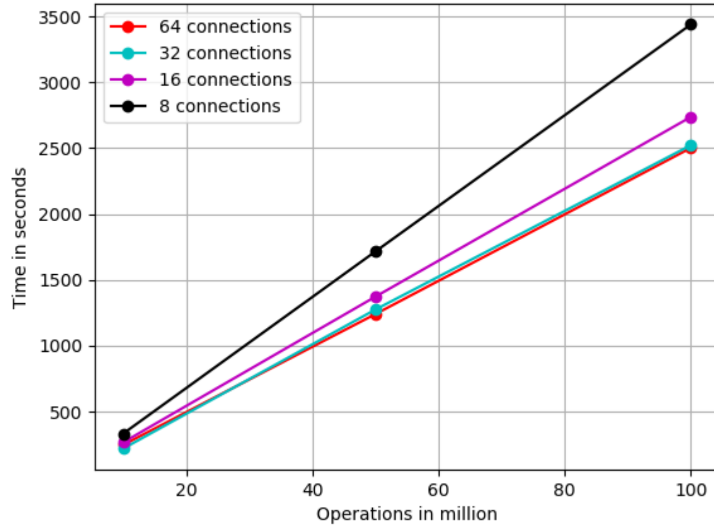


Figure 36: B+-tree performance scaling for 10, 50 and 100 million random-key update operations.

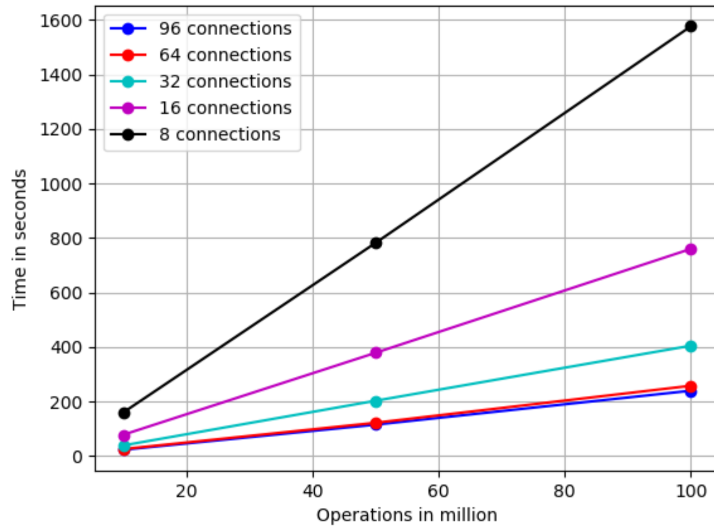


Figure 37: Bw-tree performance scaling for 10, 50 and 100 million monotone-key update operations.

Scaling properties: read + update, random-key: Since there is barely a difference, performance wise, between read- and update-operations, it is no wonder that a combination of both scales just as well as the other two do individually on the OpenBw-tree, shown in Figure 39. The B+-tree's scaling performance, shown in Figure 40, lies almost perfectly between its results for pure update and pure read workloads shown in Figure 36 and 32 respectively. To be more specific, the B+-tree executes 100 million updates with

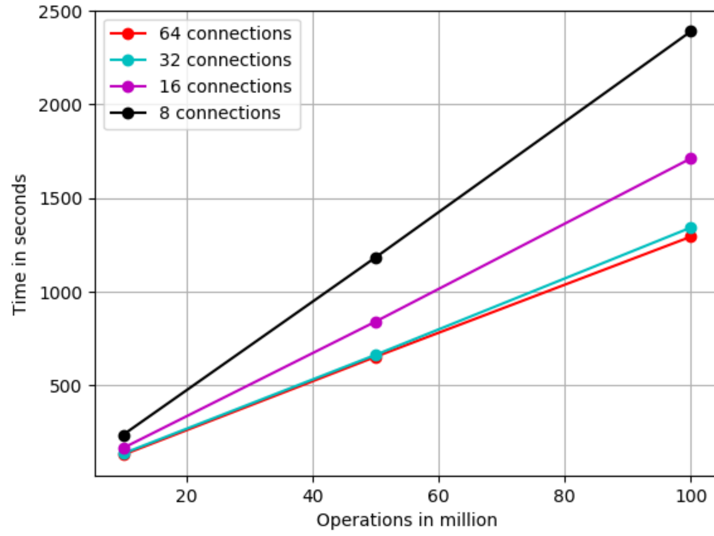


Figure 38: B+-tree performance scaling for 10, 50 and 100 million monotone-key update operations.

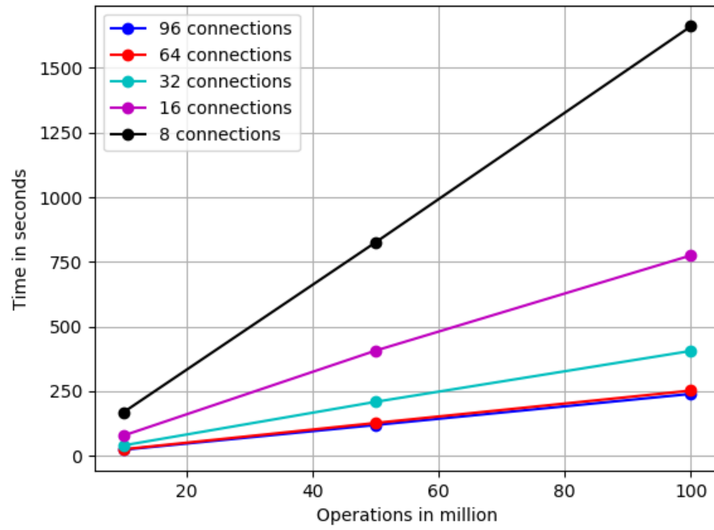


Figure 39: Bw-tree performance scaling for 10, 50 and 100 million random-key rand + update operations.

random keys in roughly 2500 seconds using 64 connections. It needs a little bit more than 2500 seconds and roughly 1500 seconds for the 50/50 read + update workload of the same size using 8 and 64 connections respectively. Lastly, 100 million read operations take approximately 1600 seconds to execute with 8 connections. It makes sense that the scaling graphs are in between as the workload consists of 50% of each operation type. However, the fact that a scaling graph-set ends where the next one starts should be a

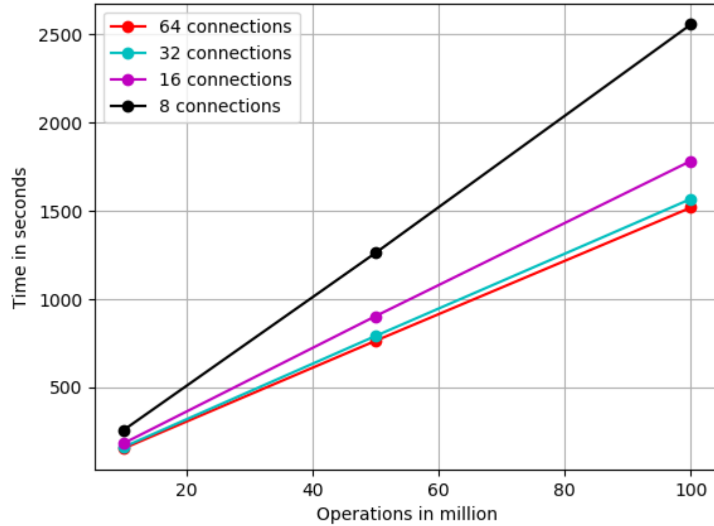


Figure 40: B+-tree performance scaling for 10, 50 and 100 million random-key read + update operations.

coincidence since the OpenBw-tree as well as the 50/50 read + update monotone-key version do not share this outcome.

Another interesting observation is that 50 million read operations and 50 million update operations, done with random-keys and a set of connections C_S needs exactly the same time to execute as the 50/50 read + update 100 million operations workload using C_S connections. This counts for both data structures.

In the end, the OpenBw-tree and B+-tree improve with 84.8% and 45.9% on the 100 million operations workload respectively.

Scaling properties: read + update, monotone-key: The OpenBw-tree's scaling potential, shown in Figure 41, offers nothing new, even in this last case. It scales with a total of 85.1%. The B+-tree scales up to 72.2% and does present an interesting case. Figure 42 does hardly look different to any of the other scaling graphs. It shows better scaling for the B+-tree for the monotone-key 50/50 workload than for the random counterpart. However, there are two sets of comparisons that result in an interesting observation.

First, the 50/50 read + update monotone-key workloads scale and perform almost as good as the pure read workloads using monotone keys, see Figure 34. It almost looks like the update operations have very little to say compared to the read operations which seems counter-intuitive. Looking at the pure update scalings in Figure 38, it does scale much worse than the 50/50 and pure read workloads.

Second, while one could add together the results for 50 million reads and 50 million

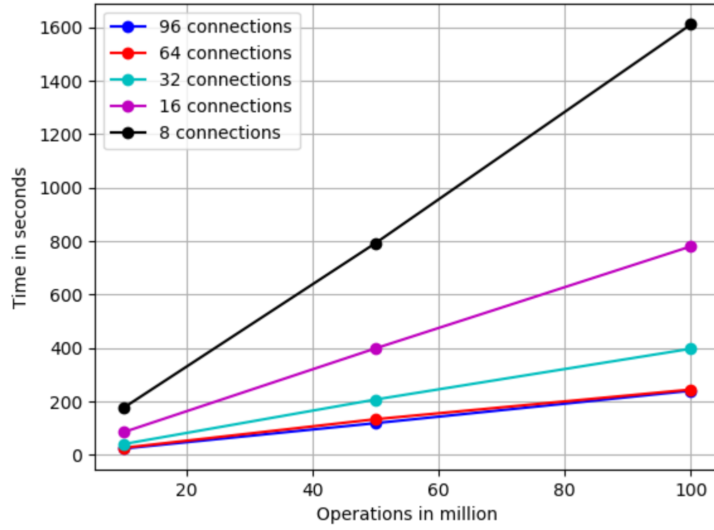


Figure 41: Bw-tree performance scaling for 10, 50 and 100 million monotone-key read + update operations.

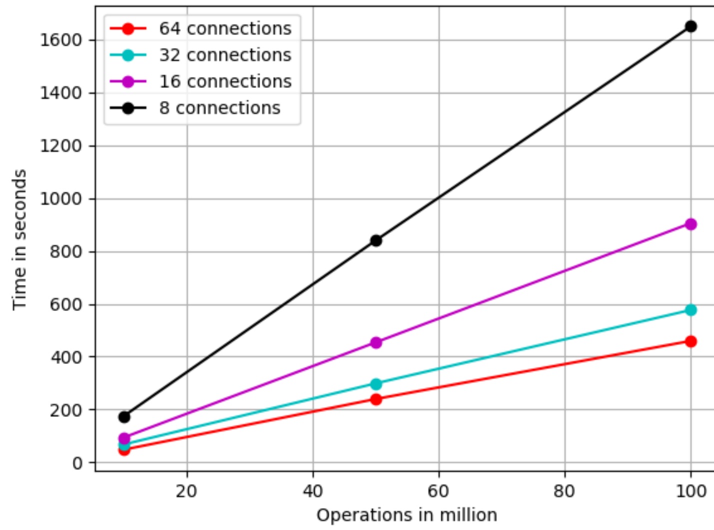


Figure 42: B+-tree performance scaling for 10, 50 and 100 million monotone-key read + update operations.

updates using random keys to get the result for the 50/50 read + update 100 million operations workload, the same principle does not apply for the 50/50 monotone-key workload. In fact, the sum is lower than it should be. Since the OpenBw-tree had no different results than usual, there cannot be anything wrong with the workload-files themselves.

The theory mentioned above, regarding the relation between cost and scaling, could explain this situation. Monotone-key workloads do, in general, perform better than their random-key counterparts due to the key-distribution. Read workloads do perform fastest on the B+-tree, compared to other operation types, as the worker threads on the server do not block each other. Update operations, however, cause the threads to do so. While the 50 and 100 million updates workloads are executed on indexes having 50 and 100 million records respectively, the 50/50 100 million monotone-key workload executes only half as much updates as there are records in the index, resulting, theoretically, in much less idle time for the worker threads. Now, since there is less idle time, meaning operations take less time to execute, the 50/50 workloads might be able to compensate the update-operation's longer execution time to roughly scale as good as the pure read monotone-key workloads.

4 Discussion

This section starts by gathering the results of section 3.3 that will be used to summarise the findings. Lastly, a short analysis of how the limitations influenced the results is given.

4.1 Overall results

The four figures below, Figure 43, 44, 45 and 46, show all performance test-, as well as all scaling-results for the OpenBw-tree and B+-tree gathered in one image each.

The OpenBw-tree's images depict how equally well it handled all workloads due to the lack of serious bottlenecks like in the B+-tree. As a result, scaling was also relatively linear for all workloads-key-combinations.

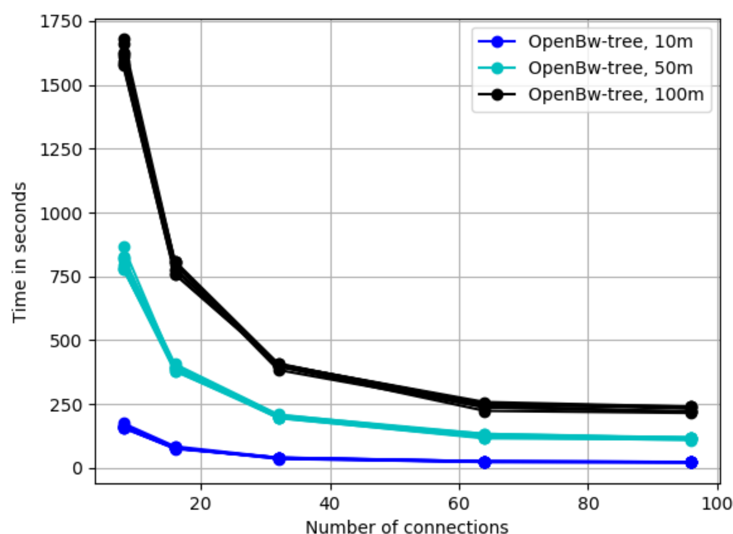


Figure 43: Bw-tree performances for all **10, 50 and 100 million monotone-, and random-key workloads with 8, 16, 32, 64 and 96 connections**.

There were, however, strong differences in the results for InnoDB's B+-tree. While the performance results for 10 million operations of any workload were relatively similar in terms of execution time, the same cannot be said for the larger workloads, 50 and 100 million operations, as they more or less split up into three groups each. These groups, starting with the slowest performing workloads, consist of random-key insert and update workloads, followed by the second group being made up of monotone-key inserts, updates, and the random-key read + update workload. The last and best performing group consists of random-, and monotone-key reads as well as monotone-key 50/50 read + update workload.

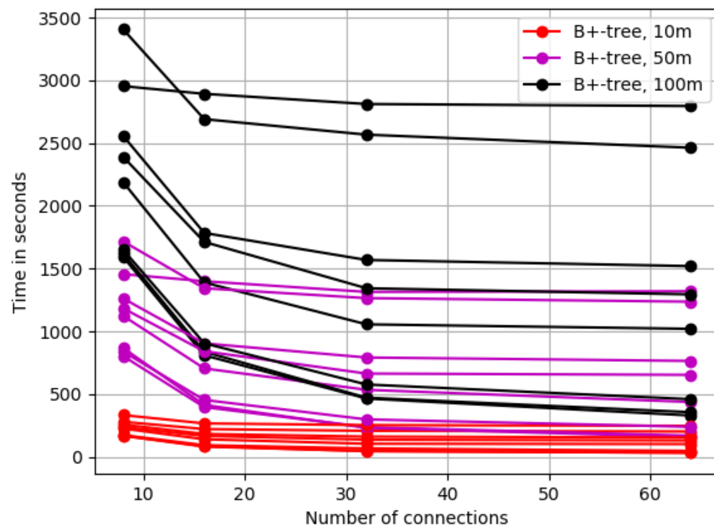


Figure 44: B+-tree performances for all **10, 50 and 100 million monotone-, and random-key workloads with 8, 16, 32 and 64 connections**.

The B+-tree's scaling properties can hardly be put into groups at all as the graphs are all over the place. The only thing that can be said in general is that the more connections are used, the more the performance improves.

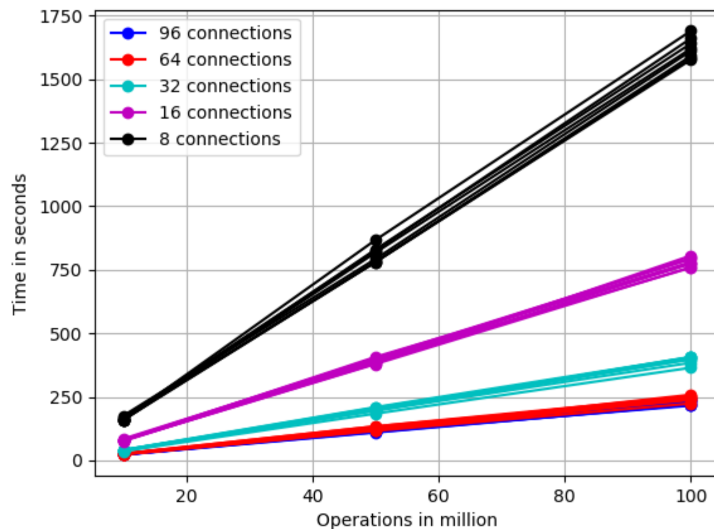


Figure 45: All Bw-tree performance scalings for 10, 50 and 100 million monotone-, and random-key workloads.

Inserts: All in all, the OpenBw-tree dominates all insert-workload key-type combinations, regardless of the number of connections used, the workload size, or key-type.

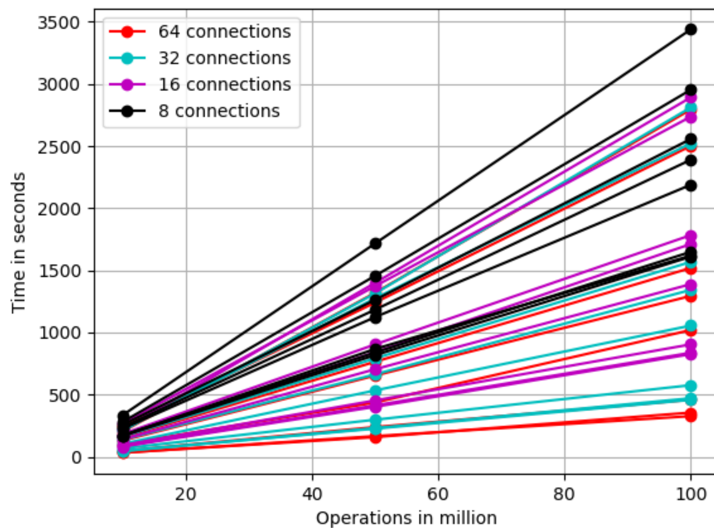


Figure 46: All B+-tree performance scalings for 10, 50 and 100 million monotone-, and random-key workloads.

While monotone-keys gave better results, random-keys are more realistic as data usually is not sorted by key before insertion into the database.

Due to latching, the B+-tree cannot keep up with the OpenBw-tree when it comes to scaling. However, both data structures are not that dissimilar in theory. The current, main difference should be the lack of transaction-support in the OpenBw-tree. Implementing it would either require the use of latches and locks, as in InnoDB's B+-tree, or other sophisticated solutions[13].

Reads: Both data structures cope practically equally well with read operations, regardless of workload size or connection count. However, Memcached had to run with more than the 4 default threads for the B+-tree to perform as good as the OpenBw-tree. It did turn out that read operations were the only ones benefiting from that. Key type, monotone or random, had little to say for the results on both data structures indicating that there were few cache misses in general, also for the random-key reads.

Both data structures already perform equally during reads only. Hence, it is questionable if the implementation of transactions in the OpenBw-tree would change that.

Updates: The OpenBw-tree, again, outperformed the B+-tree. Updates use read- and write-locks which are a much more dominant bottleneck for the B+-tree than the network delay. As with inserts, an OpenBw-tree supporting transactions would probably perform worse on the update workloads.

Reads and Updates The 50/50 read + update workloads, again dominated by the OpenBw-tree's performance, did show some useful results as it was possible to compare

them to the pure read and pure update workloads. Increasing Memcached's threads did also somewhat improve the performance due to the 50% read operations. In the end, the random-key version was dominated by the cost of the update operations while the monotone-key version almost negated the update's extra cost compared to the read's.

Connections: The number of connections usually had the intended effect of increasing performance and thus reduce execution time for a given workload and also helped find the data structures' scaling capabilities. There were, however, some exceptions. Due to latches and Memcached's threads, the B+-tree varied in behaviour under different workloads. Monotone-key workloads and read operations made the most use of extra connections while the insert workload, using random keys practically could not make use of them at all.

The OpenBw-tree, in general, did hardly improve anymore using more than 64 connections. This is also the case the B+-tree running random-, and monotone-key workloads, otherwise it tended to stop improving somewhere between 16 and 32 connections due to one or more bottlenecks.

Workload size: The different workload sizes were useful to determine how performance and scaling capabilities would develop as more and more operations had to be executed. It also showed that the workload size in itself is unlikely to become a bottleneck by overloading the data structures.

Performance dampers The listing below summarises things that had an impact on performance but that were not included by choice. For instance, using batch size 1 on both data structures and the record distribution, referring to random and monotone keys, do not count here. Starting with the most important ones, marked bold, but also mentioning minor factors.

- **OpenBw-tree**
  - **Network delay**: Most certainly, network delay was the most costly part of the tests run with the OpenBw-tree as it effects every single query from the client and feedback from the server. As both server and client CPU's were never fully occupied, it is likely that the network delay was the bottleneck in this case. However, keep in mind that this cost should be roughly equal for both client-server pairs as they transfer the same data back and forth.
  - Cache misses/invalidation: Due to the mapping table, cache misses should be reduced by a lot as nodes are not modified in place.
  - Retries: Workloads of 10 million operations running with 96 connections had a little over 2% operations retries for inserts, close to 0 for reads and 1.4% for updates. These numbers would be lower the fewer connections are used as the worker threads could interfere less with each other due to CaS operations. Increasing the workload size would also increase the number of retires, but percentage wise, these values would stay the same. In the end, as the number of retries was small compared to the actual workload size, the effect on performance was very limited.

- **B+-tree**
  - **Latches and locks**: As using more Memcached threads had no effect on most workload's performances, the use of latches and locks, enabling InnoDB to implement transactions, is the most likely candidate for the bottleneck with the biggest performance impact on the B+-tree. However, it is not clear if this bottleneck resides somewhere in the Memcached plugin itself or in InnoDB.
  - **Network delay**: Network delay was only a partial bottleneck for the B+-tree as most workload types struggled more with latching and locks than with the transfer rates. Only read-related workloads, that came close to the OpenBw-tree's performance, were impacted by the network delay. This might also, to some degree, count for the 50/50 read + update workloads.
  - **Cache misses/invalidation**: It is a little bit hard to determine the actual effects of cache misses as locks and latches are also involved in delaying the execution process. However, update- and insert-operations modify in-place, which potentially causes a lot of readings from memory rather than cache.
  - Memcached threads: Even though using more threads had some impact on read operations, in general, the overall performance was barely impacted as the numbers in table 5 show.

4.2 Limitations

Original Code: The fact that Microsoft's original code was not available for this project had somewhat of an effect on the results. Since the team at Microsoft Research also build a storage engine for their Bw-tree implementation a more reliable comparison could have been possible.

Storage engine vs In-memory: There are a few issues that arise when comparing a purely in-memory standalone data structure with a fully fledged storage engine.

- **Transactions**: The lack of transaction support in the OpenBw-tree is probably what hurts the attempt of a comparison the most as this is probably what caused the highest costs in InnoDB's B+-tree during insert-, and update operations.
- **Logging**: Even though writing the log buffer and flushing log file seemed to barely have an effect on performance, it is also something the OpenBw-tree lacks. Additionally, a more relevant comparison for Oracle would mean to make the OpenBw-tree do undo and redo logging.
- **Persistence**: Another property that the OpenBw-tree is lacking is persistence due to it being solely in-memory. Flushing the buffer pool to disk every now and then, combined with redo and undo logs would provide recoverability after a system crash or blackout.
- **Multi-version concurrency control (MVCC)**: Lastly, InnoDB uses multi-version concurrency control to improve performance and avoid lock contention. The OpenBw-tree also uses database snapshots for each operation. However, it

is unclear in what way they differ from InnoDB's MVCC, especially since the OpenBw-tree lacks transaction support.

5 Conclusion

5.1 Conclusion

A team at Microsoft research published their test results in 2013 for a new B-tree type, the Bw-tree, made for modern hardware systems. In 2018, another group of researchers questioned the claims that were made five years before as they found not nearly as good results as Microsoft did. Now, Oracle had an interest in finding out what the Bw-tree's potential could be for their own systems, more accurately, InnoDB.

Aside from the test results, this thesis also provides a detailed Bw-tree theory part. It is a combination of the descriptions of the Bw-tree's core elements, features, and function given in the papers that were the background for this theses. Additionally, the OpenBw-tree's code was used to fill gaps in the details as well.

When it comes to the practical results, so did the OpenBw-tree as expected, outperforming InnoDB's B+-tree in practically every case due to its linear scaling capabilities. However, it should be kept in mind that there are some important differences between the data structures, most and foremost the OpenBw-tree lacking transaction support. The B+-tree's scaling capabilities, on the other hand, seemed heavily dependent on per operation cost. Thus, insert operations scaled the least, followed by updates, while reads were practically able to match the OpenBw-tree's performance.

Even though the OpenBw-tree often outperformed the in-memory B+-tree implementation in its paper, it is questionable how much Oracle would gain by integrating a Bw-tree into InnoDB. The main issue could be that the Bw-tree would be pressed into the structures that were not built for an index of its kind, being modern hardware oriented, thus losing potential. The only way to avoid this might result in substantial changes of InnoDB itself. So rather to integrate the Bw-tree into InnoDB, InnoDB would be rebuilt for the Bw-tree.

5.2 Further work

Further work can go in two directions. It is possible to essentially continue the work done for this thesis or start in a different way entirely.

Continuing It is possible to take the OpenBw-tree as a starting point and integrate crucial functionality, like transaction support, logging, persistence, and MVCC via InnoDB. Oracle could find out how well InnoDB and the Bw-tree work together by integrating a prototype into the storage engine to get more realistic test results. The issue here is that the index was made with modern hardware specifications in mind, rather using CaS operations than latches.

Restarting The second option would be to restart either from scratch or at least reuse the OpenBw-tree's code and build a new, modern-hardware-oriented storage engine on top of it as done by the researchers at Microsoft[12]. However, this could easily be a cost-intensive project with somewhat uncertain outcome as Microsoft never published the code for their implementations.

6 References

[1] B. Aker. libmemcached, 2011. URL https://dev.mysql.com/doc/refman/8.0/en/innodb-memcached-setup.html.

[2] A. B. Aman Singh. Performance monitoring unit. URL http://rts.lab.asu.edu/web_438/project_final/Talk%209%20Performance%20Monitoring%20Unit.pdf.

[3] O. Corporation. 15.13 innodb startup options and system variables, 2019. URL https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html.

[4] O. Corporation. Mysql connector/c++ 8.0 developer guide, 2019. URL https://dev.mysql.com/doc/connector-cpp/8.0/en/.

[5] O. Corporation. 15.7.1 innodb locking, 2019. URL https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html.

[6] O. Corporation. 15.19 innodb memcached plugin, 2019. URL https://dev.mysql.com/doc/refman/8.0/en/innodb-memcached.html.

[7] O. Corporation. 16.2.2.1 memcached command-line options, 2019. URL https://docs.oracle.com/cd/E17952_01/mysql-5.6-en/ha-memcached-cmdline-options.html.

[8] O. Corporation. 8.12.3.1 how mysql uses memory, 2019. URL https://dev.mysql.com/doc/refman/8.0/en/memory-use.html.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015. ISBN 0133970779, 9780133970777.

[10] G. Graefe. A survey of b-tree locking techniques. *ACM Trans. Database Syst.*, 35 (3):16:1–16:26, July 2010. ISSN 0362-5915. doi: 10.1145/1806907.1806908. URL http://doi.acm.org/10.1145/1806907.1806908.

[11] Y. Inc. Yahoo cloud serving benchmark, 2010. URL https://research.yahoo.com/news/yahoo-cloud-serving-benchmark.

[12] J. Levandoski, D. Lomet, and S. Sengupta. Llama: A cache/storage subsystem for modern hardware. VLDB - Very Large Data Bases, August 2013. URL https://www.microsoft.com/en-us/research/publication/llama-a-cachestorage-subsystem-for-modern-hardware/.

[13] J. Levandoski, D. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. IEEE, April 2013. URL https://www.microsoft.com/en-us/research/publication/the-bw-tree-a-b-tree-for-new-hardware/.

[14] S. Levandoski, Lomet. The bw-tree: A b-tree on steroids, 2013. URL `http://www.hpts.ws/papers/2013/bw-tree-hpts2013.pdf`.

[15] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. ISBN 9780123742605.

[16] Z. Wang. index-microbench. `https://github.com/wangziqi2016/index-microbench`, 2018.

[17] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 473–488, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196895. URL `http://doi.acm.org/10.1145/3183713.3196895`.

Appendix A Memcached client code

main.hpp

```
#ifndef MEMCACHEDINTERFACE_MAIN_HPP
#define MEMCACHEDINTERFACE_MAIN_HPP

// These are YCSB workloads
enum {
    WORKLOAD_R,
    WORKLOAD_U,
    WORKLOAD_D,
    WORKLOAD_RU,
};

// These are key types we use for running the benchmark
enum {
    RAND_KEY,
    MONO_KEY,
    RDTSC_KEY,
    EMAIL_KEY,
};

void reset_mysql();
void *execute_init_load(void *thread_args);
void *execute_extra_load(void *thread_args);
void *execute_delete_load(void *thread_args);

#endif //MEMCACHEDINTERFACE_MAIN_HPP
```

main.cpp

```
#include <cstring>
#include <cstdlib>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <list>
#include <pthread.h>
#include <cmath>
#include "main.hpp"
#include <iostream>
```

```
#include <fstream>
#include <ctime>
#include <chrono>
#include <netinet/tcp.h>
#include <libmemcached/memcached.h>
#include <vector>
#include <mysql/mysql.h>
#include <unistd.h>
#include <cassert>

struct tuple_t {
    enum operation_t { INSERT, READ, UPDATE, DELETE };
    operation_t operation;
    std::string key;
    std::string value;

    tuple_t(const std::string &operation_arg, std::string key_arg, std::string value_arg)
    : key(std::move(key_arg)), value(std::move(value_arg))
    {
        if (operation_arg == "INSERT")
            operation = INSERT;
        else if (operation_arg == "READ")
            operation = READ;
        else if (operation_arg == "UPDATE")
            operation = UPDATE;
        else if (operation_arg == "DELETE")
            operation = DELETE;
        else
            assert(false);
    }

    tuple_t(const tuple_t &t) = default;
    tuple_t(tuple_t &&t) = default;
};


struct thread_arg_t{
    int start_i;
    int end_i;
};


struct res_t {
    int init_s;
    int init_f;

    float t1;
```

```
    res_t(int init_s_arg, int init_f_arg, float t1_arg)
    : init_s(init_s_arg), init_f(init_f_arg), t1(t1_arg)
    {}

    res_t(const res_t &t) = default;
    res_t(res_t &&r) = default;
};


struct res_extr_t{
    int insert_s;
    int insert_f;
    int read_s;
    int read_f;
    int update_s;
    int update_f;
    int delete_s;
    int delete_f;

    float t2;

    res_extr_t(int insert_s_arg, int insert_f_arg,
               int read_s_arg, int read_f_arg, int update_s_arg, int update_f_arg,
               int delete_s_arg, int delete_f_arg, float t2_arg)
            : insert_s(insert_s_arg), insert_f(insert_f_arg),
              read_s(read_s_arg), read_f(read_f_arg),
              update_s(update_s_arg), update_f(update_f_arg),
              delete_s(delete_s_arg), delete_f(delete_f_arg),
              t2(t2_arg)
    {}

    res_extr_t(const res_extr_t &t) = default;
    res_extr_t(res_extr_t &&r) = default;
};

static std::vector<tuple_t> tuples;

static std::vector<tuple_t> extr_tuples;

static std::vector<std::string> del_keys;

static std::vector<res_t> init_results;
static std::vector<res_extr_t> extr_results;
static std::vector<res_t> del_results;

//static const char *server_address = "frigg04.no.oracle.com";
```

```
//static const char *server_address = "10.172.139.128";
static const char *server_address = "loki08.no.oracle.com";
static const u_int16_t memcached_port = 11217;
static const u_int16_t mysqld_port = 13000;

bool init = true;
bool del = false;
bool insert_only = false;
bool extras_only = false;
bool inserted = false;
bool no_inserts = false;

void load_file(const std::string &path)
{
    if (insert_only && ((path.find("read") != std::string::npos) || (insert_only && path
        return;

    std::string line;
    std::string op;
    std::string key;

    std::ifstream file(path);

    auto value = (uint64_t)&tuples;
    auto extr_value = (uint64_t)&extr_tuples;

    if (file.is_open())
    {
        printf("Starting to load data.\n");
        while (file >> op >> key)
        {
            if (init){
                tuples.emplace_back(op, key, std::to_string(value++));
            } else if (del){
                del_keys.emplace_back(key);
            } else{
                extr_tuples.emplace_back(op, key, std::to_string(extr_value++));
            }
        }
        if (init){
            printf("Done loading init-data\n");
        }
        else if (del){
            printf("Done loading delete-data\n");
        }
        else {
            printf("Done loading extra-data\n");
```

```
        }

        file.close();
    } else {
        printf("Error, could not open file.\n");
        exit(1);
    }
    init = false;
}


void setup_mysql(bool retry)
{
    MYSQL *connection;
    connection = mysql_init(nullptr);

    if(!(mysql_real_connect(connection, server_address, "admin", "", "test", mysqld_port
    {
        fprintf(stderr, "\nError: %s [%d]\n", mysql_error(connection), mysql_errno(connec
        exit(1);
    }
    printf("Connected to server! Preparing server ...\n");


////////////////////////////////////////////////////////////////////////////////////////

    std::vector<int> mysql_feedback;
    ////////////////////////////////////////////////////////////////////////////////////

    // Query_01
    mysql_feedback.emplace_back(mysql_query(connection, "create database innodb_memcache"

    // Query_02
    mysql_feedback.emplace_back(mysql_query(connection, "use innodb_memcache"));

    // Query_03
    mysql_feedback.emplace_back(mysql_query(connection, "CREATE  TABLE IF NOT EXISTS `cac
                        "    `policy_name` VARCHAR(40) PRIMARY KEY,\n"
                        "    `get_policy` ENUM('innodb_only', 'cache_only', 'caching
                        "    NOT NULL ,\n"
                        "    `set_policy` ENUM('innodb_only', 'cache_only','caching'
                        "    NOT NULL ,\n"
                        "    `delete_policy` ENUM('innodb_only', 'cache_only', 'cachi
                        "    NOT NULL,\n"
                        "    `flush_policy` ENUM('innodb_only', 'cache_only', 'cachin
                        "    NOT NULL\n"
                        "    ) ENGINE = innodb"));
```

```
    // Query_04
    mysql_feedback.emplace_back(mysql_query(connection, "CREATE  TABLE IF NOT EXISTS ‘co
                                "    ‘name‘ varchar(50) not null primary key,\n"
                                "    ‘db_schema‘ VARCHAR(250) NOT NULL,\n"
                                "    ‘db_table‘ VARCHAR(250) NOT NULL,\n"
                                "    ‘key_columns‘ VARCHAR(250) NOT NULL,\n"
                                "    ‘value_columns‘ VARCHAR(250),\n"
                                "    ‘flags‘ VARCHAR(250) NOT NULL DEFAULT \"0\",\n"
                                "    ‘cas_column‘ VARCHAR(250),\n"
                                "    ‘expire_time_column‘ VARCHAR(250),\n"
                                "    ‘unique_idx_name_on_key‘ VARCHAR(250) NOT NULL\n"
                                "    ) ENGINE = InnoDB"));

    // Query_05
    mysql_feedback.emplace_back(mysql_query(connection, "CREATE  TABLE IF NOT EXISTS ‘co
                                "    ‘name‘ varchar(50) not null primary key,\n"
                                "    ‘value‘ varchar(50)) ENGINE = InnoDB"));

    ////////////////////////////////////////////////////////////////////////////////////
    // Query_06
    mysql_feedback.emplace_back(mysql_query(connection,
                R"(INSERT INTO cache_policies VALUES("cache_policy", "innodb_only", "inn

    // Query_07
    mysql_feedback.emplace_back(mysql_query(connection, R"(INSERT INTO config_options VA

    // Query_08
    mysql_feedback.emplace_back(mysql_query(connection, R"(INSERT INTO containers VALUES

    // Query_09
    mysql_feedback.emplace_back(mysql_query(connection, "USE test"));

    // Query_10
    mysql_feedback.emplace_back(mysql_query(connection, "CREATE TABLE t1 (c1 VARCHAR(32)

    // Query_11
    mysql_feedback.emplace_back(mysql_query(connection, "INSTALL PLUGIN daemon_memcached
////////////////////////////////////////////////////////////////////////////////////////

    int failure = 0;
    for (int i = 0; i < (int)mysql_feedback.size(); i++) {
        if (mysql_feedback[i] != 0){
            std::cout << "An error occurred during server setup on operation nr: " + std
            failure = 1;
            break;
        }
```

```
    }
    mysql_close(connection);

    if (failure){
        reset_mysql();
        if (!retry)
            setup_mysql(true);

        else if(retry) {
            printf("Can't connect. Aborting...\n");
            exit(1);
        }
    }
    usleep(1000000);
}


void reset_mysql()
{
    printf("Resetting...\n");
    MYSQL *connection;
    connection = mysql_init(nullptr);

    if(!(mysql_real_connect(connection, server_address, "admin", "", "test", mysqld_port
    {
        fprintf(stderr, "\nError: %s [%d]\n", mysql_error(connection), mysql_errno(connec
        return;
        //exit(1);
    }

    mysql_query(connection, "DROP TABLE t1");
    mysql_query(connection, "UNINSTALL PLUGIN daemon_memcached");
    mysql_query(connection, "DROP DATABASE innodb_memcache");
    mysql_close(connection);
}


static memcached_st* setup_memcached()
{
    std::string server_string;
    server_string.append("--SERVER=");
    server_string.append(server_address);
    server_string.append(":");
    server_string.append(std::to_string(memcached_port));
    memcached_st *server= memcached(server_string.c_str(), server_string.length());
    memcached_behavior_set(server, MEMCACHED_BEHAVIOR_RCV_TIMEOUT, 3000);
    memcached_behavior_set(server, MEMCACHED_BEHAVIOR_POLL_TIMEOUT, 3000);
```

```
    return server;
}


void choose_files(int wl, int kt, int ins, int ext)
{

    if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 10 && ext == 10){
        load_file("../../index-microbench-master/workloads/10mil/10mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/10mil/deleterand.dat");
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 10){
        load_file("../../index-microbench-master/workloads/10mil/10mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/10mil/randread.dat");
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 10){
        load_file("../../index-microbench-master/workloads/10mil/10mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/10mil/randupdate.dat");
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 10 && ext == 10){
        load_file("../../index-microbench-master/workloads/10mil/10mil/loadmono.dat");
        load_file("../../index-microbench-master/workloads/10mil/10mil/deletemono.dat");
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 10){
        load_file("../../index-microbench-master/workloads/10mil/10mil/loadmono.dat");
        load_file("../../index-microbench-master/workloads/10mil/10mil/monoread.dat");
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 10){
        load_file("../../index-microbench-master/workloads/10mil/10mil/loadmono.dat");
        load_file("../../index-microbench-master/workloads/10mil/10mil/monoupdate.dat");
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 50){
        load_file("../../index-microbench-master/workloads/10mil/50mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/50mil/randread.dat");
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 50){
        load_file("../../index-microbench-master/workloads/10mil/50mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/50mil/randupdate.dat");
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 50){
        load_file("../../index-microbench-master/workloads/10mil/50mil/loadmono.dat");
        load_file("../../index-microbench-master/workloads/10mil/50mil/monoread.dat");
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 50){
        load_file("../../index-microbench-master/workloads/10mil/50mil/loadmono.dat");
        load_file("../../index-microbench-master/workloads/10mil/50mil/monoupdate.dat");
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 100){
        load_file("../../index-microbench-master/workloads/10mil/100mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/100mil/randread.dat");
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 100){
        load_file("../../index-microbench-master/workloads/10mil/100mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/100mil/randupdate.dat")
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 100){
        load_file("../../index-microbench-master/workloads/10mil/100mil/loadmono.dat");
        load_file("../../index-microbench-master/workloads/10mil/100mil/monoread.dat");
```

```
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 100){
    load_file("../../index-microbench-master/workloads/10mil/100mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/10mil/100mil/monoupdate.dat")
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 200){
    load_file("../../index-microbench-master/workloads/10mil/200mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/10mil/200mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 200){
    load_file("../../index-microbench-master/workloads/10mil/200mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/10mil/200mil/randupdate.dat")
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 200){
    load_file("../../index-microbench-master/workloads/10mil/200mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/10mil/200mil/monoread.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 200){
    load_file("../../index-microbench-master/workloads/10mil/200mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/10mil/200mil/monoupdate.dat")


} else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 50 && ext == 10){
    load_file("../../index-microbench-master/workloads/50mil/10mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/10mil/deleterand.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 10){
    load_file("../../index-microbench-master/workloads/50mil/10mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/10mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 10){
    load_file("../../index-microbench-master/workloads/50mil/10mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/10mil/randupdate.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 50 && ext == 10){
    load_file("../../index-microbench-master/workloads/50mil/10mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/10mil/deletemono.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 10){
    load_file("../../index-microbench-master/workloads/50mil/10mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/10mil/monoread.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 10){
    load_file("../../index-microbench-master/workloads/50mil/10mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/10mil/monoupdate.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 50 && ext == 50){
    load_file("../../index-microbench-master/workloads/50mil/50mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/50mil/deleterand.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 50){
    load_file("../../index-microbench-master/workloads/50mil/50mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/50mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 50){
    load_file("../../index-microbench-master/workloads/50mil/50mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/50mil/randupdate.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 50 && ext == 50){
    load_file("../../index-microbench-master/workloads/50mil/50mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/50mil/deletemono.dat");
```

```
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 50){
    load_file("../../index-microbench-master/workloads/50mil/50mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/50mil/monoread.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 50){
    load_file("../../index-microbench-master/workloads/50mil/50mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/50mil/monoupdate.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 100){
    load_file("../../index-microbench-master/workloads/50mil/100mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/100mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 100){
    load_file("../../index-microbench-master/workloads/50mil/100mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/100mil/randupdate.dat")
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 100){
    load_file("../../index-microbench-master/workloads/50mil/100mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/100mil/monoread.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 100){
    load_file("../../index-microbench-master/workloads/50mil/100mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/100mil/monoupdate.dat")
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 200){
    load_file("../../index-microbench-master/workloads/50mil/200mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/200mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 200){
    load_file("../../index-microbench-master/workloads/50mil/200mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/50mil/200mil/randupdate.dat")
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 200){
    load_file("../../index-microbench-master/workloads/50mil/200mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/200mil/monoread.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 200){
    load_file("../../index-microbench-master/workloads/50mil/200mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/50mil/200mil/monoupdate.dat")


} else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 10){
    load_file("../../index-microbench-master/workloads/100mil/10mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/10mil/deleterand.dat")
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 10){
    load_file("../../index-microbench-master/workloads/100mil/10mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/10mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 10){
    load_file("../../index-microbench-master/workloads/100mil/10mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/10mil/randupdate.dat")
} else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 10){
    load_file("../../index-microbench-master/workloads/100mil/10mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/10mil/deletemono.dat")
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 10){
    load_file("../../index-microbench-master/workloads/100mil/10mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/10mil/monoread.dat");
```

```
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 10){
    load_file("../../index-microbench-master/workloads/100mil/10mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/10mil/monoupdate.dat")
} else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 50){
    load_file("../../index-microbench-master/workloads/100mil/50mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/50mil/deleterand.dat")
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 50){
    load_file("../../index-microbench-master/workloads/100mil/50mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/50mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 50){
    load_file("../../index-microbench-master/workloads/100mil/50mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/50mil/randupdate.dat")
} else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 50){
    load_file("../../index-microbench-master/workloads/100mil/50mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/50mil/deletemono.dat")
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 50){
    load_file("../../index-microbench-master/workloads/100mil/50mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/50mil/monoread.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 50){
    load_file("../../index-microbench-master/workloads/100mil/50mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/50mil/monoupdate.dat")
} else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 100){
    load_file("../../index-microbench-master/workloads/100mil/100mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/100mil/deleterand.dat"
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 100){
    load_file("../../index-microbench-master/workloads/100mil/100mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/100mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 100){
    load_file("../../index-microbench-master/workloads/100mil/100mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/100mil/randupdate.dat"
} else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 100){
    load_file("../../index-microbench-master/workloads/100mil/100mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/100mil/deletemono.dat"
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 100){
    load_file("../../index-microbench-master/workloads/100mil/100mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/100mil/monoread.dat");
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 100){
    load_file("../../index-microbench-master/workloads/100mil/100mil/loadmono.dat");
    load_file("../../index-microbench-master/workloads/100mil/100mil/monoupdate.dat"
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 200){
    load_file("../../index-microbench-master/workloads/100mil/200mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/200mil/randread.dat");
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 200){
    load_file("../../index-microbench-master/workloads/100mil/200mil/loadrand.dat");
    load_file("../../index-microbench-master/workloads/100mil/200mil/randupdate.dat"
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 200){
    load_file("../../index-microbench-master/workloads/100mil/200mil/loadmono.dat");
```

```
        load_file("../../index-microbench-master/workloads/100mil/200mil/monoread.dat");
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 200) {
        load_file("../../index-microbench-master/workloads/100mil/200mil/loadmono.dat");
        load_file("../../index-microbench-master/workloads/100mil/200mil/monoupdate.dat")
    } else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 10 && ext == 10) {
        load_file("../../index-microbench-master/workloads/10mil/10mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/10mil/read_update_rand.d
    } else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 50 && ext == 50) {
        load_file("../../index-microbench-master/workloads/50mil/50mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/50mil/50mil/read_update_rand.d
    } else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 100 && ext == 100) {
        load_file("../../index-microbench-master/workloads/100mil/100mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/100mil/100mil/read_update_rand
    } else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 10 && ext == 10) {
        load_file("../../index-microbench-master/workloads/10mil/10mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/10mil/10mil/read_update_mono.d
    } else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 50 && ext == 50) {
        load_file("../../index-microbench-master/workloads/50mil/50mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/50mil/50mil/read_update_mono.d
    } else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 100 && ext == 100) {
        load_file("../../index-microbench-master/workloads/100mil/100mil/loadrand.dat");
        load_file("../../index-microbench-master/workloads/100mil/100mil/read_update_mono
    }
}


int main(int argc, char *argv[])
{
    ////////////////////////////////////////////////////////////////////////////////////

    if (argc < 6) {
        std::cout << "Usage:\n";
        std::cout << "1. workload type: r, u, d,\n";
        std::cout << "2. key distribution: rand, mono\n";
        std::cout << "3. number of keys to insert in millions (10, 50, 100)\n";
        std::cout << "4. number of extra operations to execute in millions (10, 50, 100,
        std::cout << "5. number of threads (integer)\n";
        std::cout << "6 extra settings\n";
        std::cout << "   --hyper: Whether to pin all threads on NUMA node 0\n";
        std::cout << "   --mem: Whether to monitor memory access\n";
        std::cout << "   --numa: Whether to monitor NUMA throughput\n";
        std::cout << "   --insert-only: Whether to only execute insert operations\n";

        return 1;
    }

    int wl;
```

```
if (strcmp(argv[1], "r") == 0) {
    wl = WORKLOAD_R;
} else if (strcmp(argv[1], "u") == 0) {
    wl = WORKLOAD_U;
} else if (strcmp(argv[1], "d") == 0) {
    wl = WORKLOAD_D;
    del = true;
} else if (strcmp(argv[1], "ru") == 0) {
    wl = WORKLOAD_RU;
} else {
    fprintf(stderr, "Unknown workload: %s\n", argv[1]);
    exit(1);
}


// Then read key type
int kt;
if (strcmp(argv[2], "rand") == 0) {
    kt = RAND_KEY;
} else if (strcmp(argv[2], "mono") == 0) {
    kt = MONO_KEY;
} else if (strcmp(argv[2], "rdtsc") == 0) {
    kt = RDTSC_KEY;
} else {
    fprintf(stderr, "Unknown key type: %s\n", argv[2]);
    exit(1);
}


// Read amount insert ops in million
int insert_ops;
if (strcmp(argv[3], "10") == 0) {
    insert_ops = 10;
} else if (strcmp(argv[3], "50") == 0) {
    insert_ops = 50;
} else if (strcmp(argv[3], "100") == 0) {
    insert_ops = 100;
} else {
    fprintf(stderr, "Unknown amount of inserts: %s\n", argv[3]);
    exit(1);
}

// Read amount extra ops in million
int extra_ops;
if (strcmp(argv[4], "10") == 0) {
    extra_ops = 10;
} else if (strcmp(argv[4], "50") == 0) {
```

```
    extra_ops = 50;
} else if (strcmp(argv[4], "100") == 0) {
    extra_ops = 100;
} else if (strcmp(argv[4], "200") == 0) {
    extra_ops = 200;
} else {
    fprintf(stderr, "Unknown amount of extras: %s\n", argv[4]);
    exit(1);
}

size_t num_threads = atoi(argv[5]);

// Then read all remianing arguments
int repeat_counter = 1;
char **argv_end = argv + argc;
for(char **v = argv + 6;v != argv_end;v++) {
    if(strcmp(*v, "--insert-only") == 0) {
        insert_only = true;
    } else if(strcmp(*v, "--repeat") == 0) {
        repeat_counter = 5;
    } else if(strcmp(*v, "--extras-only") == 0) {
        extras_only = true;
    } else if(strcmp(*v, "--no-inserts") == 0) {
        no_inserts = true;
    } else {
            fprintf(stderr, "Unknown switch: %s\n", *v);
            exit(1);
    }

}

tuples.reserve(100000000);
if (del){
    del_keys.reserve(100000000);

    extr_tuples.reserve(0);
}
else {
    extr_tuples.reserve(200000000);
}

choose_files(wl, kt, insert_ops, extra_ops);

//printf("len1: %lu\n", tuples.size());
//printf("len1: %lu\n", extr_tuples.size());

/////////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
auto *threads = (u_long *) malloc(num_threads * sizeof(pthread_t));

if (num_threads > tuples.size() && !tuples.empty()) {
    num_threads = tuples.size();
} else if (num_threads > extr_tuples.size() && tuples.empty()){
    num_threads = extr_tuples.size();
}

if (num_threads == 0){
    printf("No threads remaining, aborting ...\n");
    exit(1);
}

int iteration = 0;

while (iteration < repeat_counter){

    auto start = std::chrono::high_resolution_clock::now();
    auto end = std::chrono::high_resolution_clock::now();

    void *ret[num_threads];
    unsigned int loc_res[12];

    for (unsigned int &loc_re : loc_res)
        loc_re = 0;

    if (!inserted){

        if (!no_inserts)
            setup_mysql(false);

        printf("Populating the index with %lu keys using %zu threads\n", tuples.size
        start = std::chrono::high_resolution_clock::now();
        int share_size = static_cast<int>(ceil((double)tuples.size() / (double)num_th

        for (size_t i = 0; i < num_threads; i++) {
            ////////////////////////////////////////////////////////////////////

            auto *thread_arg = new thread_arg_t;
            thread_arg->start_i = i * share_size;

            if (i != num_threads - 1)
                thread_arg->end_i = thread_arg->start_i + share_size;
            else
```

```
            thread_arg->end_i = tuples.size();

        /*
        for (tuple_t &tuple : tuples) {
            if (std::stoll(tuple.key) % num_threads == i)
                thread_arg->tuples.push_back(tuple);
        }

        printf("Thread %zu has %lu operations to insert.\n", i, thread_arg->tupl
        */
        printf("Thread %zu has %i operations to insert.\n", i, share_size);
        pthread_create(&threads[i], nullptr, execute_init_load, thread_arg);
        ////////////////////////////////////////////////////////////////////////
    }

    for (size_t i = 0; i < num_threads; i++) {
        pthread_join(threads[i], &ret[i]);
        auto *var = reinterpret_cast<unsigned int *>(ret[i]);
        loc_res[0] += var[0];
        loc_res[1] += var[1];
        delete[] (var);
    }

    end = std::chrono::high_resolution_clock::now();
    init_results.emplace_back(loc_res[0], loc_res[1], (std::chrono::duration<dou

    if (loc_res[0] + loc_res[1] != tuples.size() && !no_inserts) {
        printf("An error happened. The program handled %d records, when there ar
               tuples.size());
    }

    if (extras_only){
        inserted = true;
        for (int i = 0; i < repeat_counter - 1; i++)
            init_results.emplace_back(0, 0, 0);
    }

    if (insert_only){
        iteration++;
        printf("\nFinished iteration %d of %d\n\n", iteration, repeat_counter);
        reset_mysql();
        continue;
    }
}
////////////////////////////////////////////////////////////////////////////
////DELETE PART START////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

```
if (!del_keys.empty()){
    start = std::chrono::high_resolution_clock::now();

    printf("\nNumber of keys to delete: %lu\n", del_keys.size());
    int share_size = static_cast<int>(ceil((double)del_keys.size() / (double)num_

    for (size_t i = 0; i < num_threads; i++) {
        ////////////////////////////////////////////////////////////////////////

        auto *thread_arg = new thread_arg_t;

        thread_arg->start_i = i * share_size;

        if (i != num_threads - 1)
            thread_arg->end_i = thread_arg->start_i + share_size;
        else
            thread_arg->end_i = del_keys.size();

        /*
        for (std::string &del_key : del_keys) {
            if (std::stoll(del_key) % num_threads == i)
                thread_arg->tuples.emplace_back("DELETE", del_key, "");
        }
        printf("Thread %zu has %lu operations to delete.\n", i, thread_arg->tuple
        */
        printf("Thread %zu has %lu operations to delete.\n", i, del_keys.size())
        pthread_create(&threads[i], nullptr, execute_delete_load, thread_arg);
        ////////////////////////////////////////////////////////////////////////
    }

    for (size_t i = 0; i < num_threads; i++) {
        pthread_join(threads[i], &ret[i]);
        auto *var = reinterpret_cast<unsigned int *>(ret[i]);
        for (int j = 0; j < 2; j++)
            loc_res[j+2] += var[j];
        delete[] (var);
    }
    end = std::chrono::high_resolution_clock::now();
    del_results.emplace_back(loc_res[2], loc_res[3], (std::chrono::duration<doubl

    if (loc_res[2] + loc_res[3] != del_keys.size()) {
        printf("An error happened. The program handled %d delete records, when th
               del_keys.size());
    }
}
////////////////////////////////////////////////////////////////////////////////
```

```
////DELETE PART END//////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

start = std::chrono::high_resolution_clock::now();

if (extr_tuples.size() == 0){
    printf("No extra operations, should not be here, error in code. Aborting ...\
    exit(1);
}

printf("\nNumber of additional operations: %lu\n", extr_tuples.size());
int share_size = static_cast<int>(ceil((double)extr_tuples.size() / (double)num_t

for (size_t i = 0; i < num_threads; i++) {
    ////////////////////////////////////////////////////////////////////////////

    auto *thread_arg = new thread_arg_t;

    thread_arg->start_i = i * share_size;

    if (i != num_threads - 1)
        thread_arg->end_i = thread_arg->start_i + share_size;
    else
        thread_arg->end_i = extr_tuples.size();

    /*
    for (tuple_t &tuple : extr_tuples) {
        if (std::stoll(tuple.key) % num_threads == i)
            thread_arg->tuples.push_back(tuple);
    }
    printf("Thread %zu has %lu operations to work on.\n", i, thread_arg->tuples.s
    */
    printf("Thread %zu has %i operations to work on.\n", i, thread_arg->end_i - t
    pthread_create(&threads[i], nullptr, execute_extra_load, thread_arg);
    ////////////////////////////////////////////////////////////////////////////
}

for (size_t i = 0; i < num_threads; i++) {
    pthread_join(threads[i], &ret[i]);
    auto *var = reinterpret_cast<unsigned int *>(ret[i]);
    for (int j = 0; j < 8; j++)
        loc_res[j+4] += var[j];
    delete[] (var);
}
end = std::chrono::high_resolution_clock::now();
extr_results.emplace_back(loc_res[4], loc_res[5], loc_res[6], loc_res[7], loc_res
```

```
        if (loc_res[4] + loc_res[5] + loc_res[6] + loc_res[7] + loc_res[8] + loc_res[9] +
            printf("An error happened. The program handled %d extra records, when there a
                   extr_tuples.size());
        }

        iteration++;
        printf("\nFinished iteration %d of %d\n\n", (iteration), repeat_counter);
        if (!extras_only)
            reset_mysql();
    }

    free(threads);

    for (int i = 0; i < repeat_counter; i++){
        if (!insert_only && del_keys.empty()){
            printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec - part 2: INS_S
                   (i+1), init_results[i].init_s, init_results[i].init_f, init_results[i]
                   extr_results[i].insert_s, extr_results[i].insert_f, extr_results[i].re
        } else if (!del_keys.empty()){
            printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec - part 2: DEL_S
                   (i+1), init_results[i].init_s, init_results[i].init_f, init_results[i]
        } /*else if (extras_only){
            printf("Iter%d part 2: INS_S: %d | INS_F: %d | READ_S: %d | READ_F: %d | UPD_
                   (i+1), extr_results[i].insert_s, extr_results[i].insert_f, extr_result
        } */else {
            printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec\n",
                   (i+1), init_results[i].init_s, init_results[i].init_f, init_results[i]
        }
    }
}


inline memcached_return_t send_query(memcached_st *memcached, tuple_t::operation_t op, co
{
    if (op == tuple_t::INSERT){
        memcached_return_t result = memcached_add(memcached, key.c_str(), key.length(), v
        if (result == MEMCACHED_SUCCESS)
            return MEMCACHED_STORED;
        return result;
    }
    else if (op == tuple_t::READ){
        size_t ret_val_len;
        uint32_t flags;
        memcached_return_t ret;
        const char* res = memcached_get(memcached, key.c_str(), key.length(), &ret_val_le

        if (res != nullptr){
```

```
            delete res;
            return MEMCACHED_SUCCESS;
        }
        delete res;
        return MEMCACHED_READ_FAILURE;
    }
    else if (op == tuple_t::UPDATE){
        memcached_return_t result = memcached_replace(memcached, key.c_str(), key.length
        if (result == MEMCACHED_SUCCESS)
            return MEMCACHED_DATA_EXISTS;
        else if (result == MEMCACHED_NOTSTORED)
            return MEMCACHED_DATA_DOES_NOT_EXIST;
        return result;
    }
    else if (op == tuple_t::DELETE){
        memcached_return_t result = memcached_delete(memcached, key.c_str(), key.length(
        if (result == MEMCACHED_SUCCESS || result == MEMCACHED_TIMEOUT)
            return MEMCACHED_DELETED;
        else if (result != MEMCACHED_FAILURE)
            return result;
        return MEMCACHED_NOTFOUND;
    }
    else {
        printf("Op %d failed.\n", op);
        return MEMCACHED_NOT_SUPPORTED;
    }
}


void* execute_init_load(void *thread_args)
{
    const int options = 2;
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++) {
        ret[i] = 0;
    }

    int res[60];
    for (int &re : res)
        re = 0;

    int ops_counter = 0;
    auto *thread_arg = (struct thread_arg_t*)thread_args;
    memcached_st* memcached = setup_memcached();

    auto s = std::chrono::high_resolution_clock::now();
```

```
    if (!no_inserts){

        for (int i = thread_arg->start_i; i < thread_arg->end_i; i++){
            if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
                s = std::chrono::high_resolution_clock::now();

            memcached_return_t result = send_query(memcached, tuples[i].operation, tuples
            ///Handling server feedback////////////////////////////////////////////////
            if (result == MEMCACHED_STORED){ //insert success
                ret[0]++;
            }
            else if (result == MEMCACHED_NOTSTORED){ //insert failed
                if (ops_counter % 400000 == 0)
                    std::cout << "Operation " << tuples[i].operation << " on key-value-pa
                ret[1]++;
            } else {
                //std::cerr << "Something went wrong, error code \"" << memcached_strerro
                res[result]++;
            }
            ///////////////////////////////////////////////////////////////////////////

            if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
                printf("Insert operation took: %.9f sec\n", (std::chrono::duration<double

            ops_counter++;
            if(ops_counter % 500000 == 0){
                printf("Thread handled %.2f%% of its insert-operations, %d of %i.\n", ((
            }
        }
    }

    memcached_flush_buffers(memcached);
    memcached_free(memcached);
    delete(thread_arg);

    return (void*) ret;
}


void* execute_extra_load(void *thread_args)
{
    const int options = 4*2; //4 operation types * 2 (success/failure)
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++)
        ret[i] = 0;

    int res[60];
```

```
for (int &re : res)
    re = 0;

int ops_counter = 0;
auto *thread_arg = (struct thread_arg_t*)thread_args;
memcached_st* memcached = setup_memcached();

auto s = std::chrono::high_resolution_clock::now();

for (int i = thread_arg->start_i; i < thread_arg->end_i; i++){
    if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
        s = std::chrono::high_resolution_clock::now();

    memcached_return_t result = send_query(memcached, extr_tuples[i].operation, extr_

    ///Handling server feedback/////////////////////////////////////////////////////

    if (result == MEMCACHED_STORED) //insert success
        ret[0]++;
    else if (result == MEMCACHED_SUCCESS) //read success
        ret[2]++;
    else if (result == MEMCACHED_DATA_EXISTS) //update success
        ret[4]++;
    else if (result == MEMCACHED_DELETED) //delete success
        ret [6]++;

    else if (result == MEMCACHED_NOTSTORED){ //insert failed
        //std::cout << "Operation " + extr_ops[i] + " on key-value-pair: " + extr_tup
        ret[1]++;
    } else if (result == MEMCACHED_READ_FAILURE){ //read failed
        //std::cout << "Operation " + extr_ops[i] + " on key : " + extr_tuples[i].key
        ret[3]++;
    } else if (result == MEMCACHED_DATA_DOES_NOT_EXIST){ //update failed
        //std::cout << "Operation " + extr_ops[i] + " on key : " + extr_tuples[i].key
        ret[5]++;
    } else if (result == MEMCACHED_NOTFOUND){ //delete failed
        //std::cout << "Operation " + extr_ops[i] + " on key : " + extr_tuples[i].key
        ret[7]++;
    } else {
        //std::cerr << "Something went wrong, error code \"" << memcached_strerror(me
        if (tuples[i].operation == tuple_t::INSERT){
            ret[1]++;
            res[result]++;
        }
        else if (tuples[i].operation == tuple_t::READ){
            ret[3]++;
            res[result]++;
```

```
            }
            else if (tuples[i].operation == tuple_t::UPDATE){
                ret[5]++;
                res[result]++;
            }
            else if (tuples[i].operation == tuple_t::DELETE){
                ret[7]++;
                res[result]++;
            }
        }
        //////////////////////////////////////////////////////////////////////////////

        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            printf("Update operation took: %.9f sec\n", (std::chrono::duration<double, st
        ops_counter++;
        if(ops_counter % 500000 == 0){
            printf("Thread handled %.2f%% of its extra-operations, %d of %i.\n", ((float
        }
    }

    memcached_flush_buffers(memcached);
    memcached_free(memcached);
    delete(thread_arg);

    return (void*) ret;
}


void* execute_delete_load(void *thread_args)
{
    const int options = 2;
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++)
        ret[i] = 0;

    int res[60];
    for (int &re : res)
        re = 0;

    int ops_counter = 0;
    auto *thread_arg = (struct thread_arg_t*)thread_args;
    memcached_st* memcached = setup_memcached();

    for (int i = thread_arg->start_i; i < thread_arg->end_i; i++){
        memcached_return_t result = send_query(memcached, tuple_t::DELETE, del_keys[i], "
        ///Handling server feedback/////////////////////////////////////////////////////
        if (result == MEMCACHED_DELETED) {//delete success
```

```
            ret [0]++;
        } else if (result == MEMCACHED_NOTFOUND){ //delete failed
            //std::cout << "Operation " << tuple.operation << " on key : " + tuple.key +
            ret[1]++;
        } else {
            //std::cerr << "Something went wrong, error code \"" << memcached_strerror(me
            res[result]++;
            ret[1]++;
        }
        ////////////////////////////////////////////////////////////////////////////////

        ops_counter++;
        if(ops_counter % 500000 == 0){
            printf("Thread handled %.2f%% of its delete-operations, %d of %i.\n", ((floa
        }
    }

    for (int i = 0; i < 60; i++){
        if (res[i] != 0)
            printf("%d was %d\n", i, res[i]);
    }


    memcached_flush_buffers(memcached);
    memcached_free(memcached);
    delete(thread_arg);

    return (void*) ret;
}
```

Appendix B OpenBw-tree client side code

clientSide.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <inttypes.h>
#include <cmath>
#include <vector>
#include <time.h>
#include <chrono>

#include "./pcm/pcm-memory.cpp"
#include "./pcm/pcm-numa.cpp"
#include "./papi_util.cpp"

#include "microbench.h"

#include <cstring>
#include <cctype>
#include <atomic>

thread_local long skiplist_steps = 0;
std::atomic<long> skiplist_total_steps;

struct tuple_t {
    std::string key;
    std::string value;

    tuple_t(std::string key_arg, std::string value_arg)
            : key(std::move(key_arg)), value(std::move(value_arg))
    {}

    tuple_t(const tuple_t &t) = default;
    tuple_t(tuple_t &&t) = default;
};
```

```
struct res_t {
    int init_s;
    int init_f;
    float t1;

    res_t(int init_s_arg, int init_f_arg, float t1_arg)
            : init_s(init_s_arg), init_f(init_f_arg), t1(t1_arg)
    {}

    res_t(const res_t &t) = default;
    res_t(res_t &&r) = default;
};


struct res_extr_t{
    int insert_s;
    int insert_f;
    int read_s;
    int read_f;
    int update_s;
    int update_f;
    int delete_s;
    int delete_f;

    float t2;

    res_extr_t(int insert_s_arg, int insert_f_arg,
               int read_s_arg, int read_f_arg, int update_s_arg, int update_f_arg,
               int delete_s_arg, int delete_f_arg, float t2_arg)
            : insert_s(insert_s_arg), insert_f(insert_f_arg),
              read_s(read_s_arg), read_f(read_f_arg),
              update_s(update_s_arg), update_f(update_f_arg),
              delete_s(delete_s_arg), delete_f(delete_f_arg),
              t2(t2_arg)
    {}

    res_extr_t(const res_extr_t &t) = default;
    res_extr_t(res_extr_t &&r) = default;
};


struct thread_arg_t{
    size_t start_i;
    size_t end_i;
    int tr_idx;
};
```

```
static std::vector<tuple_t> tuples;
static std::vector<tuple_t> extr_tuples;
static std::vector<tuple_t> delete_tuples;

int iteration = 0;

static std::vector<res_t> init_results;
static std::vector<res_extr_t> extr_results;
static std::vector<res_t> del_results;

static std::vector<int> _sockets;

static char *server_address = "loki08.no.oracle.com";
//static char *server_address = "10.172.139.128";
static const u_int16_t server_port = 11217;

//#define USE_TBB

#ifdef USE_TBB
#include "tbb/tbb.h"
#endif

// Enable this if you need pre-allocation utilization
//#define BWTREE_CONSOLIDATE_AFTER_INSERT

#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
#ifdef USE_TBB
  #warning "Could not use TBB and BwTree consolidate together"
  #endif
#endif

#ifdef BWTREE_COLLECT_STATISTICS
#ifdef USE_TBB
#warning "Could not use TBB and BwTree statistics together"
#endif
#endif

// Whether insert interleaves
//#define INTERLEAVED_INSERT

// Whether read operatoin miss will be counted
//#define COUNT_READ_MISS

typedef uint64_t keytype;
typedef std::less<uint64_t> keycomp;

static const uint64_t key_type=0;
```

```
static const uint64_t value_type=1; // 0 = random pointers, 1 = pointers to keys

extern bool hyperthreading;

// This is the flag for whather to measure memory bandwidth
static bool memory_bandwidth = false;
// Whether to measure NUMA Throughput
static bool numa = false;
// Whether we only perform insert
static bool insert_only = false;

static bool del = false;

// We could set an upper bound of the number of loaded keys
static int64_t max_init_key = -1;

#include "util.h"

/*
 * MemUsage() - Reads memory usage from /proc file system
 */
size_t MemUsage() {
    FILE *fp = fopen("/proc/self/statm", "r");
    if(fp == nullptr) {
        fprintf(stderr, "Could not open /proc/self/statm to read memory usage\n");
        exit(1);
    }

    unsigned long unused;
    unsigned long rss;
    if (fscanf(fp, "%ld %ld %ld %ld %ld %ld %ld", &unused, &rss, &unused, &unused, &unuse
        perror("");
        exit(1);
    }

    (void)unused;
    fclose(fp);

    return rss * (4096 / 1024); // in KiB (not kB)
}
/////////////////////////////////////////////////////////////////////////////////////////
int setup_connection(char* server_address, u_int16_t port)
{
    // Try to create a socket
    int _socket = socket(AF_INET, SOCK_STREAM, 0);
    if (_socket < 0) {
        perror("Cannot create a socket"); exit(1);
```

```
    }

    // Fill in the address of server
    struct sockaddr_in server;
    memset(&server, 0, sizeof(server));

    char* server_host = "localhost";
    if (server_address != nullptr) server_host = server_address;


    // Resolve the server address (convert from symbolic name to IP number)
    struct hostent *host = gethostbyname(server_host);
    if (host == NULL) {
        perror("Cannot define host address"); exit(1);
    }
    server.sin_family = AF_INET;
    u_int16_t server_port = port;
    server.sin_port = htons(server_port);

    // Write resolved IP address of a server to the address structure
    memmove(&(server.sin_addr.s_addr), host->h_addr_list[0], 4);

    // Connect to a remote server
    int res = connect(_socket, (struct sockaddr*) &server, sizeof(server));
    if (res < 0) {
        perror("Cannot connect"); exit(1);
    }
    //printf("Connected, ready to send data.\n");

    return _socket;
}


int send_query(int _socket, const std::string &op_key, const std::string &val)
{
    char buffer[512];
    long res = write(_socket, (op_key+" "+val+"\0").c_str(), ((op_key.length())+(val.leng
    if (res < 0){
        //std::cout << "An error occurred sending info to the server, shutting down. Err
        //exit(1);
        return -1;
    }
    long recv = read(_socket, buffer, 512);
    if (recv < 0) {
        //std::cout << "An error occurred receiving info from the server, shutting down.
        //exit(1);
        return -1;
```

```
    }
    return atoi(buffer);
}


//================================================================
// PREPARE
//================================================================
void send_info(int info, int _socket)
{
    long res = write(_socket, (to_string(info)+"\0").c_str(), (size_t) to_string(info).le
    if (res < 0){
        std::cout << "An error occurred sending info to the server, shutting down. Error
        exit(1);
    }
    char buffer[512];
    long recv = read(_socket, buffer, 512);
    if (recv < 0) {
        std::cout << "An error occurred receiving info to the server, shutting down. Erro
        exit(1);
    }
}
void prepare_server(int wl, int index_type, int num_thread, int repeat_counter)
{
    printf("\nWant to send: %d %d %d %d %d %d %d %d %d\n", wl, index_type, num_thread, in

    int _socket = setup_connection(server_address, 11211);

    send_info(wl, _socket);
    send_info(index_type, _socket);
    send_info(num_thread, _socket);
    send_info(insert_only, _socket);
    send_info(numa, _socket);
    send_info(hyperthreading, _socket);
    send_info(memory_bandwidth, _socket);
    send_info(repeat_counter, _socket);
    send_info(del, _socket);

    for (int i = 0; i < num_thread; i++)
        _sockets.emplace_back(setup_connection(server_address, 11211));

    close(_socket);
}


//================================================================
// EXECUTE
```

```
//==================================================================
void* execute_init_load(void *thread_args)
{
    const int options = 2;
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++) {
        ret[i] = 0;
    }

    auto *thread_arg = (struct thread_arg_t*)thread_args;

    int share_size = thread_arg->end_i - thread_arg->start_i;
    send_info(share_size, _sockets[thread_arg->tr_idx]);

    auto s = std::chrono::high_resolution_clock::now();

    for (size_t i = thread_arg->start_i; i < thread_arg->end_i; i++){
        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            s = std::chrono::high_resolution_clock::now();

        int res = send_query(_sockets[thread_arg->tr_idx], tuples[i].key, tuples[i].value
        ///Handling server feedback////////////////////////////////////////////////////

        if (res == BWTREE_STORED){
            ret[0]++;
        }
        else if (res == BWTREE_NOTSTORED){
            std::cout << "Op-key " + tuples[i].key + " with value: " + tuples[i].value +
            ret[1]++;
        }
        else {
            //std::cout << "An unknown error happened during initial insertion, exiting.
            //exit(1);
            ret[1]++;
        }
        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            printf("Update operation took: %.9f sec\n", (std::chrono::duration<double, st
        ////////////////////////////////////////////////////////////////////////////////
    }
    delete(thread_arg);

    return (void*) ret;
}


void* execute_delete_load(void *thread_args)
{
```

```
    const int options = 2;
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++)
        ret[i] = 0;

    auto *thread_arg = (struct thread_arg_t*) thread_args;

    int share_size = thread_arg->end_i - thread_arg->start_i;
    send_info(share_size, _sockets[thread_arg->tr_idx]);

    for (size_t i = thread_arg->start_i; i < thread_arg->end_i; i++) {
        int res = send_query(_sockets[thread_arg->tr_idx], delete_tuples[i].key, delete_t
        ///Handling server feedback////////////////////////////////////////////////////

        if (res == BWTREE_DELETED) {
            ret[0]++;
        } else if (res == BWTREE_NOTDELETED){
            std::cout << "Operation " + delete_tuples[i].key + " on key : " + delete_tupl
            ret[1]++;
        } else {
            std::cout << "An unknown error happened during deletions, exiting...";
            exit(1);
        }
        ///////////////////////////////////////////////////////////////////////////////

    }
    delete (thread_arg);

    return (void *) ret;
}


void* execute_extra_load(void *thread_args)
{
    const int options = 4*2; //4 operation types * 2 (success/failure)
    auto *ret = new unsigned int[options];
    for (int i = 0; i < options; i++)
        ret[i] = 0;

    auto *thread_arg = (struct thread_arg_t *) thread_args;

    int share_size = thread_arg->end_i - thread_arg->start_i;
    send_info(share_size, _sockets[thread_arg->tr_idx]);

    auto s = std::chrono::high_resolution_clock::now();

    int fail_counter = 0;
```

```
    for (size_t i = thread_arg->start_i; i < thread_arg->end_i; i++) {
        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            s = std::chrono::high_resolution_clock::now();

        int res = send_query(_sockets[thread_arg->tr_idx], extr_tuples[i].key, extr_tupl
        ///Handling server feedback/////////////////////////////////////////////////////
        if (res == BWTREE_STORED) //insert success
            ret[0]++;
        else if (res == BWTREE_READ) //read success
            ret[2]++;
        else if (res == BWTREE_UPDATED) //update success
            ret[4]++;
        else if (res == BWTREE_DELETED) //delete success
            ret [6]++;

        else if (res == BWTREE_NOTSTORED){ //insert failed
            std::cout << "Op-key " + extr_tuples[i].key + " with value: " + extr_tuples[
            ret[1]++;
        } else if (res == BWTREE_NOTREAD){ //read failed
            std::cout << "Op-key: " + extr_tuples[i].key + " was unsuccessful, key does 
            ret[3]++;
        } else if (res == BWTREE_NOTUPDATED){ //update failed
            std::cout << "Op-key: " + extr_tuples[i].key + " with value " + extr_tuples[
            ret[5]++;
        } else if (res == BWTREE_NOTDELETED){ //delete failed
            std::cout << "Op-key: " + extr_tuples[i].key + " was unsuccessful, key does 
            ret[7]++;
        } else {
            //std::cout << "Something went wrong; error code: " + to_string(res) + "; Wa
            //exit(1);
            fail_counter++;
        }
        if (fail_counter >= 10){
            printf("To many errors, abort iteration ... \n");
            break;
        }

        if (i == thread_arg->start_i || i+1 == thread_arg->end_i)
            printf("Update operation took: %.9f sec\n", (std::chrono::duration<double, s
        ////////////////////////////////////////////////////////////////////////////////
    }
    delete (thread_arg);

    return (void *) ret;
}
```

```
////////////////////////////////////////////////////////////////////////////////

//==============================================================
// LOAD
//==============================================================
inline void load(int wl, int kt, int index_type, int ins, int ext)
{

    std::string init_file;
    std::string txn_file;
    std::string delete_file;

    if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/10mil/10mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/10mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/10mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadmono.dat";
        delete_file = "../index-microbench-master/workloads/10mil/10mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/10mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 10){
        init_file = "../index-microbench-master/workloads/10mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/10mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 50){
        init_file = "../index-microbench-master/workloads/10mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/50mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 50){
        init_file = "../index-microbench-master/workloads/10mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/50mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 50){
        init_file = "../index-microbench-master/workloads/10mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/50mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 50){
        init_file = "../index-microbench-master/workloads/10mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/50mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 100){
        init_file = "../index-microbench-master/workloads/10mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/100mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 100){
        init_file = "../index-microbench-master/workloads/10mil/100mil/loadrand.dat";
```

```
        txn_file = "../index-microbench-master/workloads/10mil/100mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 100){
        init_file = "../index-microbench-master/workloads/10mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/100mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 100){
        init_file = "../index-microbench-master/workloads/10mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/100mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 10 && ext == 200){
        init_file = "../index-microbench-master/workloads/10mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/200mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 10 && ext == 200){
        init_file = "../index-microbench-master/workloads/10mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/10mil/200mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 10 && ext == 200){
        init_file = "../index-microbench-master/workloads/10mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/200mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 10 && ext == 200){
        init_file = "../index-microbench-master/workloads/10mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/10mil/200mil/monoupdate.dat";


    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/50mil/10mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/10mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/10mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadmono.dat";
        delete_file = "../index-microbench-master/workloads/50mil/10mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/10mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 10){
        init_file = "../index-microbench-master/workloads/50mil/10mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/10mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/50mil/50mil/deleterand.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/50mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadrand.dat";
```

```
        txn_file = "../index-microbench-master/workloads/50mil/50mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadmono.dat";
        delete_file = "../index-microbench-master/workloads/50mil/50mil/deletemono.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/50mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 50){
        init_file = "../index-microbench-master/workloads/50mil/50mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/50mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 100){
        init_file = "../index-microbench-master/workloads/50mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/100mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 100){
        init_file = "../index-microbench-master/workloads/50mil/100mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/100mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 100){
        init_file = "../index-microbench-master/workloads/50mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/100mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 100){
        init_file = "../index-microbench-master/workloads/50mil/100mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/100mil/monoupdate.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 50 && ext == 200){
        init_file = "../index-microbench-master/workloads/50mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/200mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 50 && ext == 200){
        init_file = "../index-microbench-master/workloads/50mil/200mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/50mil/200mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 50 && ext == 200){
        init_file = "../index-microbench-master/workloads/50mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/200mil/monoread.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 50 && ext == 200){
        init_file = "../index-microbench-master/workloads/50mil/200mil/loadmono.dat";
        txn_file = "../index-microbench-master/workloads/50mil/200mil/monoupdate.dat";


    } else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadrand.dat";
        delete_file = "../index-microbench-master/workloads/100mil/10mil/deleterand.dat"
    } else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/10mil/randread.dat";
    } else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadrand.dat";
        txn_file = "../index-microbench-master/workloads/100mil/10mil/randupdate.dat";
    } else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 10){
        init_file = "../index-microbench-master/workloads/100mil/10mil/loadmono.dat";
```

```
    delete_file = "../index-microbench-master/workloads/100mil/10mil/deletemono.dat"
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 10){
    init_file = "../index-microbench-master/workloads/100mil/10mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/10mil/monoread.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 10){
    init_file = "../index-microbench-master/workloads/100mil/10mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/10mil/monoupdate.dat";
} else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 50){
    init_file = "../index-microbench-master/workloads/100mil/50mil/loadrand.dat";
    delete_file = "../index-microbench-master/workloads/100mil/50mil/deleterand.dat"
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 50){
    init_file = "../index-microbench-master/workloads/100mil/50mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/100mil/50mil/randread.dat";
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 50){
    init_file = "../index-microbench-master/workloads/100mil/50mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/100mil/50mil/randupdate.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 50){
    init_file = "../index-microbench-master/workloads/100mil/50mil/loadmono.dat";
    delete_file = "../index-microbench-master/workloads/100mil/50mil/deletemono.dat"
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 50){
    init_file = "../index-microbench-master/workloads/100mil/50mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/50mil/monoread.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 50){
    init_file = "../index-microbench-master/workloads/100mil/50mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/50mil/monoupdate.dat";
} else if (kt == RAND_KEY && wl == WORKLOAD_D && ins == 100 && ext == 100){
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadrand.dat";
    delete_file = "../index-microbench-master/workloads/100mil/100mil/deleterand.dat'
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 100){
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/100mil/100mil/randread.dat";
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 100){
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/100mil/100mil/randupdate.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_D && ins == 100 && ext == 100){
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadmono.dat";
    delete_file = "../index-microbench-master/workloads/100mil/100mil/deletemono.dat'
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 100){
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/100mil/monoread.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 100){
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/100mil/monoupdate.dat";
} else if (kt == RAND_KEY && wl == WORKLOAD_R && ins == 100 && ext == 200){
    init_file = "../index-microbench-master/workloads/100mil/200mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/100mil/200mil/randread.dat";
} else if (kt == RAND_KEY && wl == WORKLOAD_U && ins == 100 && ext == 200){
```

```
    init_file = "../index-microbench-master/workloads/100mil/200mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/100mil/200mil/randupdate.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_R && ins == 100 && ext == 200){
    init_file = "../index-microbench-master/workloads/100mil/200mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/200mil/monoread.dat";
} else if (kt == MONO_KEY && wl == WORKLOAD_U && ins == 100 && ext == 200) {
    init_file = "../index-microbench-master/workloads/100mil/200mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/200mil/monoupdate.dat";
}

else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 10 && ext == 10) {
    init_file = "../index-microbench-master/workloads/10mil/10mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/10mil/10mil/read_update_rand.da
} else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 50 && ext == 50) {
    init_file = "../index-microbench-master/workloads/50mil/50mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/50mil/50mil/read_update_rand.da
} else if (kt == RAND_KEY && wl == WORKLOAD_RU && ins == 100 && ext == 100) {
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadrand.dat";
    txn_file = "../index-microbench-master/workloads/100mil/100mil/read_update_rand.
} else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 10 && ext == 10) {
    init_file = "../index-microbench-master/workloads/10mil/10mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/10mil/10mil/read_update_mono.da
} else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 50 && ext == 50) {
    init_file = "../index-microbench-master/workloads/50mil/50mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/50mil/50mil/read_update_mono.da
} else if (kt == MONO_KEY && wl == WORKLOAD_RU && ins == 100 && ext == 100) {
    init_file = "../index-microbench-master/workloads/100mil/100mil/loadmono.dat";
    txn_file = "../index-microbench-master/workloads/100mil/100mil/read_update_mono.
} else {
    printf("Unknown file, aborting ...\n");
    exit(1);
}

std::ifstream infile_load(init_file);

std::string op;
std::string key;
int range;
std::string line;

std::string insert("INSERT");
std::string read("READ");
std::string update("UPDATE");
std::string scan("SCAN");
std::string remove("DELETE");

auto value = (uint64_t)&tuples;
```

```
printf("Loading init-file...\n");
while (infile_load >> op >> key) {

    if (op.compare(insert) != 0) {
        std::cout << "READING LOAD FILE FAIL!\n";
        return;
    }
    tuples.emplace_back(key, to_string(value++));

}
infile_load.close();

// If we do not perform other transactions, we can skip txn file
if(insert_only) {
    return;
}

if(!delete_file.empty()){
    printf("Loading delete-file...\n");
    std::ifstream infile_del(delete_file);

    while (infile_del >> op >> key) {
        if (op.compare(remove) == 0)
            delete_tuples.emplace_back((to_string(OP_REMOVE)+" "+key), ""); //Using "
        else {
            std::cout << "UNRECOGNIZED CMD!\n";
            infile_del.close();
            return;
        }
    }
    infile_del.close();
}

// If we also execute transaction then open the
// transacton file here
if(!txn_file.empty()){
    printf("Loading extras-file...\n");
    std::ifstream infile_txn(txn_file);
    auto extr_value = (uint64_t)&extr_tuples;

    while (infile_txn >> op >> key) {
        if (op.compare(insert) == 0)
            extr_tuples.emplace_back((to_string(OP_INSERT)+" "+key), to_string(extr_v
        else if (op.compare(read) == 0)
            extr_tuples.emplace_back((to_string(OP_READ)+" "+key), "0");
        else if (op.compare(update) == 0)
            extr_tuples.emplace_back((to_string(OP_UPSERT)+" "+key), to_string(extr_v
```

```
            else if (op.compare(remove) == 0)
                extr_tuples.emplace_back((to_string(OP_REMOVE)+" "+key), ""); //Using ""
            else {
                std::cout << "UNRECOGNIZED CMD!\n";
                infile_txn.close();
                return;
            }
        }

        infile_txn.close();
    }
    //printf("\n\nLength 1: %lu, length 2: %lu\n\n", tuples.size(), extr_tuples.size());
    return;
}

//==============================================================
// EXEC
//==============================================================
inline void exec(int num_thread)
{

    //WRITE ONLY TEST-----------------
    int count = (int)tuples.size();
    fprintf(stderr, "Populating the index with %d keys using %d threads\n", count, num_th

    auto start = std::chrono::high_resolution_clock::now();

    void *ret[num_thread];
    unsigned int loc_res[12];
    for (unsigned int &loc_re : loc_res)
        loc_re = 0;

    int share_size = static_cast<int>(ceil((double)tuples.size() / (double)num_thread));
    auto *threads = (u_long*) malloc(num_thread * sizeof(pthread_t));

    for (int i = 0; i < num_thread; i++)
    {
        auto *thread_arg = new thread_arg_t;

        thread_arg->tr_idx = i;

        thread_arg->start_i = i * share_size;

        if (i != num_thread -1)
        {
            thread_arg->end_i = thread_arg->start_i + share_size;
        } else {
```

```
            thread_arg->end_i = (int)tuples.size();
        }

        pthread_create(&threads[i], NULL, execute_init_load, thread_arg);
    }

    for (size_t i = 0; i < num_thread; i++){
        pthread_join(threads[i], &ret[i]);
        auto *var = reinterpret_cast<unsigned int *>(ret[i]);
        loc_res[0] += var[0];
        loc_res[1] += var[1];
    }


    auto end = std::chrono::high_resolution_clock::now();
    init_results.emplace_back(loc_res[0], loc_res[1], (std::chrono::duration<double, std

    printf("Inserts iterasjon it: %d | successes: %d | failures: %d | time: %.2f sec\n",


    //printf("\nExecuting input file 1 took %f seconds.\n", (std::chrono::duration<double
    //printf("Succeeded operations: %u, Failed operations: %u\n\n", all_res[iteration][0
    if (loc_res[0] + loc_res[1] != tuples.size()) {
        printf("An error happened. The program handled %d records, when there are suppose
               tuples.size());
    }

    if (insert_only){
        free(threads);
        return;
    }

    ////////////////////////////////////////////////////////////////////////////////////
    ////DELETE PART START///////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////

    if (del){
        start = std::chrono::high_resolution_clock::now();
        share_size = static_cast<int>(ceil((double)delete_tuples.size() / (double)num_thr

        printf("\nNumber of keys to delete: %lu\n", delete_tuples.size());

        for (int i = 0; i < num_thread; i++)
        {
            auto *thread_arg = new thread_arg_t;

            thread_arg->tr_idx = i;
```

```
        thread_arg->start_i = i * share_size;

        if (i != num_thread -1)
        {
            thread_arg->end_i = thread_arg->start_i + share_size;
        } else {
            thread_arg->end_i = (int)delete_tuples.size();
        }

        pthread_create(&threads[i], NULL, execute_delete_load, thread_arg);
    }

    for (size_t i = 0; i < num_thread; i++){
        pthread_join(threads[i], &ret[i]);
        auto *var = reinterpret_cast<unsigned int *>(ret[i]);
        //printf("threadnr: %zu, var0: %u, var1: %u\n", i, var[0], var[1]);
        loc_res[2] += var[0];
        loc_res[3] += var[1];
    }

    end = std::chrono::high_resolution_clock::now();
    del_results.emplace_back(loc_res[2], loc_res[3], (std::chrono::duration<double, s
    //printf("\nExecuting input file 2 took %f seconds.\n", (std::chrono::duration<do
    //printf("Succeeded operations: %u, Failed operations: %u\n\n", all_res[iteratio
    if (loc_res[2] + loc_res[3] != delete_tuples.size()) {
        printf("An error happened. The program handled %d records, when there are su
               delete_tuples.size());
    }

}
/////////////////////////////////////////////////////////////////////////////////////
////DELETE PART END/////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////

start = std::chrono::high_resolution_clock::now();
share_size = static_cast<int>(ceil((double)extr_tuples.size() / (double)num_thread))

printf("\nNumber of additional operations: %lu\n", extr_tuples.size());

for (int i = 0; i < num_thread; i++)
{
    auto *thread_arg = new thread_arg_t;

    thread_arg->tr_idx = i;

    thread_arg->start_i = i * share_size;
```

```
        if (i != num_thread -1)
        {
            thread_arg->end_i = thread_arg->start_i + share_size;
        } else {
            thread_arg->end_i = (int)extr_tuples.size();
        }

        pthread_create(&threads[i], NULL, execute_extra_load, thread_arg);
    }

    for (size_t i = 0; i < num_thread; i++) {
        pthread_join(threads[i], &ret[i]);
        auto *var = reinterpret_cast<unsigned int *>(ret[i]);
        for (int j = 0; j < 8; j++)
            loc_res[j+4] += var[j];
    }
    free(threads);

    end = std::chrono::high_resolution_clock::now();
    extr_results.emplace_back(loc_res[4], loc_res[5], loc_res[6], loc_res[7], loc_res[8]

    printf("Read/Update iteration %d: INS_S: %d | INS_F: %d | READ_S: %d | READ_F: %d | U
           iteration, loc_res[4], loc_res[5], loc_res[6], loc_res[7], loc_res[8], loc_res

    if (loc_res[4] + loc_res[5] + loc_res[6] + loc_res[7] + loc_res[8] + loc_res[9] + loc
        printf("An error happened. The program handled %d extra records, when there are s
               extr_tuples.size());
    }

    return;
}

/*
 * run_rdtsc_benchmark() - This function runs the RDTSC benchmark which is a high
 *                         contention insert-only benchmark
 *
 * Note that key num is the total key num
 */
void run_rdtsc_benchmark(int index_type, int thread_num, int key_num)
{
    Index<keytype, keycomp> *idx = getInstance<keytype, keycomp>(index_type, key_type);

    auto func = [idx, thread_num, key_num](uint64_t thread_id, bool) {
        size_t key_per_thread = key_num / thread_num;

        threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);
```

```
        uint64_t *values = new uint64_t[key_per_thread];

        int gc_counter = 0;
        for(size_t i = 0;i < key_per_thread;i++) {
            // Note that RDTSC may return duplicated keys from different cores
            // to counter this we combine RDTSC with thread IDs to make it unique
            // The counter value on a single core is always unique, though
            uint64_t key = (Rdtsc() << 6) | thread_id;
            values[i] = key;
            //fprintf(stderr, "%lx\n", key);
            idx->insert(key, reinterpret_cast<uint64_t>(values + i), ti);
            gc_counter++;
            if(gc_counter % 4096 == 0) {
                ti->rcu_quiesce();
            }
        }

        ti->rcu_quiesce();

        delete [] values;

        return;
    };

    if(numa == true) {
        PCM_NUMA::StartNUMAMonitor();
    }

    double start_time = get_now();
    StartThreads(idx, thread_num, func, false);
    double end_time = get_now();

    if(numa == true) {
        PCM_NUMA::EndNUMAMonitor();
    }

    // Only execute consolidation if BwTree delta chain is used
#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
    idx->AfterLoadCallback();
#endif

    double tput = key_num * 1.0 / (end_time - start_time) / 1000000; //Mops/sec
    std::cout << "insert " << tput << "\n";

    return;
}
```

```
int main(int argc, char *argv[])
{
    if (argc < 6) {
        std::cout << "Usage:\n";
        std::cout << "1. workload type: r, u, d,\n";
        std::cout << "2. key distribution: rand, mono\n";
        std::cout << "3. index type: bwtree skiplist masstree btreeolc btreertm\n";
        std::cout << "4. number of keys to insert in millions (10, 50, 100)\n";
        std::cout << "5. number of extra operations to execute in millions (10, 50, 100,
        std::cout << "6. number of threads (integer)\n";
        std::cout << "7 extra settings\n";
        std::cout << "   --hyper: Whether to pin all threads on NUMA node 0\n";
        std::cout << "   --mem: Whether to monitor memory access\n";
        std::cout << "   --numa: Whether to monitor NUMA throughput\n";
        std::cout << "   --insert-only: Whether to only execute insert operations\n";

        return 1;
    }

    // Then read the workload type
    int wl;
    if (strcmp(argv[1], "r") == 0) {
        wl = WORKLOAD_R;
    } else if (strcmp(argv[1], "u") == 0) {
        wl = WORKLOAD_U;
    } else if (strcmp(argv[1], "d") == 0) {
        wl = WORKLOAD_D;
        del = true;
    } else if (strcmp(argv[1], "ru") == 0) {
        wl = WORKLOAD_RU;
    } else {
        fprintf(stderr, "Unknown workload: %s\n", argv[1]);
        exit(1);
    }

    // Then read key type
    int kt;
    if (strcmp(argv[2], "rand") == 0) {
        kt = RAND_KEY;
    } else if (strcmp(argv[2], "mono") == 0) {
        kt = MONO_KEY;
    } else if (strcmp(argv[2], "rdtsc") == 0) {
        kt = RDTSC_KEY;
    } else {
        fprintf(stderr, "Unknown key type: %s\n", argv[2]);
```

```
    exit(1);
}

int index_type;
if (strcmp(argv[3], "bwtree") == 0)
    index_type = TYPE_BWTREE;
else if (strcmp(argv[3], "masstree") == 0)
    index_type = TYPE_MASSTREE;
else if (strcmp(argv[3], "btreeolc") == 0)
    index_type = TYPE_BTREEOLC;
else if (strcmp(argv[3], "skiplist") == 0)
    index_type = TYPE_SKIPLIST;
else if (strcmp(argv[3], "btreertm") == 0)
    index_type = TYPE_BTREERTM;
else if (strcmp(argv[3], "none") == 0)
    // This is a special type used for measuring base cost (i.e.
    // only loading the workload files but do not invoke the index)
    index_type = TYPE_NONE;
else {
    fprintf(stderr, "Unknown index type: %d\n", index_type);
    exit(1);
}

// Read amount insert ops in million
int insert_ops;
if (strcmp(argv[4], "10") == 0) {
    insert_ops = 10;
} else if (strcmp(argv[4], "50") == 0) {
    insert_ops = 50;
} else if (strcmp(argv[4], "100") == 0) {
    insert_ops = 100;
} else {
    fprintf(stderr, "Unknown amount of inserts: %s\n", argv[4]);
    exit(1);
}

// Read amount extra ops in million
int extra_ops;
if (strcmp(argv[5], "10") == 0) {
    extra_ops = 10;
} else if (strcmp(argv[5], "50") == 0) {
    extra_ops = 50;
} else if (strcmp(argv[5], "100") == 0) {
    extra_ops = 100;
} else if (strcmp(argv[5], "200") == 0) {
    extra_ops = 200;
} else {
```

```
        fprintf(stderr, "Unknown amount of extras: %s\n", argv[5]);
        exit(1);
    }

    // Then read number of threads using command line
    int num_thread = atoi(argv[6]);
    if(num_thread < 1 || num_thread > 96) {
        fprintf(stderr, "Do not support %d threads\n", num_thread);
        exit(1);
    } else {
        fprintf(stderr, "Number of threads: %d\n", num_thread);
    }

    // Then read all remianing arguments
    int repeat_counter = 1;
    char **argv_end = argv + argc;
    for(char **v = argv + 7;v != argv_end;v++) {
        if(strcmp(*v, "--hyper") == 0) {
            // Enable hyoerthreading for scheduling threads
            hyperthreading = true;
        } else if(strcmp(*v, "--mem") == 0) {
            // Enable memory bandwidth measurement
            memory_bandwidth = true;
        } else if(strcmp(*v, "--numa") == 0) {
            numa = true;
        } else if(strcmp(*v, "--insert-only") == 0) {
            insert_only = true;
        } else if(strcmp(*v, "--repeat") == 0) {
            // If we repeat, then exec() will be called for 5 times
            repeat_counter = 5;
        } else if(strcmp(*v, "--max-init-key") == 0) {
            max_init_key = atoll(*(v + 1));
            if(max_init_key <= 0) {
                fprintf(stderr, "Illegal maximum init keys: %ld\n", max_init_key);
                exit(1);
            }

            // Ignore the next argument
            v++;
        } else {
            fprintf(stderr, "Unknown switch: %s\n", *v);
            exit(1);
        }
    }

    if(max_init_key != -1) {
        fprintf(stderr, "Maximum init keys: %ld\n", max_init_key);
```

```
        fprintf(stderr, "  NOTE: Memory is not affected in this case\n");
    }

#ifdef COUNT_READ_MISS
    fprintf(stderr, "  Counting read misses\n");
#endif

#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
    fprintf(stderr, "  BwTree will considate after insert phase\n");
#endif

#ifdef USE_TBB
    fprintf(stderr, "  Using Intel TBB to run concurrent tasks\n");
#endif

#ifdef INTERLEAVED_INSERT
    fprintf(stderr, "  Interleaved insert\n");
#endif

#ifdef BWTREE_COLLECT_STATISTICS
    fprintf(stderr, "  BwTree will collect statistics\n");
#endif

    fprintf(stderr, "Leaf delta chain threshold: %d; Inner delta chain threshold: %d\n",
            LEAF_DELTA_CHAIN_LENGTH_THRESHOLD,
            INNER_DELTA_CHAIN_LENGTH_THRESHOLD);

#ifndef BWTREE_USE_MAPPING_TABLE
    fprintf(stderr, "  BwTree does not use mapping table\n");
  if(wl != WORKLOAD_C) {
    fprintf(stderr, "Could only use workload C\n");
    exit(1);
  }

  if(index_type != TYPE_BWTREE) {
    fprintf(stderr, "Could only use BwTree\n");
    exit(1);
  }
#endif

#ifndef BWTREE_USE_CAS
    fprintf(stderr, "  BwTree does not use CAS\n");
#endif

#ifndef BWTREE_USE_DELTA_UPDATE
    fprintf(stderr, "  BwTree does not use delta update\n");
  if(index_type != TYPE_BWTREE) {
```

```
    fprintf(stderr, "Could only use BwTree\n");
  }
#endif

#ifdef USE_OLD_EPOCH
    fprintf(stderr, "  BwTree uses old epoch\n");
#endif

    // If we do not interleave threads on two sockets then this will be printed
    if(hyperthreading == true) {
        fprintf(stderr, "  Hyperthreading for thread 10 - 19, 30 - 39\n");
    }

    if(repeat_counter != 1) {
        fprintf(stderr, "  Repeat for %d times (NOTE: Memory number may not be correct)\n
                repeat_counter);
    }

    if(memory_bandwidth == true) {
        if(geteuid() != 0) {
            fprintf(stderr, "Please run the program as root in order to measure memory ba
            exit(1);
        }

        fprintf(stderr, "  Measuring memory bandwidth\n");

        PCM_memory::InitMemoryMonitor();
    }

    if(numa == true) {
        if(geteuid() != 0) {
            fprintf(stderr, "Please run the program as root in order to measure NUMA oper
            exit(1);
        }

        fprintf(stderr, "  Measuring NUMA operations\n");

        // Call init here to avoid calling it mutiple times
        PCM_NUMA::InitNUMAMonitor();
    }

    if(insert_only == true) {
        fprintf(stderr, "Program will exit after insert operation\n");
    }

    tuples.reserve(100000000);
    extr_tuples.reserve(200000000);
```

```
fprintf(stderr, "  BTree element pair count: %lu\n",
        (uint64_t)btreeolc::BTreeLeaf<uint64_t, uint64_t>::maxEntries);

// If the key type is RDTSC we just run the special function
if(kt != RDTSC_KEY) {

    load(wl, kt, index_type, insert_ops, extra_ops);
    printf("Finished loading workload file (mem = %lu)\n", MemUsage());

    prepare_server(wl, index_type, num_thread, repeat_counter);

    if(index_type != TYPE_NONE) {
        // Then repeat executing the same workload

        while(iteration < repeat_counter) {

            exec(num_thread);
            iteration++;
            printf("\nFinished iteration %d of %d\n\n", iteration, repeat_counter);
            printf("Finished running benchmark (mem = %lu)\n", MemUsage());
        }
    } else {
        fprintf(stderr, "Type None is selected - no execution phase\n");
    }

    for (int i = 0; i < repeat_counter; i++){
        if (!insert_only && delete_tuples.empty()){
            printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec - part 2: I
                   (i+1), init_results[i].init_s, init_results[i].init_f, init_resul
                   extr_results[i].insert_s, extr_results[i].insert_f, extr_results[
        } else if (!delete_tuples.empty()){
            printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec - part 2: D
                   (i+1), init_results[i].init_s, init_results[i].init_f, init_resul
                   extr_results[i].insert_s, extr_results[i].insert_f, extr_results[
        } else {
            printf("Iter%d part 1: INS_S: %d | INS_F: %d | took %.2f sec\n",
                   (i+1), init_results[i].init_s, init_results[i].init_f, init_resul
        }
    }

} else {
    fprintf(stderr, "Running RDTSC benchmark...\n");
    run_rdtsc_benchmark(index_type, num_thread, 50 * 1000 * 1000);
}
```

```
    for (int i = 0; i < num_thread; i++)
        close(_sockets[i]);

    exit_cleanup();

    return 0;
}
```

Appendix C OpenBw-tree server side code

serverSide.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "./pcm/pcm-memory.cpp"
//#include "./pcm/pcm-numa.cpp"
#include "./papi_util.cpp"

#include "microbench.h"

#include <cstring>
#include <cctype>
#include <atomic>

#define COUNT_READ_MISS

thread_local long skiplist_steps = 0;
std::atomic<long> skiplist_total_steps;

//#define USE_TBB

#ifdef USE_TBB
#include "tbb/tbb.h"
#endif

// Enable this if you need pre-allocation utilization
//#define BWTREE_CONSOLIDATE_AFTER_INSERT

#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
#ifdef USE_TBB
  #warning "Could not use TBB and BwTree consolidate together"
  #endif
#endif

#ifdef BWTREE_COLLECT_STATISTICS
```

```
#ifdef USE_TBB
  #warning "Could not use TBB and BwTree statistics together"
  #endif
#endif

// Whether insert interleaves
//#define INTERLEAVED_INSERT

// Whether read operatoin miss will be counted
//#define COUNT_READ_MISS

typedef uint64_t keytype;
typedef std::less<uint64_t> keycomp;

static const uint64_t key_type=0;
static const uint64_t value_type=1; // 0 = random pointers, 1 = pointers to keys

extern bool hyperthreading;

// This is the flag for whather to measure memory bandwidth
static bool memory_bandwidth;
// Whether to measure NUMA Throughput
static bool numa;
// Whether we only perform insert
static bool insert_only;

bool del;

// We could set an upper bound of the number of loaded keys
static int64_t max_init_key = -1;

static std::vector<int> _listeners;

#include "util.h"

int setup_server(u_int16_t port)
{
    int _socket;

    struct sockaddr_in server;
    int iSetOption = 1;

    _socket = socket(AF_INET, SOCK_STREAM, 0);
    setsockopt(_socket, SOL_SOCKET, SO_REUSEADDR, (char*)&iSetOption, sizeof(iSetOption))
    if (_socket < 0) {
        perror("Cannot create a socket"); exit(1);
    }
```

```
    bzero((char *) &server, sizeof(server));
    u_int16_t server_port = port;
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(port);
    // The bind function associates a local address with a socket.
    if (bind(_socket, (struct sockaddr *) &server,
             sizeof(server)) < 0) {
        perror("ERROR on binding.");
    }
    // The listen function places a socket in a state in which it is listening for an inc
    listen(_socket,5);

    return _socket;
}


int setup_listener(int _socket)
{
    struct sockaddr_in client_addr;
    socklen_t clilen;
    clilen = sizeof(client_addr);
    // The accept function permits an incoming connection attempt on a socket.
    int new_socket = accept(_socket, (struct sockaddr *) &client_addr, &clilen);
    if (new_socket < 0)
    {
        perror("ERROR on accept\n");
    }
    return new_socket;
}

void handle_info(char *buffer, int _listener)
{
    long recv = read(_listener, buffer, 512);
    if (recv < 0) {
        perror("ERROR reading from socket\n");
    } else {
        recv = write(_listener, "OK\0", 3);
        if (recv < 0) {
            perror("ERROR writing to socket\n");
        }
    }
}

void handle_socket_response(long res)
{
    if (res < 0){
```

```
        printf("An error occurred while sending or receiving data from the client. Abort
        exit(-1);
    }
};

//==============================================================
// EXEC
//==============================================================
inline void exec(int wl, int index_type, int num_thread)
{

    Index<keytype, keycomp> *idx = getInstance<keytype, keycomp>(index_type, key_type);

#ifdef USE_TBB
    tbb::task_scheduler_init init{num_thread};

  std::atomic<int> next_thread_id;
  next_thread_id.store(0);

  auto func = [idx, &init_keys, &values, &next_thread_id](const tbb::blocked_range<size_
    size_t start_index = r.begin();
    size_t end_index = r.end();

    threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);

    int thread_id = next_thread_id.fetch_add(1);
    idx->AssignGCID(thread_id);

    int gc_counter = 0;
    for(size_t i = start_index;i < end_index;i++) {
      idx->insert(init_keys[i], values[i], ti);
      gc_counter++;
      if(gc_counter % 4096 == 0) {
        ti->rcu_quiesce();
      }
    }

    ti->rcu_quiesce();
    idx->UnregisterThread(thread_id);

    return;
  };

  idx->UpdateThreadLocal(num_thread);
  tbb::parallel_for(tbb::blocked_range<size_t>(0, count), func);
  idx->UpdateThreadLocal(1);
#else
```

```
    auto func = [idx, num_thread, index_type](uint64_t thread_id, bool) {

        threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);

        int _listener = _listeners[thread_id];

        char buffer[512];
        std::string k_s, v_s;

        handle_info(buffer, _listener);
        int share_size = atoi(buffer);
        //printf("My share size: %d\n", share_size);

        int gc_counter = 0;
//#ifdef INTERLEAVED_INSERT
//        for(size_t i = thread_id;i < total_num_key;i += num_thread) {
//#else
        for(int i = 0;i < share_size;i++) {
//#endif
            handle_socket_response(read(_listener, buffer, 512));

            istringstream iss(buffer);
            iss >> k_s >> v_s;

            keytype key = std::stoull(k_s, 0, 10);
            uint64_t value = std::stoull(v_s, 0, 10);

#ifdef BWTREE_USE_DELTA_UPDATE
            //printf("Want to insert key %li and val %li\n", (keytype) key, value);
            bool res = idx->insert((keytype)key, value, ti);
#else
            bool res = idx->insert_bwtree_fast((keytype)key, value);
#endif
            if (res)
                handle_socket_response(write(_listener, (to_string(BWTREE_STORED) + "\0"
            else
                handle_socket_response(write(_listener, (to_string(BWTREE_NOTSTORED) + "

            gc_counter++;
            if(gc_counter % 4096 == 0) {
                ti->rcu_quiesce();
            }
            if(gc_counter % 500000 == 0){
                printf("Thread handled %.2f%% of its insert-operations, %d of %d.\n", ((
            }
        }
```

```
        ti->rcu_quiesce();
        return;
    };
////////////////////////////////////////////////////////////////////////////////////
    /*
    if(memory_bandwidth == true) {
        PCM_memory::StartMemoryMonitor();
    }
    if(numa == true) {
        PCM_NUMA::StartNUMAMonitor();
    }*/
    //auto start = std::chrono::high_resolution_clock::now();

    StartThreads(idx, num_thread, func, false);

    //auto end = std::chrono::high_resolution_clock::now();
    //printf("\nExecuting input file 1 took %f ms.\n\n", std::chrono::duration<double, st
    /*
    if(memory_bandwidth == true) {
        PCM_memory::EndMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::EndNUMAMonitor();
    }*/
#endif

    // Only execute consolidation if BwTree delta chain is used
#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
    fprintf(stderr, "Starting consolidating delta chain on each level\n");
  idx->AfterLoadCallback();
#endif

    // If the workload only executes load phase then we return here
    if(insert_only) {
        delete idx;
        return;
    }
    ////////////////////////////////////////////////////////////////////////////////
    ////DELETE PART START////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////

    if (del){
        auto del_func = [num_thread, idx](uint64_t thread_id, bool) {

            int _listener = _listeners[thread_id];
```

```
char buffer[512];
std::string o_s, k_s, v_s;

handle_info(buffer, _listener);
int share_size = atoi(buffer);
//printf("My second share size: %d\n", share_size);

threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);

int counter = 0;
for(int i = 0;i < share_size;i++) {

    handle_socket_response(read(_listener, buffer, 512));

    istringstream iss(buffer);
    iss >> o_s >> k_s >> v_s;

    int op = std::stoi(o_s);
    keytype key = std::stoull(k_s, 0, 10);
    uint64_t value = (v_s == "" ? 0 : std::stoull(v_s, 0, 10));
    bool res;

    if (op == OP_REMOVE){
        res = idx->remove(key, value, ti);
    } else {
        bool res = false;
        printf("Unknown operation %d\n", op);
    }

    if (res)
        handle_socket_response(write(_listener, (to_string(BWTREE_DELETED) +
    else
        handle_socket_response(write(_listener, (to_string(BWTREE_NOTDELETED)

    counter++;
    if(counter % 4096 == 0) {
        ti->rcu_quiesce();
    }
    if(counter % 500000 == 0){
        printf("Thread handled %.2f%% of its delete-operations, %d of %d.\n"
    }
}

// Perform GC after all operations
ti->rcu_quiesce();

return;
```

```
    };
    /*
    if(memory_bandwidth == true) {
        PCM_memory::StartMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::StartNUMAMonitor();
    }
    */
    //start = std::chrono::high_resolution_clock::now();
    StartThreads(idx, num_thread, del_func, false);
    //end = std::chrono::high_resolution_clock::now();
    //printf("\nExecuting input file 2 took %f seconds.\n", (std::chrono::duration<do
    /*
    if(memory_bandwidth == true) {
        PCM_memory::EndMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::EndNUMAMonitor();
    }*/
}

/////////////////////////////////////////////////////////////////////////////////////
////DELETE PART END/////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////
// This is used to count how many read misses we have found
std::atomic<size_t> read_miss_counter{}, read_hit_counter{};
read_miss_counter.store(0UL);
read_hit_counter.store(0UL);

auto func2 = [num_thread, idx, &read_miss_counter, &read_hit_counter](uint64_t threa

    int _listener = _listeners[thread_id];

    char buffer[512];
    std::string o_s, k_s, v_s;

    handle_info(buffer, _listener);
    int share_size = atoi(buffer);
    //printf("My second share size: %d\n", share_size);

    std::vector<uint64_t> v;
    v.reserve(10);

    threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);
```

```
        int counter = 0;
        for(int i = 0;i < share_size;i++) {

            handle_socket_response(read(_listener, buffer, 512));

            istringstream iss(buffer);
            iss >> o_s >> k_s >> v_s;

            int op;
            keytype key;
            uint64_t value;

            try {
                op = std::stoi(o_s);
                key = std::stoull(k_s, 0, 10);
                value = (v_s == "" ? 0 : std::stoull(v_s, 0, 10));
            }
            catch (std::invalid_argument) {
                op = 10;
            }

            if (op == OP_INSERT) { //INSERT
                if (idx->insert(key, value, ti))
                    handle_socket_response(write(_listener, (to_string(BWTREE_STORED) + "
                else
                    handle_socket_response(write(_listener, (to_string(BWTREE_NOTSTORED)

            } else if (op == OP_READ) { //READ
                v.clear();

#ifdef BWTREE_USE_MAPPING_TABLE
                idx->find(key, &v, ti);
#else
                idx->find_bwtree_fast(key, &v);
#endif

                // If we count read misses then increment the
                // counter here if the vetor is empty
#ifdef COUNT_READ_MISS
                if(v.size() == 0UL){
                    read_miss_counter.fetch_add(1);
                    handle_socket_response(write(_listener, (to_string(BWTREE_NOTREAD) +
                }
                else{
                    read_hit_counter.fetch_add(1);
                    handle_socket_response(write(_listener, (to_string(BWTREE_READ) + "\
```

```
                }
#endif
            } else if (op == OP_UPSERT) { //UPDATE

                if(!idx->upsert(key, value, ti)) // The negation is correct here since th
                    handle_socket_response(write(_listener, (to_string(BWTREE_UPDATED) +
                else
                    handle_socket_response(write(_listener, (to_string(BWTREE_NOTUPDATED)

            } else if (op == OP_REMOVE){
                if(idx->remove(key, value, ti))
                    handle_socket_response(write(_listener, (to_string(BWTREE_DELETED) +
                else
                    handle_socket_response(write(_listener, (to_string(BWTREE_NOTDELETED)
            } else {
                handle_socket_response(write(_listener, (to_string(BWTREE_UNKNOWNOP) + "\
                printf("Unknown operation %d\n", op);
            }

            counter++;
            if(counter % 4096 == 0) {
                ti->rcu_quiesce();
            }
            if(counter % 500000 == 0){
                printf("Thread handled %.2f%% of its extra-operations, %d of %d.\n", ((fl
            }
        }

        // Perform GC after all operations
        ti->rcu_quiesce();

        return;
    };
    /*
    if(memory_bandwidth == true) {
        PCM_memory::StartMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::StartNUMAMonitor();
    }
    */
    //start = std::chrono::high_resolution_clock::now();
    StartThreads(idx, num_thread, func2, false);
    //end = std::chrono::high_resolution_clock::now();
    //printf("\nExecuting input file 2 took %f seconds.\n", (std::chrono::duration<double
    /*
```

```
    if(memory_bandwidth == true) {
        PCM_memory::EndMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::EndNUMAMonitor();
    }
    */
    // Print out how many reads have missed in the index (do not have a value)
#ifdef COUNT_READ_MISS
    fprintf(stderr,
          "  Read misses: %lu; Read hits: %lu\n",
          read_miss_counter.load(),
          read_hit_counter.load());
#endif

    delete idx;

    return;
}

void process(int _socket){


    /////////////////////////////////////////////////////////////////////////////////////
    printf("Waiting for client to connect ... \n");
    int _listener = setup_listener(_socket);

    char buffer[512];

    handle_info(buffer, _listener);
    int wl = atoi(buffer);

    handle_info(buffer, _listener);
    int index_type = atoi(buffer);

    handle_info(buffer, _listener);
    int num_thread = atoi(buffer);

    handle_info(buffer, _listener);
    insert_only = atoi(buffer);

    handle_info(buffer, _listener);
    numa = atoi(buffer);

    handle_info(buffer, _listener);
    hyperthreading = atoi(buffer);
```

```
    handle_info(buffer, _listener);
    memory_bandwidth = atoi(buffer);

    handle_info(buffer, _listener);
    int repeat_counter = atoi(buffer);

    handle_info(buffer, _listener);
    del = atoi(buffer);

    for (int i = 0; i < num_thread; i++)
        _listeners.emplace_back(setup_listener(_socket));

    printf("\nReceived: %d %d %d %d %d %d %d %d %d\n", wl, index_type, num_thread, insert

    close(_listener);
    ////////////////////////////////////////////////////////////////////////////////////

    while (repeat_counter > 0){
        auto start = std::chrono::high_resolution_clock::now();

        exec(wl, index_type, num_thread);

        auto end = std::chrono::high_resolution_clock::now();
        //printf("\nThe whole thing took %f seconds.\n\n", (std::chrono::duration<double
        repeat_counter--;
        std::cout << "\n";
    }

    for (int i = 0; i < _listeners.size(); i++)
        close(_listeners[i]);

}

int main()
{
    int _socket = setup_server(11211);
    for (int i = 0; i < 250; i++){
        process(_socket);
        printf("done it %d\n", i);
    }
    close(_socket);
}
```

Appendix D Python script code

main.py

```
def main():

    ##########################################################################################
    workload_path = '/export/pzirpins/MasterOppgave/clion/CLionProjects/index-microbench-
    additions = ['10mil/10mil/read_update_rand.dat', '50mil/50mil/read_update_rand.dat',

    loads = [10000000, 50000000, 100000000]
    for i in range(3):
        print('Starting with files ' + str(i + 1) + ' ...')
        r_file = open(workload_path + 'seq_read_' + str(loads[i]) + '.dat', 'w+')
        u_file = open(workload_path + 'seq_upd_' + str(loads[i]) + '.dat', 'w+')
        for j in range(loads[i]):

            r_file.write('READ ' + str(j) + '\n')
            u_file.write('UPDATE ' + str(j) + '\n')

        print('Finished with file r+u' + str(i + 1) + ' ...')

        r_file.close()
        u_file.close()
    for i in range(3):
        file = open(workload_path + 'seq_read_update_' + str(loads[i]) + '.dat', 'w+')
        current_file = open(workload_path + additions[i], 'r')

        print('Starting with file r+u' + str(i+1) + ' ...')
        index = 0

        with current_file as open_file_object:
            for line in open_file_object:
                op = line.replace('\n', '').split(' ')[0]

                file.write(op + ' ' + str(index) + '\n')
                index += 1

        file.close()
        current_file.close()


main()
```